Київський національний університет ім. Т. Шевченка
Факультет комп’ютерних наук та кібернетики

Терещенко В.М.

Аналіз методів розв’язання оптимізаційних задач обчислювальної геометрії

Навчальний посібник

з виконання лабораторних робіт з курсу «Обчислювальна геометрія комп’ютерна графіка»

для студентів факультету комп’ютерних наук та кібернетики

Київ 2020

Вступ

Інженери при розробці сучасних інформаційних та комп'ютерних технологій досить часто стикаються із технічними чи технологічними проблемами, які зводяться , так чи інакше, до задач комбінаторної геометрії (computational geometry). Розв'язання задач обчислювальної геометрії потребує пошуку оптимальних рішень так, як вони визначені на великих множинах даних, елементи яких являють собою складні геометричні об'єкти (точки, відрізки, полігони, многогранники, розбиття простору та інші складні геометричні об'єкти). Такий пошук не завжди приносить бажані результати, оскільки потрібен не просто розв'язок геометричної задачі, але й щоб він задовольняв оптимізаційні вимоги такі, як час виконання та об'єм використаної пам'яті. Іншими словами, необхідно знайти рішення, яке б дозволяло отримати найефективніший із можливих розв'язків задачі. Таким чином на відміну від класичної математики нам необхідно знайти не лише розв'язок задачі в принципі, а знайти найефективніший розв'язок. У цьому й полягає складність розв'язання задач обчислювальної геометрії. Тому ідеї , підходи, методи які дають ефективні рішення задач обчислювальної геометрії є досить актуальними для фахівців в області ІКТ.
У посібнику, який ми пропонуємо викладені деякі ефективні підходи та алгоритми розв'язання задач одного із важливих на практиці класу задач обчислювальної геометрії - оптимізація. Задачі ці пов'язані із оптимальним вписанням та описанням різних фігур (коло, многокутник, еліпс, тощо). Список задач цього класу представлено нижче, таб. 1.

№	код	Назва та постановка задачі	Оцінки складності
		A	
1	AO1	Задача лінійного програмування На заданій множині точок розв'язати задачу лінійного програмування.	$O(n)$
		Задачі на охоплення	
2	ОМК(АО2)	Найменше коло На заданій множині S із n точок на площині побудувати коло найменшого радіусу, яке б охоплювало S.	$O(n^2)$
3	ОМТ(АО3)	Трикутник найменшої площі На заданій множині S із n точок на площині побудувати побудувати трикутник найменшої площі, який би охоплював S.	$O(n^2)$
4	ОМП(АО4)	Прямокутник найменшої площі На заданій множині S із n точок на площині побудувати прямокутник найменшої площі, який би охоплював S.	$O(n^2)$
5	ОМЕ(ВО32)	Найменший охоплюючий еліпс. На заданій множині S із n точок на площині побудувати еліпс найменшої площі, який би охоплював S.	$O(n^2)$
		Задачі на вписання	
6	ВМКО(АО21)	Найбільше коло вписане в опуклу оболонку	$O(n^2)$

		На заданій множині S із n точок на площині побудувати опуклу оболонку і вписати в неї коло найбільшого радіусу.	
7	ВМТО (АО31)	Трикутник найбільшої площі вписаний в о.о. На заданій множині S із n точок на площині побудувати опуклу оболонку і вписати в неї трикутник найбільшої площі.	$O(n^2)$
8	ВМПО(АО41)	Прямокутник найбільшої площі впис. в о.о. На заданій множині S із n точок на площині побудувати опуклу оболонку і вписати в неї прямокутник найбільшої площі.	$O(n^2)$
9	ВМЕО (ВО4)	Найбільший еліпс вписаний в опуклу оболонку На заданій множині S із n точок на площині побудувати опуклу оболонку і вписати еліпс максимальної площі.	$O(n^2)$
10	ВМКЗ (ВО21)	Найбільше коло вписане в зірковий многокутник В заданий зірковий n-кутник вписати коло найбільшого радіусу.	
11	ВМЕЗ (ВО21е)	Найбільше еліпс вписаний в зірковий многокутник В заданий зірковий n-кутник вписати еліпс найбільшої площі.	
12	ВМПЗ (АО41п)	Прямокутник найбільшої площі впис. в з.м. В заданий зірковий n-кутник вписати прямокутник найбільшої площі.	$O(n^2)$
13	ВКМ1 (ВО10)	Найбільше коло в *n*- кутнику На площині задано коло та n точок. Вписати в коло *n*- кутник, сторони якого проходять через задані точки.	
14	ВКМ2 (ВО11)	Найбільше коло в *n*- кутнику з паралельними сторонами В задане в коло вписати *n*- кутник, *k* сторін якого (необов'язково сусідніх) проходять через *k* заданих точок, а решта *n* - *k* сторін паралельні заданим.	
		Задачі на розташування	
15	РМК (АО5)	Найбільше порожнє коло На заданій обмеженій (прямокутником чи опуклою оболонкою) множині S із *n* точок на площині побудувати коло найбільшого радіусу, яке б не містило всередині жодної точки множини S.	$O(n^2)$
16	РМП (АО6)	Найбільший порожній прямокутник . На заданій обмеженій (прямокутником чи опуклою оболонкою) множині S із *n* точок на площині побудувати прямокутник найбільшої площі, який би не містив всередині жодної точки множини S.	$O(n^2)$
17	РМЕ (АО7)	Найбільший порожній еліпс. На заданій обмеженій (прямокутником чи опуклою оболонкою) множині S із *n* точок на площині побудувати еліпс найбільшої площі, який би не містив всередині жодної точки множини S.	$O(n^2)$

Таблиця 1. Список задач оптимізації

Всі ці задачі мають велике практичне застосування. Так, зокрема, задачу про мінімальне охоплююче коло (ОМК) можна використати, щоб знайти мінімальну допустиму потужність радіопередавача і його місце розташування для забезпечення зв'язком певного населеного пункту; задачу про прямокутник найменшої площі (ОМП), що охоплює множину точок, можна використовувати для оптимізації регіонального пошуку, а задачу про пошук найбільшого порожнього кола (РМК) для пошуку місця розташування якогось шкідливого підприємства, яке має бути якнайдалі від населених пунктів, тощо.

АЛГОРИТМІЧНІ ІНСТРУМЕНТИ

Спочатку ми розглянемо підходи та основні алгоритмічні інструменти (стратегії, структури даних та інтегральні геометричні структури), які дозволяють розробляти ефективні алгоритми розв’язування задач обчислювальної геометрії і утому числі, задач оптимізації, поданих у табл.1. Які ж існують підходи до розробки ефективних алгоритмів розв’язання задач обчислювальної геометрії перш за усе задачі, які мають своє безпосереднє застосування в інженерних чи наукових проектах?

Як показує досвід співпраці із відомими ІТ компаніями, найбільш поширений такий підхід:

- якщо немає обмежень по часу
  - розв'язання «в лоб» або перебірний алгоритм;
- якщо є обмеження по часу:
  - пошук існуючих популярних методів розв'язання типових задач;
- якщо задача не типова - пошук (інтернет пошук) ідей розв'язання схожих задач в різних джерелах інформації (журнальні статті, статті конференцій, інтернет);

Якщо постановка та розв’язання такої задачі і можна знайти в літературі, проте це не завжди вдається зробити. Це не буде власне постановка нашої задачі, а скоріше щось схоже. Окрім цього, зазвичай ідею розв’язання задачі можна знайти в інформаційних джерелах, але при цьому у більшості випадків деталі і проблемні моменти реалізації, як правило, приховуються. І тому досить часто приходить розчарування, від такого підходу. Інший підхід пов'язаний із застосуванням існуючих узагальнених алгоритмічних стратегій, проте це теж не завжди дозволяє адаптувати ефективний алгоритм розв’язання поставленої задачі.

Ми пропонуємо інший підхід для розробки ефективних методів, який дозволяє розробляти ефективні прямі чи узагальнені алгоритми, і базується на:

1) аналізі задачі, природи, структури та формату вхідних даних та самого результату;

2) встановленні межі ефективності розв’язання задачі (нижні, верхні та оптимальні оцінки складності);

3) визначенні основних властивостей вхідних та вихідних даних задачі;

4) визначенні інструментів та ефективної стратегії розв'язання задачі:

- розробка прямого алгоритму розв’язання;
- застосування відомих узагальнених методів та підходів розв’язання задачі;

5) створенні нових чи використання існуючих структур даних чи геометричних структур, які дозволяють отримати результати відповідно до обраної стратегії.

Розглянемо застосування цього підходу на прикладі розв’язання задач геометричного пошуку та побудови опуклої оболонки.

Задача 1(задача локалізації точки). *Нехай задане n – вершинне розбиття G(V,E) евклідового простору E^d (у випадку двовимірного простору – планарний граф) і точка Z. Визначити область розбиття, яка містить точку Z (Локалізувати точку Z на заданому розбитті) (рис.1).*

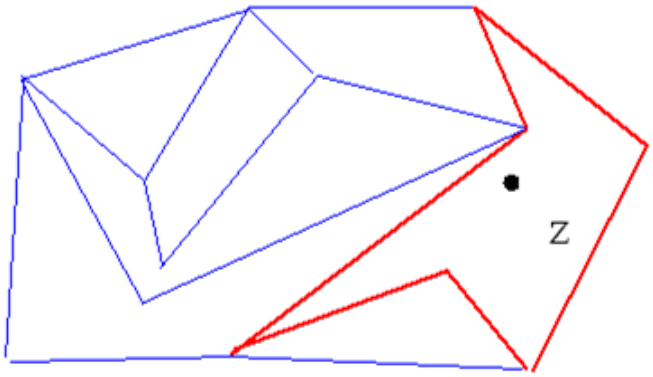


Рис.1. Аналіз

Аналіз задачі.

1. З'ясуємо перш за все клас задач, до якого відноситься наша задача. Це клас геометричного пошуку (ГП), підклас – задача локалізації точки.
2. Аналіз входу : *n*- вершинний планарний граф(задається у вигляді G(V,E)) і точка *Z*. На виході має бути многокутник (у вигляді списку вершин чи ребер) , який містить *Z*.
3. Аналіз розв'язання задачі у випадку звичайного перебору. Підрахуємо кількість многокутників (буде порядку $O(n)$) та кількість сторін многокутника (буде $O(n)$).
4. **Процедура** (перебір): $\forall$ *i*-го *n*-кутника перевірити належність Z. Загальний час пошуку буде порядку $O(n \times n)$ у найгіршому випадку.

Постає питання чи можна зменшити час розв'язання задачі? Щоб відповісти на запитання, необхідно встановити межі ефективності розв'язання задачі. Тобто, яка теоретично найбільша швидкість розв'язання розглядуваної задачі (класу задач) заданого розміру може бути досягнута? Для відповіді на ці запитання скористаємося наступним підходом.

Встановлення межі ефективності розв'язання задачі.

Існує система оцінок складності алгоритмів розв'язування задач розмірності N, запропонована Кнутом [1]. Згідно системи Кнута ефективність задачі можна визначити за допомогою верхніх нижніх та оптимальних оцінок складності:

верхні оцінки:

$O(f(n)) = \{ g(n) | \exists C>0 \text{ і } N_0>0: g(n) \le Cf(n), \forall n \ge N_0\}$ (по суті **sup{** g(n) **}**

нижні оцінки:

$\Omega(f(n)) = \{ g(n) | \exists C>0 \text{ і } N_0>0: g(n) \ge Cf(n), \forall n \ge N_0\}$ (по суті **inf{** g(n) **}**

ефективні оцінки:

$$\theta(f(n)) = \{ g(n) | \exists C_1, C_2, N_0 > 0: C_1 f(n) \le g(n) \le C_2 f(n), \forall n \ge N_0 \}.$$

Для встановлення теоретичної межі ефективності задачі необхідно перш за усе встановити нижню оцінку складності так, як вона вказує найменшу можливу швидкість розв'язання задачі. Знаючи нижню оцінку складності, ми намагатимемось шукати (розробляти) стратегію, яка дозволяє побудувати алгоритм, оцінка складності (верхня чи оптимальна) якого буде близька до цієї оцінки складності у найгіршому випадку. У багатьох випадках встановити нижню оцінку складності не просто, проте існує метод звідності задач [2-8], який дозволяє встановлювати нижні та верхні оцінки складності до деяких задач.

Метод перетворення(звідності) задач

Означення. *Задача **A** може бути перетворена в задачу **B** (задача **A** зводиться до задачі **B**), якщо:*

*1. Вхідні дані задачі **A** перетворюються у вхідні дані задачі **B** з часом $r_{in}(n)$;*

*2. Розв'язується задача **B**;*

*3. Результат розв'язку задачі **B** перетворюється у правильний розв'язок задачі **A** з часом $r_{out}(n)$.*

Якщо кроки 1 та 3 можна виконати за час $O(r(n)) = \max\{r_{in}(n), r_{out}(n)\}$, де n – розмір задачі **A**, то кажуть, що задача **A** є $r(n)$ звідною до **B** і позначають так:

$$\mathbf{A} \propto_{r(n)} \mathbf{B} . \qquad (1)$$

Звідність не є симетричним відношенням. Якщо задачі **A** та **B** взаємно перетворюємі, то вони називаються *еквівалентними*. Мають місце дві ключові теореми, які дозволяють побудувати схему встановлення оцінок складності задачі.

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

Наслідок 1.1. *Якщо відома нижня оцінка складності задачі **A** і задача $\mathbf{A} \propto_{r(n)} \mathbf{B}$ то нижня оцінка задачі **A** буде нижньою оцінкою задачі **B**.*

Припустимо нам необхідно встановити нижню оцінку складності для деякої задачі обчислювальної геометрії $Z(n)$. Використовуючи означення звідності, знаходимо на множині класів звідності задач обчислювальної геометрії KOG_Z деяку задачу A, яка зводиться до нашої задачі Z. А далі застосовуючи наслідок 1.1 досить легко встановлюємо нижню оцінку складності задачі Z.

Теорема 2. *(верхні оцінки методом перетворення). Якщо задачу **B** можна розв'язати за час $T(n)$ і задача $\mathbf{A} \propto_{r(n)}$ (**A** перетворюєма в **B**), то **A** можна розв'язати за час, який не перевищує $T(n)+O(r(n))$.*

Наслідок 2.1. *Якщо задачу **B** можна розв'язати за час T(n) і задача **A** $\propto_{r(n)}$ **B**, то усі алгоритми задачі **B** із часом T(n) можна застосувати до розв'язання задачі **A** із часом, який не перевищуватиме O(T(n).*

Використовуючи означення звідності знаходимо задачу B із множини класів звідності KOG_Z до якої зводиться наша задача Z. Тоді уся множина алгоритмів задачі B згідно теореми 2 та наслідку 2.1, є підмножиною множини алгоритмів задачі *Z*, серед яких є алгоритм бажаної ефективності.

Множина класів звідності задач обчислювальної геометрія достатньо велика і формально її можна позначити у вигляді:

$$KOG_Z = \{K^{OG}_{f_1(N)}, K^{OG}_{f_2(N)}, \ldots, K^{OG}_{f_s(N)}\} \qquad (2)$$

Наприклад наступний список задач входить до класів звідності порядку $(K^{OG}_{\Omega(N\log N)})$.

1. Перевірка унікальності елементів: чи є на заданій множині два рівних елементи.
2. Перевірка рівності включення множин.
3. Перевірка перетину множин.
4. Перевірка на значимість: задана множина з *S* точок на площині: чи усі точки належать опуклій оболонці множини точок.
5. Перевірка ε- близькості: Задана множина з *S*, чи є серед елементів цієї множини два елементи, які знаходяться на відстані ε.
6. Об'єднання інтервалів.
7. (Задача сортування). Задана множина з *S*, необхідно їх відсортувати.

Отже повертаючись до нашої задачі методом звідності можна встановити нижню оцінку складності задачі локалізації точки на площині.

Нижня оцінка складності задачі локалізації точки. *Задача бінарного пошуку розмірності N зводиться за час O(1) до задачі локалізації точки, а тому нижня оцінка складності задачі локалізації точки рівна* $\Omega(\log N)$.

Доведення цього факту наведено у монографії [5]. Таким чином нам необхідно розробити стратегію, яка б дозволяла розробляти алгоритми локалізації точки на планарному розбитті G(V,E) з оцінками складності $O(\log n)$ та $\Theta(\log n)$.

Визначення інструментів та стратегії розв'язання:

Отже метою стратегії є досягнення оцінки складності: $O(\log n)$ та $\Theta(\log n)$.

Ми маємо $O(n)$ многокутників та кількість сторін многокутника у найгіршому випадку. Тому повний перебір дає $O(n^2)$. Нам необхідно знайти рішення, яке дозволить будувати алгоритми локалізації за час $O(\log n)$. Тобто нам необхідно зменшити час перевірки належність многокутнику до $O(1)$. Це можна зробити, якщо замість *N*-кутників будемо мати *k*-кутники (підрозбиття на *k*-кутники (наприклад

трикутники)), при цьому $k << N$. Далі представити k-кутники у вигляді структур даних, наприклад дерева пошук. Таким чином рішення полягає у наступному:

1) підрозбити на області задане планарне розбиття G(V,E) так, щоб перевірка належності многокутника була $O(1)$. Варіанти можуть бути такими: розбиття на трикутники, смуги, чотирикутники та монотонні ланцюги, рис. 2 - 4.

2) Одержане підрозбиття представити у вигляді бінарного дерева пошуку.

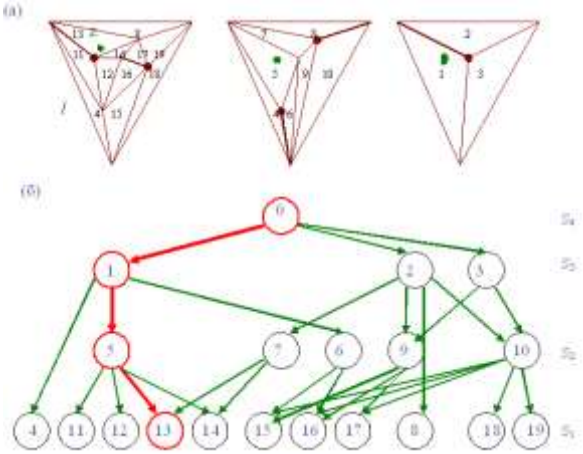


Рис.2. Розбиття на трикутники. Пошук $O(\log n)$.

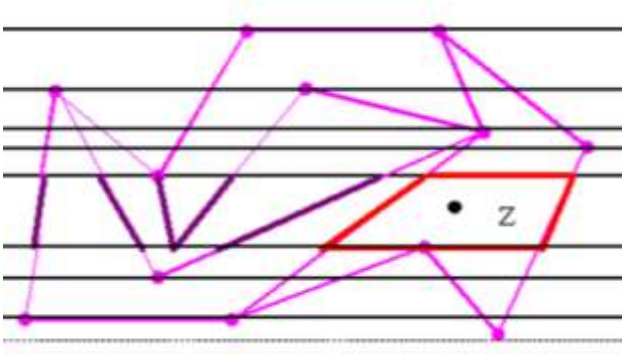


Рис.3. Розбиття на смуги. Пошук $O(\log n)$

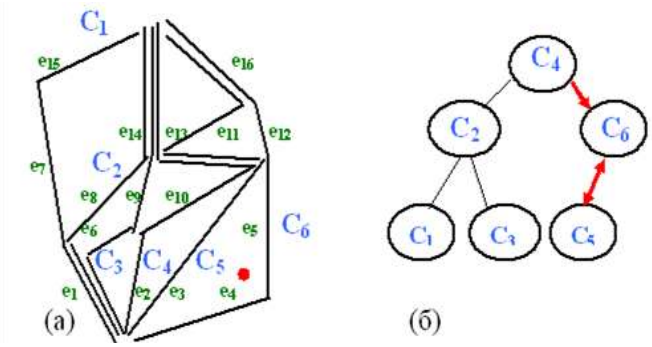


Рис.4. Розбиття на ланцюги. Пошук $O(\log^2 n)$

Створення та використання структур даних

Вхідні дані в обчислювальній геометрії мають свою специфіку і зазвичай являють собою множини геометричних об'єктів складної структури: множина точок, множина ребер, множина многокутників, множина многогранників, множина об'єктів обертання, планарне розбиття, тощо. Для організації ефективної роботи алгоритмів над такими множинами даних , останні треба представити у вигляді структур даних. Найбільш поширеними представленнями множин геометричних об'єктів в обчислювальній геометрії: списки (односторонні, одно та двозв`язні кільцеві списки, стеки, черги, РСПЗ, тощо), дерева (бінарні та k-арні дерева, дерево відрізків, k-d-дерево, АВЛ дерева, зчеплена черга, червоно-чорне дерево, тощо), хеш таблиці [9-12]. Коротко охарактеризуємо основні структури даних та ефективність їх застосування на деяких прикладах задач.

Означення. *Опис складних об'єктів засобами більш простих типів даних, які безпосередньо представляються у машині, називається* ***структурами даних.***

Склад структур даних: структура пам'яті для зберігання даних, способи її формування, модифікація та доступ до даних, набір операцій над елементами даних.

Типи структур даних [5, 9-12]

Означення.

1. Списком (довжини n) називається впорядкована послідовність елементів a_1, a_2, ..., a_n. Список розміру 0 називається **порожнім.**

2. Список реалізується за допомогою масиву або зв'язування його елементів вказівниками (**зв'язаний список**). У зв'язаному списку елементи лінійно впорядковані.

3. Односторонній зв'язаний список (однозв'язний) **–** це зв'язаний список який містить вказівник на наступний елемент. **Кільцевий односторонній зв'язаний список (кільцевий однозв'язний)** - це однозв'язний список із вказівником з останнього елемента на перший, рис. 5.

4. **Двосторонній зв'язаний список** (двозв'язний) містить три поля: ключ та два вказівники – наступний та попередній. Кільцевий **двосторонній зв'язаний список** (кільцевий двозв'язний) - це список із вказівником з останнього елемента на перший і з першого на останній, рис. 5.

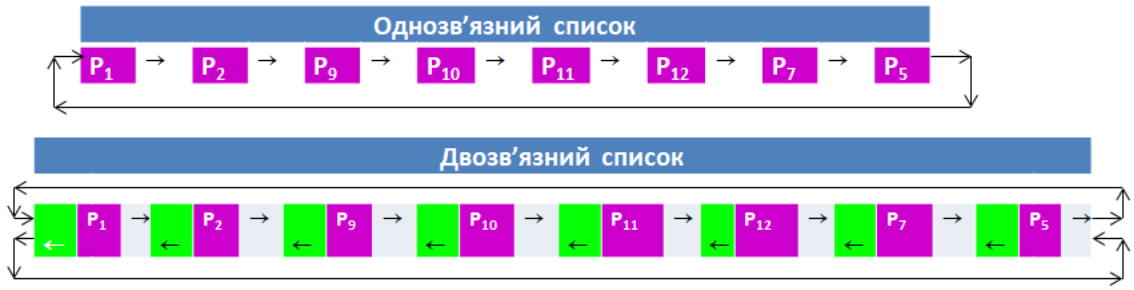


Рис.5.

Прикладом застосування таких списків є представлення многокутників (полігонів), рис.6.

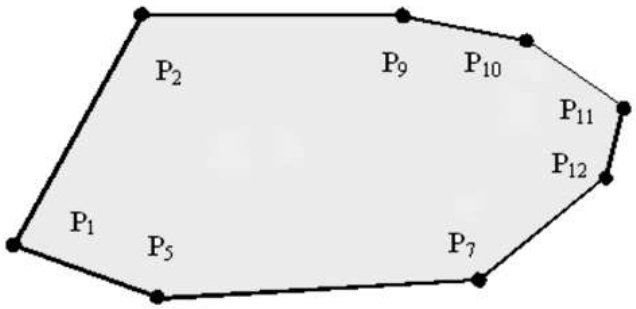


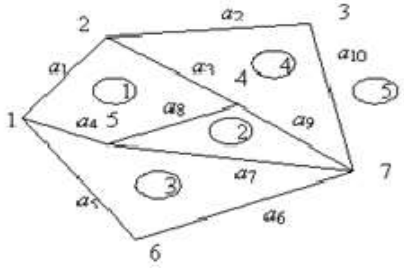
Рис. 6. Полігон

5. **Стеки** та **черги** – це динамічні множини (або спеціальні типи списків), в яких елемент що додається, визначається структурою множини. Стек - це лінійний список, де операції додавання і вилучення елемента та доступу до елемента виконуються тільки в його кінці. Черга -це лінійний список, в якому елементи вилучаються з початку списку, а додаються в його кінці.

6. **Реберний список з подвійними зв'язками (РСПЗ) [5].**

Досить часто в задачах обчислювальної геометрії використовується у тій чи іншій формі спеціальний список для представлення планарного розбиття – реберний список з подвійними зв'язками (РСПЗ). Він представляється у вигляді шестивимірного масиву таким чином:

Нехай V = {v_1, v_2, …, v_N }, а E = {e_1, e_2,…,e_M }. Головна компонента РСПЗ для планарного графа (V, E) це **реберний вузол** , який містить чотири інформаційні поля (V_1, V_2, F_1, F_2) і два поля вказівників (P_1 і P_2). Поле V_1 - містить початок ребра, поле V_2- містить його кінець; так ребро отримує умовну орієнтацію. Поля F_1 і F_2 містять імена граней, які лежать відповідно праворуч і ліворуч від ребра, орієнтованого від V_1 до V_2. Вказівник P_1 (відповідно P_2) задає реберний вузол, який містить перше ребро , яке зустрічається слідом за ребром (V_1, V_2), при повороті від нього проти часової стрілки навколо V_1 (відповідно V_2) . Імена граней і вершин можуть бути задані цілими числами. На рис. 7 подано приклад представлення планарного розбиття за допомогою РСПЗ.



	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

Рис. 7

7. Бінарні дерева [5, 9-12].

Дерево відрізків. Використовується в задачах регіонального пошуку, як самостійна структура або входить як складова дерева регіонів. Дерево відрізків визначено на цілочисельному інтервалі [L, N] і будується рекурсивно як двійкове дерево, рис. 8. На дереві визначено операції вставки та вилучення інтервалу.

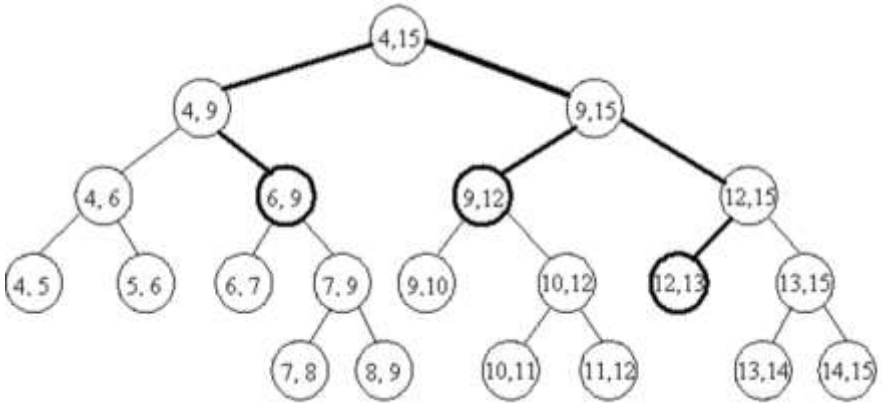


Рис. 8. Дерево відрізків для інтервалу [4,15]; Вставка інтервалу [6, 13] в T(4,15). [6, 13]= [6, 9]Ú [9, 12]Ú [12, 13]. Вузли віднесення: [6, 9], [9, 12], [12, 13].

Розглянемо застосування цієї структури на прикладі задачі регіонального пошуку.

Задача 2(задача регіонального пошуку). *На заданій множині S із n точок та запитного регіону R , знайти підмножину точок множини S (або їх кількість), які містяться в регіоні R, рис.9 а.*

Нижня оцінка складності. Задача бінарного пошуку розмірності *N* зводиться за час *O*(1) до задачі регіонального пошуку, а тому нижня оцінка складності задачі регіонального пошуку $\Omega(\log n)$.

До розв'язання цієї задачі застосовується метод дерева регіонів, основою якого є структура даних – дерево регіонів, рис.9 б.

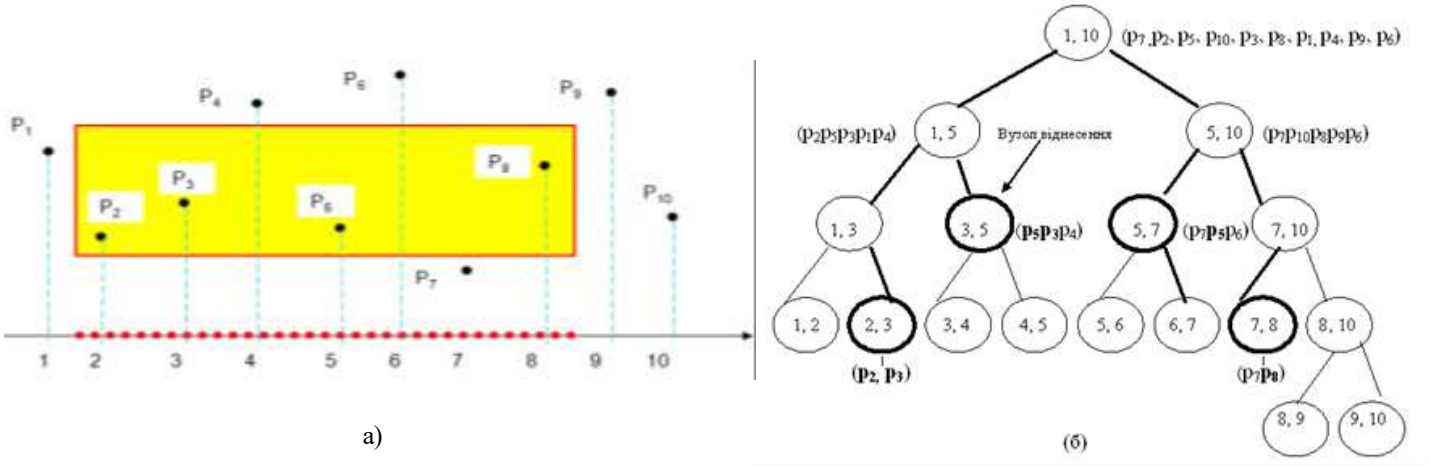


Рис. 9.

Ідея методу. Будується дворівнева структура даних (дерево регіонів), перший рівень якої – дерево відрізків по одній із координат (наприклад *x),* а другий – вузли дерева відрізків прошиті упорядкованими списками по іншій координаті (наприклад *y).*На побудованому дереві регіонів виконується операції вставки інтервалу (проекція запитного регіону на вісь *OX),* що дозволяє визначити вузли віднесення. Далі у вузлах віднесення застосовується дихотомія пошуку по іншій координаті (*y*).

Багатовимірне двійкове дерево (*k-d*-дерево) [5]

Простота та гнучкість реалізації *k-d* дерева пошуку дозволяє використовувати його у комп'ютерних іграх, моделюванні, комп'ютерній графіці, в задачах комп'ютерного зору, базах даних, а також інших алгоритмічних задачах. На рис.10 показано приклад такого дерева для множини *S* із *n* точок на площині. Для його побудови застосовується рекурсивна процедура, яка на кожному кроці рекурсії почергово (по координаті x та y) визначає медіану впорядкованих списків $U_x = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11}\}$, $U_y = \{P_5, P_7, P_2, P_8, P_{10}, P_3, P_4, P_1, P_{11}, P_9, P_6\}$ і розбиває по ній вертикаллю чи горизонталлю площину, що містить залану множину точок, рис.10 а.

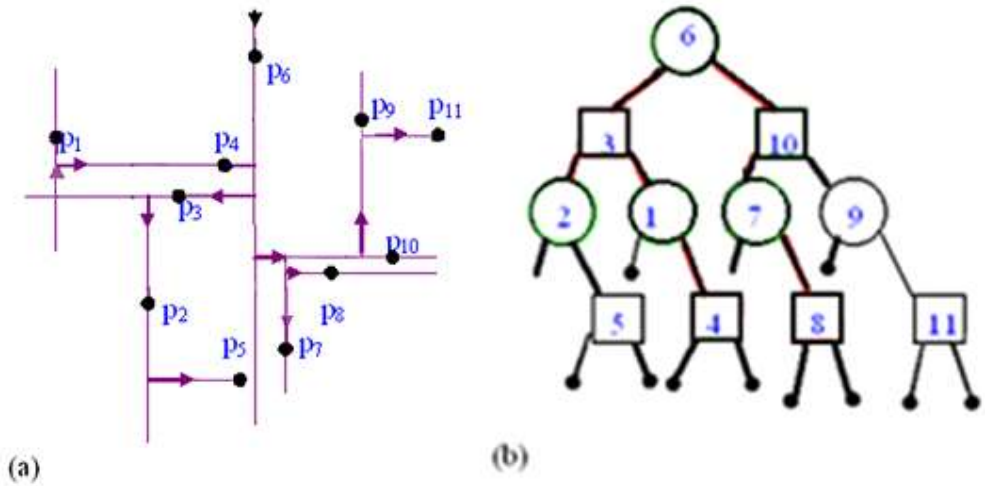


Рис.10.

Паралельно розбиттю формується бінарне дерево пошуку рис.10 б, яке називають багатовимірним деревом пошуку (*k-d* дерева пошуку). Зокрема ця структура даних ефективно використовується для розв'язання задачі 2, для запитного регіону прямокутної форми, сторони якого паралельні осям координат, рис. 11.

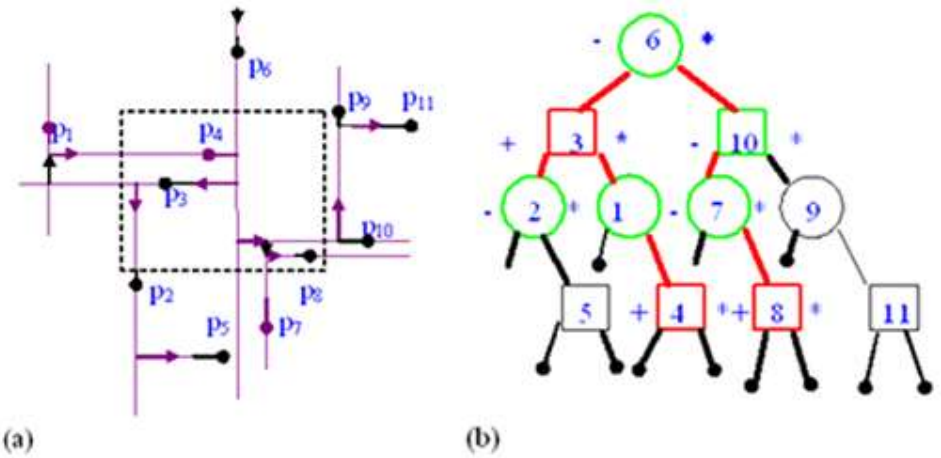


Рис.11.

Дерево інтервалів [9,10]

Ця структура даних використовується при розв'язанні одного із класів задач регіонального поуку типу задачі 2.

Задача 3. Задано *N* ортогональних відрізків на площині і запитний регіон *R* (прямокутник). Необхідно знайти множину відрізків, яка має не порожній перетин із запитним регіоном *R*, рис.12.

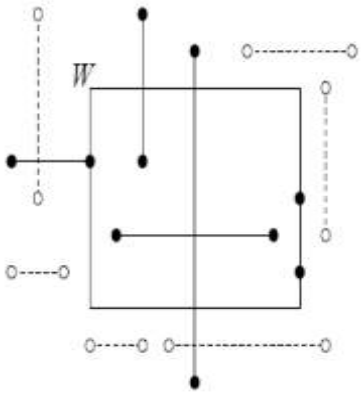


Рис.12. Перетин ортогональних відрізків із запитним регіоном

Розв'язок. Структура даних **дерево інтервалів** (у поєднанні із деревом регіонів) час $O(\log^2 n)$. Тобто, для розв'язання задачі по обом координатам окремо застосовується дерево інтервалів. На рис. 13 подано приклад дерева інтервалів для відрізків паралельних осі OX.

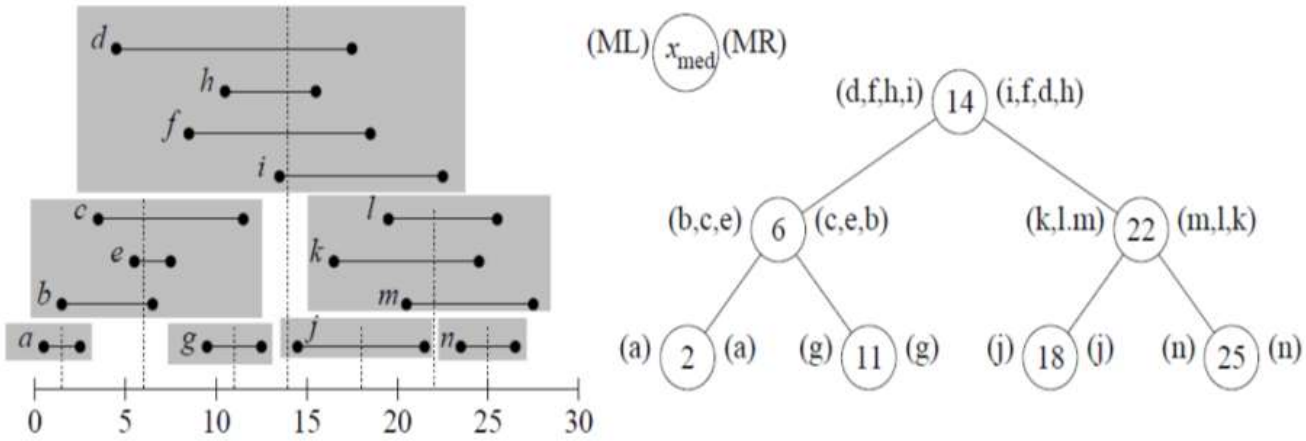


Рис. 13.

Зчеплена черга[5]

Зчеплена черга - це бінарне дерево з вказівниками, вузлами якого є точки заданої множини. Опуклу оболонку можна представити у вигляді лінійного списку CH(S) ={P_5, P_1, P_2, P_9, P_{10}, P_{11}, P_{12}, P_7} і у той же час у вигляді зчепленої черги, рис.14.

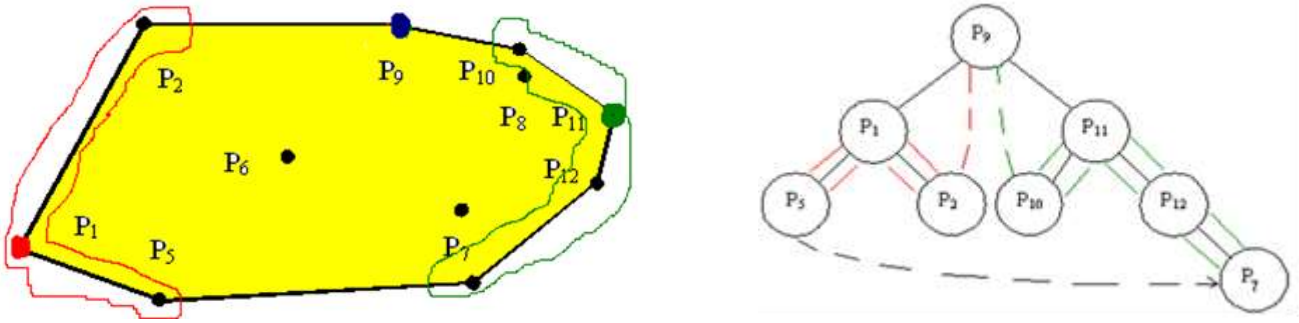


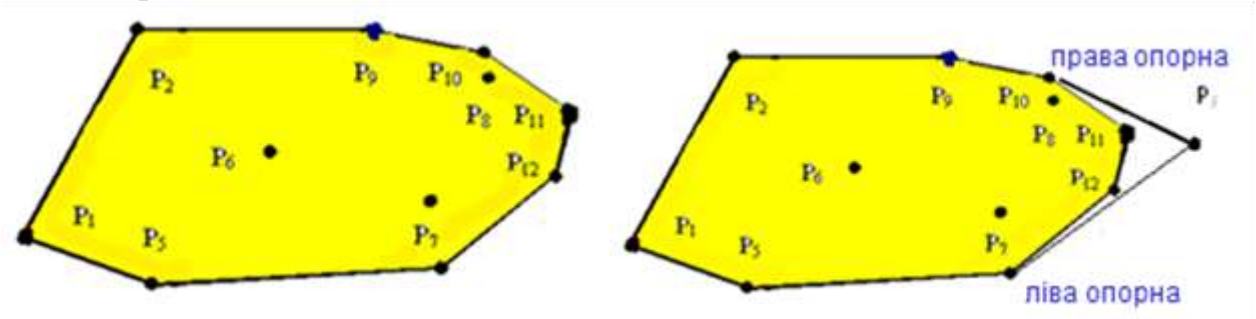
Рис.14

Щоб краще зрозуміти суть цієї структури даних, розглянемо задачі у яких вона активно використовується і дозволяє побудувати надшвидкі алгоритми для розв‘язання задач пошуку опуклої оболонки, і зокрема динамічних задач. Розглянемо динамічну задачу пошуку опуклої оболонки на множині *S* із *n* точок на площині.

Задача 3. Для множини точок *S*, яка формується доданням точок з постійною затримкою надходження, побудувати структуру даних, яка підтримує поточну опуклу оболонку і дозволяє корегувати її при надходженні нової точки.

Нижня оцінка корекції відома і рівна $\Omega(\log n)$.

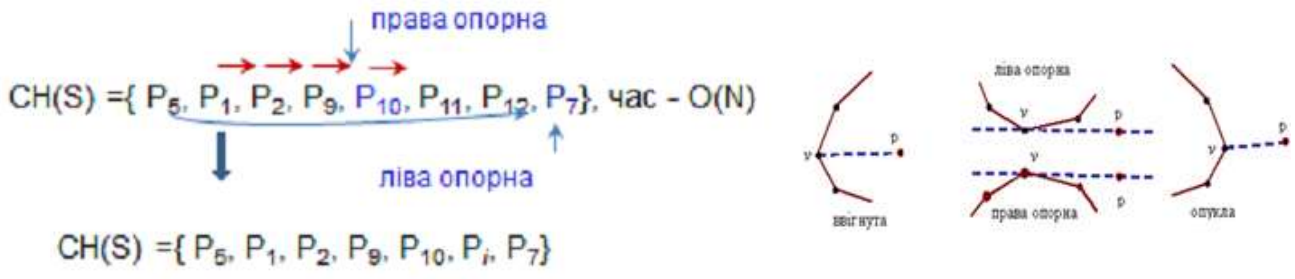
Нехай маємо поточну опуклу оболонку CH(*S*), яка підтримується у вигляді лінійного списку CH(*S*) ={ P_5, P_1, P_2, P_9, P_{10}, P_{11}, P_{12}, P_7}. Припустимо надійшла нова точка P, рис. 15.



а) б)

Рис.15.

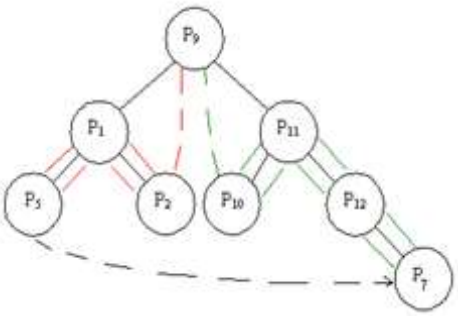
Для пошуку лівої і правої опорних точок для вставки нової точки ми рухаємось по лінійному списку CH(*S*) (використовуючи відповідну класифікацію точки(рис.16 б)) за час O(*n*) одержуємо результат у найгіршому випадку, рис. 16 а.



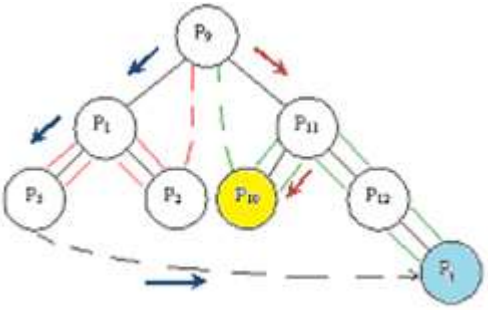
а) б)

Рис.16.

Якщо ж опуклу оболонку представити у вигляді зчепленої черги, рис. 17.а, то пошук опорних точок і вставку нової точки ми виконаємо за час $\theta(\log n)$ у найгіршому випадку, рис. 17 б.



а)



б)

Рис.17.

ЗАДАЧІ НА ОХОПЛЕННЯ

1. Побудова найменшого охоплюючого кола навколо множини точок (ОМК)

Задача є класичною і їй присвячений великий набір наукової літератури [13- 18]. Пошук ефективного алгоритму розв'язання цієї задачі почався ще у 1860 році Сильвестром [13]. Теоретична база була підведена у праці Радамахера та Тепліця у 1957 році [58] : найменше охоплюючи коло – єдине, і більше того воно або має за діаметр відрізок, що сполучає деякі дві точки множини, або є описаним колом для деякої трійки точок з вихідної множини. Таким чином, з вище сказаного, можна запропонувати простий перебірний алгоритм зі складністю $O(n^4)$. У 1972 році Елзінг та Хірн [17] покращили цей алгоритм до O (n^2).

В дослідженні операцій ця задача відома як „мінімаксна задача про розміщення центру масового обслуговування". Необхідно знайти точку $p_0 =(x_0 ,y_0)$ (центр кола), для якого найбільша з відстаней до точок заданого множини була б мінімальною, тобто задовольняла критерій:

$$P_0 = \min\max \{ (x_0 - x_i)^2+(y_0 - y_i)^2 \} \qquad (3)$$

Такий мінімаксний критерій, наприклад, використовується для визначення місцезнаходження пунктів екстреної допомоги , таких як поліцейські дільниці та станції швидкої допомоги з метою мінімізувати час надання допомоги в найгіршому випадку. Він також використовується для оптимального розміщення радіопередавача, що обслуговує N приймаючих приладів, щоб мінімізувати необхідну потужність радіопередавача (Тореґас у 1971 році [59]). Деякі автори, використовували цей критерій як задачу неперервної квадратичної оптимізації [17]. Це призвело до появи цілої низки ітеративних алгоритмів (на кшталт градієнтних методів) розв'язку цієї задачі, яка безумовно є дискретною.

У теорії розташування об`єктів ця задача відома як задача про 1-центр. Існує ряд відомих методів її розв`язання [14]. Наприклад, з застосуванням діаграми Вороного це можна зробити за час $O(n*\log(n))$ [5]. Використовуючи метод пошуку з відрізанням, Меджиддо [16] одержав алгоритм знаходження найменшої гіперсфери в m-вимірному евклідовому просторі, що оточує задану скінченну сукупність точок, який має складність $O(n)$.

Розглядувана задача має велике практичне застосування: наприклад використовуючи її можна знайти мінімальну допустиму потужність радіопередавача і його місце розташування для забезпечення зв'язком певного населеного пункту чи військової частини. З розвитком в Україні мобільних телекомунікацій актуальність цієї задачі ще підвищилась, оскільки оптимальне розташування передаючих антен значно знижує енергетичні витрати передавача і покращує якість зв'язку. Саме тому її програмна реалізація, а точніше реалізація ефективного алгоритму її розв'язання є гідним товаром на ринку IT – індустрії.

Постановка Задачі. *На площині задана множина S із n точок. Знайти найменше коло (тобто коло мінімального радіуса або таке, що обмежує круг найменшої площі), яке оточує усю множину S.*

Методи розв'язання

1.1. Алгоритм з використанням Діаграми Вороного.

Для того щоб отримати результат розв'язання задачі ОМК кращий ніж $O(n^2)$ пропонується підхід, в основі якого лежить побудова Діаграми Вороного, пошук діаметра множини точок та діаметра опуклої оболонки [5].

Ідея підходу:

1. Будуємо опуклу оболонку (границя CH(S)) ($O(n\log n)$).
2. Знаходимо найвіддаленішу пару точок {p, q}на границі опуклої оболонки CH(S). Вона утворить діаметр D шуканого кола (O(n)).
3. На D будуємо коло радіусом D\2 ($O(1)$).
4. Якщо залишились точки за межами кола, шукаємо серед них найвіддаленішу від діаметра D точку.

Детально алгоритм побудови діаметра множини точок викладено у роботах [5,18, 19], який формалізується двома теоремами.

Теорема 1.1 [18]. *Діаметр множини дорівнює діаметру її опуклої оболонки.*

Ця теорема дозволяє уникнути обчислення відстані для кожної пари точок.

Теорема 1. 2 [19]. *Діаметр опуклої фігури дорівнює найбільшій з відстаней між двома паралельними опорними прямими до цієї фігури.*

Ця теорема дозволяє зменшити зайві перевірки відстаней між парами точок: розглядати лише протилежні пари точок. Очевидно , що паралельні прямі можна провести не через *кожну* пару точок, рис.1.2. Це означає, що не обов'язково перевіряти всі відрізки в опуклій оболонці на властивість «бути діаметром». Пара точок, через яку можна провести паралельні опорні прямі, що дотикаються до опуклої оболонки, називається *протилежною парою*. Задача полягає в тому, щоб визначити усі протилежні пари, не перебираючи усі можливі варіанти точок.

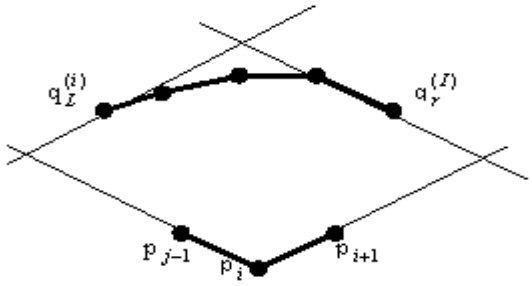


Рис. 1.1

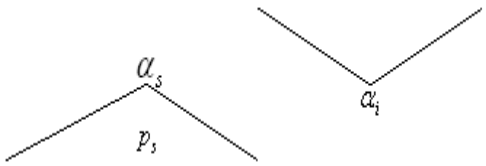


Рис.1.2

Розглянемо вершину p_i опуклого многокутника P, вершини якого пронумеровані у порядку проти годинникової стрілки, Рис.1.1. Будемо рухатись по границі многокутника P проти годинникової стрілки. Почавши рухатись в вершині p_i, будемо рухатись до тих пір, поки не досягнемо вершини $q_R^{(i)}$, що є максимально

віддаленою від $p_{i-1}p_i$. Аналогічним чином визначається вершина $q_L^{(i)}$, що максимально віддалена від p_ip_{i+1} при обході за годинниковою стрілкою, починаючи з вершини p_i. Ланцюг вершин між $q_R^{(i)}$ та $q_L^{(i)}$, включаючи ці вершини, дає множину $C(p_i)$, кожна вершина якої утворює з p_i протилежну пару. Оскільки $q_L^{(i)}$ є одночасно і $q_R^{(i+1)}$, то тривіальним стає алгоритм породження протилежних пар. Відносну

$$S = \begin{vmatrix} x & y & 1 \\ x' & y' & 1 \\ x'' & y'' & 1 \end{vmatrix} / 2 \qquad (4)$$

відстань від відрізка до точки можна обчислювати, як площу трикутника зі знаком (1), $p_i = (x, y)$, $p_{i+1} = (x', y')$, $p = (x'', y'')$. Тобто, Основна операція, що необхідна для цього - перевірка умови максимальної віддаленості точки від прямої, що обумовлюється відрізком p_ip_{i+1}. Це просто зробити , знайшовши площу трикутника (p_i, p_{i+1}, p). У роботі [5] представлено алгоритм породження протилежних пар.

Ми визначаємо всі протилежні пари за один прохід границі опуклої оболонки CH(*S*). Кількість протилежних пар, які породжені алгоритмом, не перевищує 3N/2 – лінійна відносно |CH(*S*)|. Тоді визначення діаметру CH(*S*) можна виконати за лінійний час, оскільки максимум множини визначається за час, пропорційний її потужності. Тобто згідно відомої теореми (теорема 4.19 у [5]) випливає наступне:

Теорема 1.3. [5]**:** *Діаметр опуклого n - вершинного многокутника може бути знайдений за O(n) - лінійний час.*

Під час роботи алгоритму може бути таке розташування вхідних даних (точок), що коло, яке будується на знайденому діаметрі може містити не всі точки заданої множини. У цьому випадку проблему може вирішити наступний крок алгоритму, який використовує ДВ для віддаленої точки.

Ми знаємо, що мінімальне коло, що охоплює множину , визначається або діаметром заданої множини , або трьома її точками. Саме тому, якщо коло не охоплює всю множину, тоді необхідно знайти новий центр кола найменшого радіусу, яке б охоплювало усю множину точок. Для цього варто скористатись Діаграмою Вороного, одна із властивостей якої полягає у тому, що вершини Діаграми Вороного є центрами кіл, які описуються навколо сусідніх точок, а значить центр шуканого кола співпадатиме з однією із вершин діаграми Вороного. Оскільки ДВ містить лише $O(n)$ точок , а радіуси кіл, що пов'язані з кожною з вершин діаграми, дорівнюють відстані від цієї точки до будь-якої із сусідніх відповідних їй точок, мінімальна по всім вершинам відстань дає радіус нашого кола. Отже остаточний алгоритм буде таким.

Алгоритм

1.Побудова опуклої оболонки, використовуючи ДВ $(O(n))$.
2. Знаходження протилежних пар опуклої оболонки. $(O(n))$
3. Знаходження діаметру множини *S*. $(O(n))$
4. Перевірка на охоплення діаметром D усієї множини S. $(O(n))$

5. Якщо після перевірки залишились точки за межами кола діаметром D будуємо діаграму Вороного множини точок ($O(n\log n)$).
6. Пошук вершини діаграми Вороного, яка є центром кола яке охоплює S. ($O(n)$).

Оцінка складності алгоритму

Теорема 1. 4. *Для заданої множини S із n точок на площині задачу про найменше охоплююче коло можна розв'язати методом на основі діаграми вороного за час O(nlogn) з використанням O(n) пам'яті.*
Доведенням теореми є сам алгоритм описаний покроково вище.

1.2 Алгоритм побудови базового трикутника.

Розглянемо ідею алгоритму схожого на попередній із тією лише різницею, що замість Діаграми Вороного, використовується базовий трикутник (трикутник максимальної площі, вписаний в опуклу оболонки множини S). Для цього скористаємось наступними теоремами.
Теорема 1. 5. *Коло,описане навколо опуклої оболонки проходить хоча б через одну точку , яка належить кінцю відрізка ,який є діаметром цієї опуклої оболонки.*
Теорема 1. 6. *Трикутник, який є базою для побудови кола навколо опуклої оболонки має хоча б одну вершину, що належить діаметру.*
Враховуючи ці теореми можна запропонувати наступний алгоритм пошуку трикутника:

Алгоритм пошуку трикутника

1. Якщо в опуклій оболонці більше одного діаметру - вибираємо перший з них.
2. Припускаємо спочатку, що трикутнику належать обидва кінці діаметру V_1 та V_2. Вибираємо кінці діаметра – згідно теореми вони можуть бути вершинами шуканого трикутника.
3. Визначаємо точку V_3 опуклої оболонки , яка утворює з даними двома вершинами трикутник найбільшої площі ($O(\log N)$).
4. Якщо коло описане навколо цього трикутника містить всю множину точок , то кінець алгоритму;
інакше
5. Припускаємо що лише одна точка(кінець діаметру) V_1 належить шуканому трикутнику.
6. Зафіксувавши V_1 та V_3 шукаємо нове положення точки V_2 так, щоб ці точки утворювали трикутник найбільшої площі ($O(\log N)$.
7. Якщо коло описане навколо цього трикутника містить всю множину точок , то кінець алгоритму;
інакше
8. Припускаємо що лише друга точка (кінець діаметру) V_2 належить шуканому трикутнику, повторивши попередні кроки знаходимо центр кола, описаного навколо опуклої оболонки.

9. Аналогічні міркування проводимо й для інших діаметрів, якщо для першого наші припущення не виконуються. Оскільки кількість діаметрів в найгіршому випадку може бути $O(n)$, то час на пошук трикутника $O(n\log n)$.
Отже загальний алгоритм розв'язання задачі ОМК буде таким:

Алгоритм

1. Побудова опуклої оболонки (наприклад, методом "розділяй та володарюй") ($O(n\log n)$).
2. Знаходження протилежних пар опуклої оболонки. ($O(n)$)
3. Знаходження діаметру множини S. ($O(n)$).
4. Перевірка на охоплення діаметром усієї множини S. $O(n)$
5. Пошук базового трикутника. ($O(n\log n)$)

Оцінка складності алгоритму

Теорема 1. 7. *Для заданої множини S із N точок на площині задачу про найменше охоплююче коло можна розв'язати методом базового трикутника за час O(nlogn) з використанням O(n) пам'яті.*
Доведенням теореми є сам алгоритм описаний покроково вище.

1.1.3 Алгоритм на основі пошуку трьох базових точок .

Оскільки коло можна визначити трьома точками, то можна запропонувати алгоритм схожий на попередній. На відміну від базового трикутника можна шукати вершини такого трикутника. Таким чином наступний алгоритм може бути охарактеризований наступною властивістю.
Теорема 1. 8. *Мінімальне охоплююче коло для заданих n точок на площині є описаним для деякого гострокутного трикутника з вершинами серед цих точок, або ж деяка пара точок є діаметром цього кола.*

З теореми можна зробити висновок, що побудову мінімального охоплюючого кола можна здійснити через дві або три точки:
1. Для заданої множини точок побудувати опуклу оболонку за методом Грехема.
2. Серед точок, які є вершинами побудованої в п.1 опуклої оболонки знаходиться найвіддаленіша пара. Назвемо ці 2 точки базовими.
3. Для кожної з n-2 точок (2 виключені точки – базові) виконується пошук кута, що утворений базовими точками та поточною (однією з n-2). Звісно, мова йде про кут при поточній вершині.
4. Серед знайдених в п.3 кутів потрібно вибрати три найгостріші кути і відповідні цим кутам точки. Три, оскільки, як вже зазначалося вище, для побудови мінімального кола достатньо саме трьох точок і у випадку, якщо це коло не буде проходити через одну (чи дві) з базових точок, для його побудови буде потрібно саме 3 точки, які і шукаються у цьому пункті.
5. Перебрати всі можливі комбінації базових точок та точок, знайдених у п..4. Потужність множини точок для перебору не буде залежати від N. До того ж,

можливих варіантів завжди буде 10. Для кожного варіанту набору точок будуємо коло.
6. Для побудованих у п. 4 кіл виконуємо перевірку на приналежність їм усіх n вхідних точок.
7. Серед кіл, що успішно пройшли перевірку з п.6 вибирається коло найменшої площі.

Оцінка складності алгоритму

Теорема 1. 9. *Мінімальне охоплююче коло для заданих N точок на площині можна визначити за допомогою трьох вершин, які описують коло , за час O(nlogn).*

Доведення. На етапі попередньої обробки виконується побудова опуклої оболонки вхідної множини точок. Складність її буде дорівнювати $O(n*\log(n))$. Пошук найвіддаленішої пари вимагає складності $O(n*\log(n))$. Кути з п.3 алгоритму можна знайти за лінійний час, оскільки вимагається лише обрахування кута для кожної з n точок. Побудова кіл для кожного з 10 можливих варіантів, як і генерація цих варіантів не залежить від n. Для перевірки на приналежність точки колу потрібно $O(n)$ часу. Вибір кола мінімальної площі також не залежить від n. Таким чином, алгоритм має складність $O(n*\log(n))$, на попередню обробку відводиться $O(n*\log(n))$.

Висновки.

Запропоновані 3 алгоритми мають однакову оцінку складності $O(n\log n)$ і зводяться до пошуку діаметра через опуклу оболонку та третьої точки кола. Саме пошуком третьої точки кола і відрізняються , по суті, ці підходи. Для цього у першому використовується Діаграма Вороного, у другому - базовий трикутник і третьому звичайний перебірний пошук. Враховуючи зазначене можна узагальнити перший метод, побудувавши на етапі попередньої обробки Діаграму Вороного ($O(n\log n)$), то тоді усі інші процедури, включаючи побудову опуклої оболонки, пошуку діаметра та базового трикутника будуть виконуватись за лінійний час.
Меджиддо [16] у 1983 році поставив питання, а чи є необхідним мати повну інформацію про Діаграму Вороного дальньої точки для знаходження центру найменшого охоплюю чого кола, коли достатньо досліджувати лише ті вершини діаграми, які знаходяться у певному околі від шуканого центру. Створення ним методу побудови лише потрібної частини діаграми дозволило говорити про досягнення оптимальної лінійної складності алгоритму розв'язання.

2. Трикутник найменшої площі (ОМК)

Постановка задачі. *На заданій множині S із n точок на площині побудувати трикутник найменшої площі, який би охоплював S.*

3. Побудова прямокутника найменшої площі, що містить задану множину точок (ОМП)

Вступ

Задача пошуку прямокутника найменшої, що охоплює задану множину точок (ОМП) певною мірою схожа на дві попередні задачі, проте підходи до її розв'язання мають певні особливості. Перш ніж ми перейдемо до опису підходів розв'язання задачі слід сказати декілька слів про можливе застосування розв'язків.

Так, результати розв'язання задачі можуть бути застосовано до деяких випадків оптимізації задачі регіонального пошуку. Так, досить часто деяка множина точок розглядається як один об'єкт, та вважається, що усі точки множини належать запитному регіону, якщо хоча б одна точка йому належить. В таких випадках з метою оптимізації можна розглядати перетин прямокутника, що містить таку множину, та запитного регіону. Якщо такий прямокутник не порожній, то необхідно вважати, що точки належать запитному регіону. Оптимізація тут зрозуміла: якщо в нас є багато об'єктів, кожен з яких містить велику кількість точок, то заміна під час пошуку усіх точок об'єкту одним прямокутником дійсно зменшить час пошуку.

Також, різні картографічні системи містять багато многокутників складної форми. Вони задають будинки, райони, річки, дороги тощо. Кількість таких об'єктів може досягати десятків тисяч. Проте виведення всіх їх на екран ЕОМ може зайняти дуже велику кількість часу, оскільки є повільною операцією (окрім цього, під час виводу кожного об'єкту можуть декілька разів змінюватися стилі екрану—такі як колір, тип лінії—що також сильно уповільнює вивід). Отже, бажаним було б виводити тільки ті об'єкти, які будуть відображатися хоча б частково. Очевидною є задача регіонального пошуку, де запитний регіон–прямокутник на карті, що відповідає екрану ЕОМ. Якщо ж під час пошуку використовувати не сам об'єкт, а відповідний прямокутник, то отримаємо значну оптимізацію.

Також прямокутники, побудовані таким чином, можна використовувати в статистиці для зменшення кількості даних, що зберігаються. Так, часто множину точок допустимо замінити прямокутником та числом–кількістю точок, що входять до нього. В даному випадку важливою є навіть не економія пам'яті, а суттєво збільшена швидкість роботи з даними.

Постановка Задачі. *На площині задана множина S із n точок. Знайти прямокутник найменшої площі, який оточує усю множину S.*

Методи розв'язання

3.1. Алгоритм побудови з використанням опуклої оболонки.

Алгоритм (основна ідея)

Ідея розв'язання задачі про ОПМ в цьому підході полягає у побудові опуклої оболонки заданої множини точок. Тому, перша частина алгоритму–це побудова опуклої оболонки. Так, якщо усі точки мають належати шуканому прямокутнику, то

достатньо знайти прямокутник, якому належать усі точки опуклої оболонки. Усі інші точки знаходяться всередині оболонки, отже і всередині знайденого прямокутника. Таким чином, знаходження опуклої оболонки може зменшити кількість точок, що будуть оброблятися під час пошуку. Друга частина алгоритму полягає в знаходженні самого прямокутника. Тут використовується припущення, що хоча б одна із сторін прямокутника містить ребро опуклої оболонки. Це припущення легко обґрунтувати.

Припустимо від супротивного: на кожній із сторін прямокутника лежить лише одна точка. Розглянемо лише ці чотири точки (точки, якими опукла оболонка не спирається на многокутник, і які просто обмежують кут повороту "до упору" і більше ніякої ролі не відіграють)рис. 3.1. У цьому випадку (нижня точка "лівіша" за верхню, права "нижче" за ліву) наш прямокутник – не оптимальний варіант побудови прямокутника найменшої площі. Так як при повороті за годинниковою стрілкою ми досягнемо кращого результату, зменшивши і ширину, і висоту.

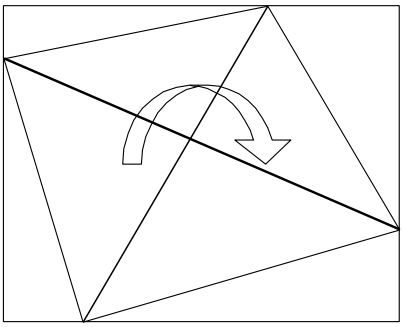


Рис. 3.1

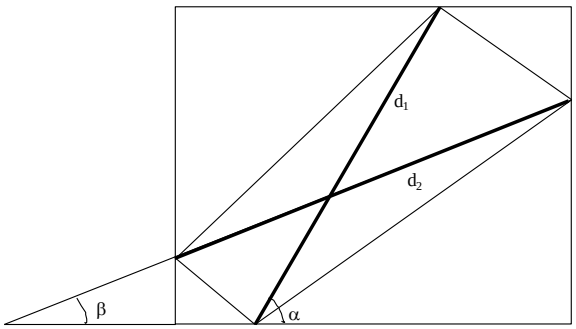


Рис. 3.2

Розглянемо випадок на рис. 3. 2. Чи можна бути впевненими, що роблячи поворот, так як ми це робили у першому випадку (рис. 3.1), результат буде кращим? І чому сторона опуклої оболонки співпадатиме зі стороною прямокутника?

Відповідаючи на ці запитання припустимо, що розміри початкового прямокутника $h = d_1 \cdot \sin\alpha$, $w = d_2 \cdot \cos\beta$. При повороті прямокутника на кут φ (за годинниковою стрілкою – кут від'ємний, проти – додатній) значення h і w змінять свої значення відповідно: $h^* = d_1 \cdot \sin(\alpha+\varphi)$, $w^* = d_2 \cdot \cos(\beta+\varphi)$. Тоді площа прямокутника матиме вигляд $S^* = h^* w^* = d_1 d_2 \sin(\alpha+\varphi) \cdot \cos(\beta+\varphi)$, звідси $dS^*/d\varphi = d_1 d_2[\cos(\alpha+\varphi)\cos(\beta+\varphi) - \sin(\alpha+\varphi)\sin(\beta+\varphi)] = d_1 d_2 \cos(\alpha+\varphi+\beta+\varphi)$. Таким чином:

1). При $\alpha+\beta<90°$ $S'>0 \Rightarrow \varphi\uparrow \Rightarrow$ можна обертати за годинниковою стрілкою. Тобто, при повороті прямокутника за годинниковою стрілкою площа прямокутника буде монотонно зменшуватись, а значить одержимо випадок на рис. 3.1, коли будуть монотонно зменшуватися ширина й висота.

2). При $\alpha+\beta>90°$ $S'<0 \Rightarrow \varphi\downarrow \Rightarrow$ можна обертати на кут $90°-\beta$ проти годинникової стрілки.

Отже, для кожного ребра оболонки шукаємо "найвищу", "найлівішу" та "найправішу" точку оболонки відносно даного ребра. Вони будуть задавати координати кутів прямокутника. Найменший серед таких прямокутників і буде шуканим. Таким чином, алгоритм складається з двох етапів:

1. Спочатку побудуємо опуклу оболонку одним із відомих методів (наприклад методом Грехема або методом «Загортання Подарунку»(gift wrapping)). Вона буде задана послідовністю точок, впорядкованою відносно найвищої точки за кутом проти годинникової стрілки. Верхня точка буде першою в списку вершин оболонки.
2. Пошук мінімального прямокутника.

Для кожного відрізку опуклої оболонки будуємо прямокутник, що містить саму верхню, саму ліву та саму праву точку відносно прямої, заданої цим відрізком; для кожного прямокутника знаходимо площу, та шукаємо найменшу площу. Більш детально:

2.1. Спочатку перейдемо в систему координат, що зв'язана з ребром. Нехай ребро задається точками з номерами *i-1, i*. Тоді початок координат буде в точці *i-1*, а ребро буде задавати напрямок осі *OX*. Вісь *OY* буде повернута на π/2 проти годинникової стрілки відносно *OX* (тобто система координат буде стандартною).

2.2. Тоді найвища точка буде мати найбільшу ординату в такій системі. Її ми шукаємо за допомогою двійкового пошуку, використовуючи впорядкованість точок оболонки.

2.3. Найправіша точка буде мати найбільшу абсцису. Її також шукаємо двійковим пошуком, проте логічно розглядати не всю оболонку, а ту її частину, що знаходиться між *i*-ою точкою та найвищою. Аналогічно шукається найлівіша точка (з найменшою абсцисою).

2.4 . Прямокутник шукається спочатку також в "реберній" системі координат. Ординати двох його нижніх кутів–*0*, а двох верхніх будуть дорівнювати ординаті знайденої найвищої точки. Абсциси лівих та правих кутів будуть рівними абсцисам, відповідно, найлівішої та найправішої точок оболонки.

Зрозуміло, що такому прямокутнику належать всі точки з множини. Площа його також легко шукається, оскільки сторони його паралельні осям "реберної" системи координат. Знайдена площа порівнюється з площею прямокутника, який вважався найменшим до розглядання даного ребра. Обирається кращій, та його координати перетворюються в екранні. Прямокутник, який буде вважатися найменшим після обробки всіх ребер, і буде мати найменшу можливу площу.

Опишемо тепер структури даних та сам алгоритм—більш детально.

Алгоритм (реалізація)

Спочатку розглянемо опис констант та змінних для зберігання точок:

```
const  MaxPointCount = 12000;
var
   PointCount: Integer;
                  PX,PY: array [0..MaxPointCount] of Real;
```

Константа `MaxPointCount` задає найбільшу можливу кількість точок, у змінній `PointCount` зберігається поточна кількість точок, які зберігаються у двох масивах:

PX,PY. Для того, щоб взнати координати *X, Y* *i*-ої змінної, потрібно взяти *i*-ий елемент відповідного масиву: PX[i], PY[i].

Під час пошуку опуклої оболонки відбувається сортування елементів цих масивів. Для обходу Грехема використовується стек, заданий масивом Stack номерів точок в масивах PX, PY та змінною Top–кількість елементів в стеку. Для подальшої роботи елементи, відповідні номерам зі стеку, копіюються у відповідні масиви OpX, OpY.

```
var
   Stack: array [1..MaxPointCount+1] of Integer;
   Top: Integer,
            OpX,OpY:array[0..MaxPointCount] of Real;
```

У змінній nh буде зберігатися кількість елементів опуклої оболонки. Варто звернути увагу на такий факт: перший (найвищий) елемент опуклої оболонки зустрічатиметься в масивах двічі OpX, OpY—як перший і останній. Таким чином опукла оболонка замикається, що спростить подальший бінарний пошук.

Опишемо тепер пошук прямокутника. Для зберігання прямокутників використовується такий тип:

```
type
   rec_t=record
            x1,y1,x2,y2,
            x3,y3,x4,y4,
   area:real;
end;
```

Перші вісім змінних зберігають координати відповідно лівого нижнього, лівого верхнього, правого верхнього та правого нижнього кутів (відносно ребра) в екранній системі координат, а area–площу прямокутника.

Як вже було сказано, пошук відбувається в циклі послідовно за ребрами опуклої оболонки. Проілюструємо порядок обходу ребер на малюнку(рис. 3.3).

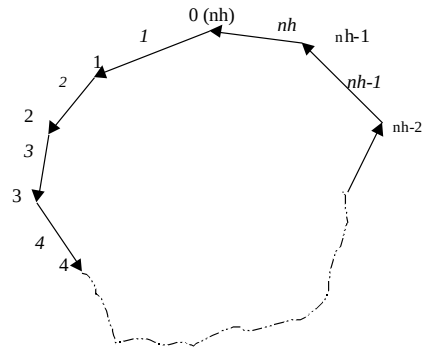


Рис.3.3.

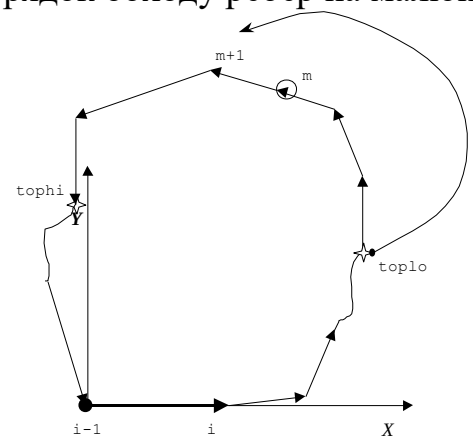


Рис.3.4.

Тут прямим шрифтом зображено номери точок (в масивах), курсивом–номери ребер (під час обходу). Змінна *i* є лічильником циклу. Перед циклом площа найменшого

прямокутника ініціалізується великим числом (10^{100}) . Під час перебору прямокутників воно зменшиться.

Розглянемо одну з ітерацій такого циклу. Спочатку знаходиться sin(⟨) та cos(⟨) кута між напрямленим ребром (віссю *OX* своєї системи координат) та віссю *OX* віконної системи координат. Якщо довжина ребра дуже маленька $(<10^{-7})$, то переходимо на наступну ітерацію. Потім шукаємо найвищу, найлівішу та найправішу вершини оболонки відносно даного ребра.

Розглянемо алгоритм пошуку на прикладі пошуку найвищої вершини. Спочатку зауважимо, що варто перепозначити вершини оболонки таким чином, щоб *i*–та вершина (кінець ребра) мала номер *0*, далі вершини нумерувалися проти годинникової стрілки та *i-1*–ша вершина (початок ребра) мала номер *nh-1*. Якщо цього не здійснити, то нумерація нашої оболонки відносно ребра була б розірваною, що ускладнить бінарний пошук. Наступна функція шукає координату *y* точки по її номеру в оболонці відносно кінця ребра.

```
function get_y(j:integer):real;
begin
   j:=(i+j) mod nh;
  get_y:=(OpX[j]-x1)*s-(OpY[j]-y1)*c;
end;
```

Перший рядок в тілі функції шукає номер точки в масиві оболонки (нагадую, що в нас *j*—номер відносно *i*–кінця поточного ребра). Друга—перетворює координату з екранної системи в реберну.

Тепер продемонструємо пошук найвіддаленішої по координаті *y* точки.

```
tophi:=nh-1;
toplo:=0;
while (toplo<tophi) do
  begin
    m:=(toplo+tophi) shr 1;
    y:=get_y(m);
    ys:=get_y(m+1);
    if (ys<y) then tophi:=m
              else toplo:=m+1;
  end;
```

Шукати точку будемо в інтервалі між точкою `toplo` (спочатку це є кінець ребра) та `tophi` (початок ребра). На кожній ітерації ми будемо знаходити точку, що лежить посередині такого інтервалу, наступну за нею точку та порівнювати їх ординати. Відповідно до результатів порівняння буде переміщено в середину початок або кінець відрізку. Зрозуміло, що не більше, ніж через $O(\log n)$ виконається умова виходу з циклу `while`–буде знайдено відповідну точку. На малюнку (рис.3.4) проілюстровано ітерацію такого циклу. З малюнку стає зрозумілим алгоритм пошуку. Після відшукання крайніх точок можна легко знайти площу прямокутника,

оскільки з координат крайніх точок в реберній системі координат легко шукається довжини його сторін. Залишилось лише перетворити координати прямокутника в екранні за допомогою наступної процедури:

```
procedure getinverse (sx,sy:real; var dx,dy:real);
begin
  dx:=(sx*c+sy*s)+x1;
  dy:=(sx*s-sy*c)+y1;
end;
```

Оцінка складності алгоритму

Теорема. 3.1. *Пошук прямокутника найменшої площі на множині S із N точок на площині можна виконати за час* $O(n\log n)$ *з використанням* $O(n)$ *пам'яті.*

Доведення. Перший етап алгоритму - побудова опуклої оболонки методом Грехема. Він має складність $O(n\log n)$. На другому етапі ми маємо цикл для усіх ребер опуклої оболонки, отже максимальна довжина циклу буде n. На кожній ітерації ми здійснюємо двійковий пошук у впорядкованій множині трьох точок. На це піде $O(\log n)$ часу. Отже, загальна складність пошуку прямокутника $O(n\log n)$. Таким чином, загальна складність всього алгоритму в цілому складатимете $O(n\log n)$.На рис. 3.5 показано приклад роботи алгоритму.

Висновки

З того, що було описано вище, можна зробити висновок, що дана програма дійсно шукає прямокутник найменшої площі, який містить задані точки, рис.3.5. Програма має зручний, інтуїтивно зрозумілий графічний інтерфейс, достатню швидкість роботи, можливість читання та запису даних у файл. Важливою перевагою є те, що обчислювальна частина програми знаходиться в окремому модулі, що дозволяє легко використати код алгоритму повторно, в інших програмах.

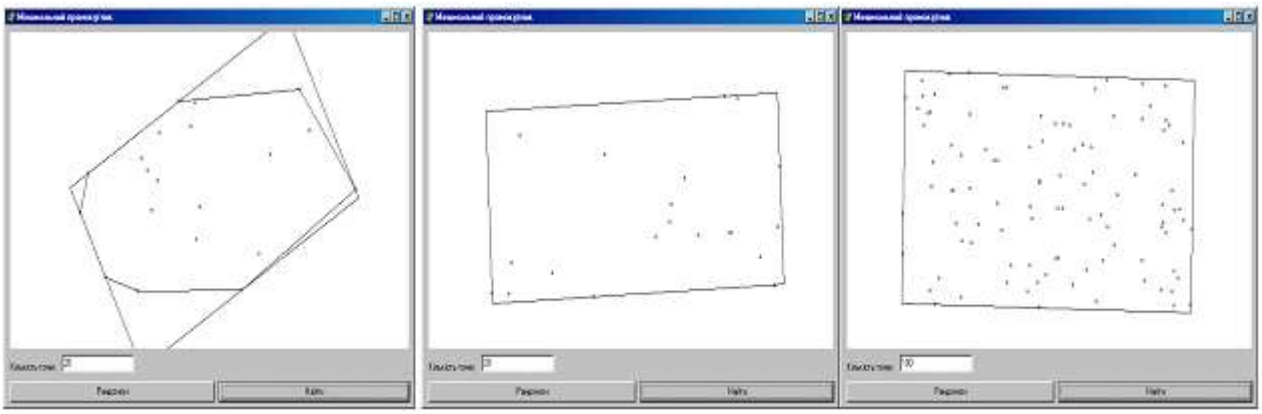


Рис. 3.5

4. Побудова найменшого охоплюючого еліпса(ОМЕ)

Застосування результатів розв'язання цієї задачі має місце в різних галузях, зокрема в медицині. Еліпс мінімальної площі використовують при пошуку області диска зорового нерва [20], або ж для виділення контурів розмитого зображення [21]. Також розв’язки задачі можна застосувати в сфері будівництва, при визначені місця під забудову будівельних об’єктів різної форми.

Щодо пошуку ефективних методів розв'язання задачі, то цій тематиці присвячено достатня кількість робіт [22, 23, 21, 24]. Так, у статі [22][24] розглядаються і досліджуються шість методів побудови еліпса (варіації з побудовою опуклої оболонки, пошуком трикутника найбільшої площі та інші). Нажаль дуже мала кількість матеріалів знаходиться у відкритому доступі. Крім того є вже і реалізовані алгоритми у вільно доступній бібліотеці CGAL [25].

Постановка задачі. *На заданій множині S із n точок на площині побудувати еліпс найменшої площі, який би охоплював S.*

Методи розв’язання

4.1. Алгоритм побудови з використанням опуклої оболонки(варіант1).

Алгоритм (основна ідея)

Перший варіант алгоритму був наступний. Серед заданих точок, знайти три такі, які утворюють трикутник максимальної площі. Через ці точки побудувати так званий еліпс Штейнера [26] (еліпс мінімальної площі, що проходить через три точки). Але в ході досліджень було встановлений контрприклад, для якого цей алгоритм не підходив. Якщо побудувати еліпс Штейнера через три точки з чотирьох, що задають прямокутник, то побудований еліпс може не захоплювати четверту точку. Таким чином було встановлено, що для побудови потрібного еліпса потрібно як мінімум чотири точки. А саме, необхідно знайти дві найвіддаленіші точки, а потім ще дві точки які утворюють трикутники найбільшої площі при умові, що вони знаходяться по різні сторони відрізка, що сполучає перші дві знайдені точки. Для реалізації цієї ідеї та пошуку цих точок потрібно побудувати опуклу оболонку до заданої множини точок і знайти шукані точки з поміж вершин опуклої оболонки. В даному випадку дві найвіддаленіші точки можна знайти за лінійний час, оскільки потрібно перевіряти лише протилежні пари точок, що знаходяться теж за лінійний час [27]. Щодо побудови еліпса, через задані точки використовується афінне перетворення [28]. Якщо еліпс має проходити через чотири точки – будується їх афінне перетворення в точки квадрата з координатами (1,1), (1,-1), (-1,-1), (-1,1). Тоді еліпс буде проекцією кола описаного навколо цього квадрата. У варіанті з трьома точками використовується правильний трикутник з координатами (1,1), $(-\sqrt{3}/2, -1/2)$,$(\sqrt{3}/2, -1/2)$ та центр (0,0). Проекція кола описаного навколо трикутника і буде шуканим еліпсом Штейнера [26].

Алгоритм (реалізація)

Для виконання алгоритму було реалізовано і використано наступні структури даних та функції, таб. 2:

TCoord = recordX, Y : Real;end;	Структура задає декартові координати точки
TPoints=^TLanka; TLanka=record Point: TCoord; First: boolean; Param: real; Index: integer; Next, Prev: TPoints; End;	Список точок з посиланням на наступні та попередні точки. First – ознака першої точки. Param – полярний кут. Index – ідентифікатор точки.
function getWidth(c1, c2: TCoord): Real;	Відстань між двома точками
function getPovorot(p1, p2, p3: TCoord): real;	Значення повороту
function getAngle(p1, p2, p3: TCoord):r eal;	Значення кута P1P2P3
function getInSidePoint(points: Tpoints): TCoord;	Пошук внутрішньої точки
function getTriangle(c1, c2, c3: TCoord): real;	Знаходження площі трикутника
function Grehem(var points: TPoints): boolean;	Обхід Грехема

Таблиця 2.

ПОБУДОВА ЕЛІПСА

Еліпс в координатах xy будується як проекція кола в координатах uv. Далі наведені математичні формули для перетворення:

$$\mathbf{x} = (x, y)^T \qquad \mathbf{u} = (u, v)^T$$

$$\mathbf{a} = (a_u, a_v)^T \qquad \mathbf{b} = (b_u, b_v)^T \qquad \mathbf{c} = (c_u, c_v)^T$$

$$x = \frac{a_0 + \mathbf{a}^T\mathbf{u}}{1 + \mathbf{c}^T\mathbf{u}} \qquad y = \frac{b_0 + \mathbf{b}^T\mathbf{u}}{1 + \mathbf{c}^T\mathbf{u}}$$

$$a_0 + \mathbf{a}^T\mathbf{u} + 0 + 0 + 0 - x\mathbf{c}^T\mathbf{u} = x$$

$$0 + 0 + 0 + b_0 + \mathbf{b}^T\mathbf{u} - y\mathbf{c}^t\mathbf{u} = y$$

$$\mathbf{p} = (a_0, a_u, a_v, b_0, b_u, b_v, c_u, c_v)^T$$

$$\mathbf{r} = (x_1, x_2, x_3, x_4, y_1, y_2, y_3, y_4)^T$$

$$\mathbf{Ap} = \mathbf{r}$$

$$\mathbf{A} = \begin{bmatrix} 1 & u_1 & v_1 & 0 & 0 & 0 & -x_1u_1 & -x_1v_1 \\ 1 & u_2 & v_2 & 0 & 0 & 0 & -x_2u_2 & -x_2v_2 \\ 1 & u_3 & v_3 & 0 & 0 & 0 & -x_3u_3 & -x_3v_3 \\ 1 & u_4 & v_4 & 0 & 0 & 0 & -x_4u_4 & -x_4v \\ 0 & 0 & 0 & 1 & u_1 & v_1 & -y_1u_1 & -y_1v_1 \\ 0 & 0 & 0 & 1 & u_2 & v_2 & -y_2u_2 & -y_2v_2 \\ 0 & 0 & 0 & 1 & u_3 & v_3 & -y_3u_3 & -y_3v_3 \\ 0 & 0 & 0 & 1 & u_4 & v_4 & -y_4u_4 & -y_4v_4 \end{bmatrix}$$

Основну частину вікна складає Дкартова система координат. Використовуючи формули перетворення [23] є можливість масштабувати та рухати видиму частину

системи паралельно осям координат. Є три способи завдання вхідної множини точок, навколо яких потрібно побудувати еліпс: ручне введення за допомогою «миші» (є можливість видалення точок) завантаження готових даних з файлу (є можливість збереження),генерація довільних точок (кількість вказується).

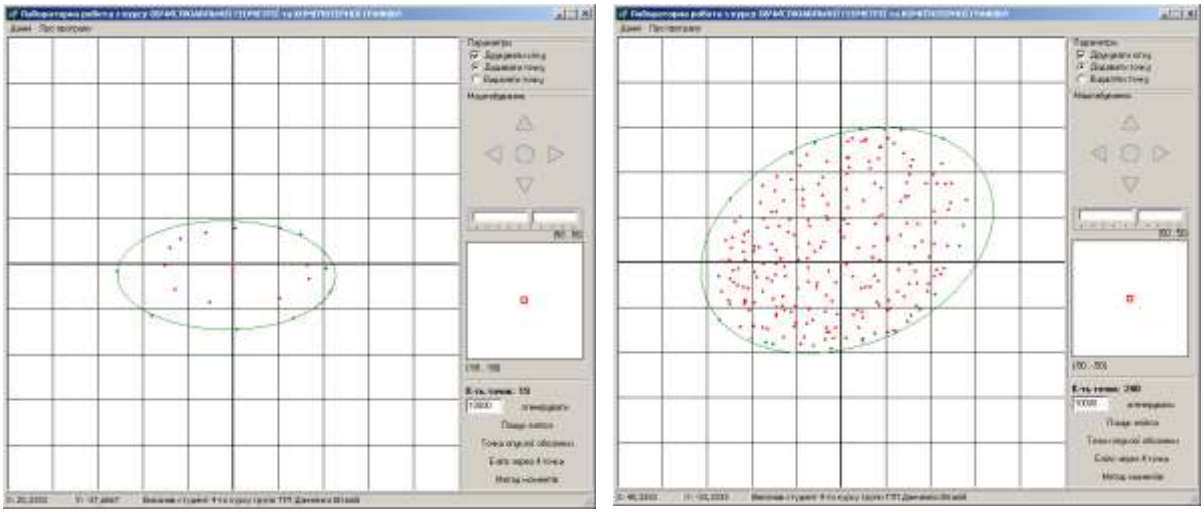


Рис. 4.1

Для отримання результату потрібно натиснуту на «Пошук еліпса». Також є окрема можливість побудови опуклої оболонки. Точки, що входять до оболонки відображаються іншим кольором. На екрані весь час відображається загальна кількість точок. При зміні розмірів форми, Декартова система координат автоматично масштабується для зручного відображення даних. Є можливість побудови еліпса методом моментів за лінійний час, але даний алгоритм не дає оптимального розв’язку.

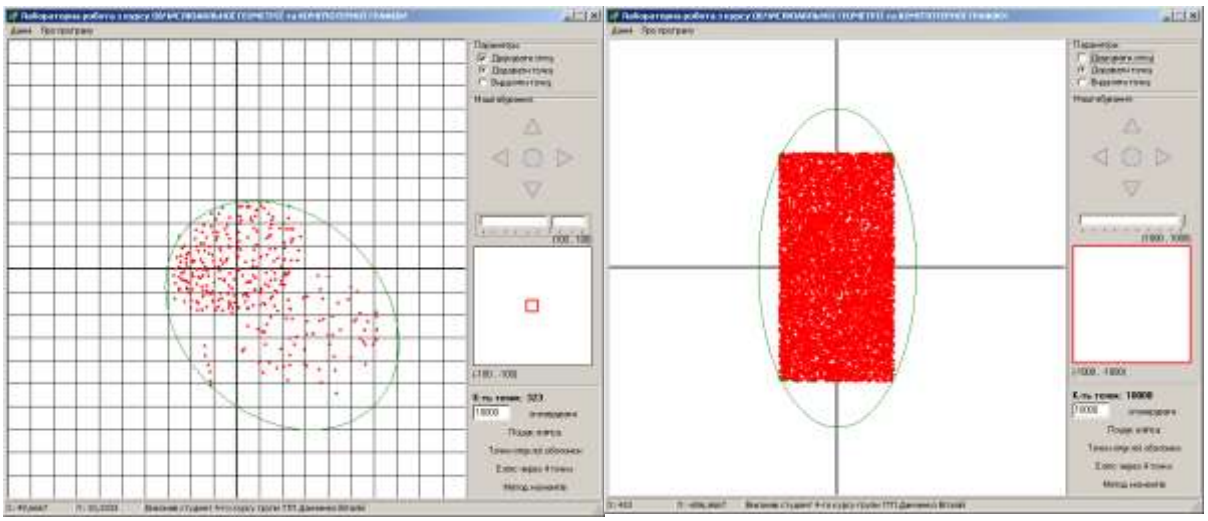


Рис. 4.2

Оцінка складності алгоритму

Теорема. 4.1. *Еліпс найменшої площі на множині S із n точок на площині можна побудувати розглянутим методом час O*(n log n) *з використанням O*(n) *пам'яті.*

Доведення. Для пошуку точок опуклої оболонки, використовувався метод Грехема [29]. Як відомо складність даного метода $O(n\log n)$. Для пошуку найвіддаленіших точок потрібний лінійний час $O(n)$[27]. Для знаходження решти двох точок теж вистачає одного проходу множини тобто часу $O(n)$. Можна зробити наступний висновок, що навіть у найгіршому випадку, коли всі задані точки будуть точками опуклої множини час виконання алгоритму становить $O(n\log n)$, таб. 3.

Операція	Час виконання
Пошук внутрішньої точки множини	$O(n)$
Лексографічне сортування за полярним кутом	$O(n\log n)$
Пошук діаметра опуклої оболонки	$O(i)$
Пошук трикутників максимальної площі	$O(i)$
Побудова еліпса	$O(1)$

Таблиця 3.

Отже, час виконання алгоритму $O(n\log n)$.

Висновки

Запропоновано та реалізовано алгоритм побудови еліпса мінімальної площі, описаного навколо заданої на площині множини точок. Даний алгоритм працює за оптимальний час $O(n\log n)$.

4.2 Алгоритм побудови з використанням опуклої оболонки (варіант2).

Алгоритм (основна ідея)

В загальному випадку еліпс задається загальним рівнянням кривої другого порядку $ax^2+bxy+cy^2+dx+ey+f=0$. Єдине, що для еліпса f=1. Таким чином ми бачимо 5 невідомих, і для того, щоб побудувати еліпс необхідно 5 точок.

Твердження 1: *Навколо опуклого 5-ти кутника завжди можна описати еліпс, такий, що точки 5-ти кутника належать еліпсу. Такий еліпс буде єдиним і найменшим охоплюючим еліпсом для даного 5-ти кутника.*

Доведення випливає з єдиного розв'язку системи лінійних рівнянь, де замість незалежних змінних підставляється координати вершин 5-ти кутника.

Побудова найменшого охоплюючого еліпсу навколо 3-х точок. Еліпсів, які описані навколо трьох точок існує нескінченна кількість, серед них існує лише єдиний найменшої площі. Для того, щоб його знайти треба розв'язати систему лінійних

рівнянь, обмеживши умовою мінімізації площі. Для випадку, коли центр еліпсу лежить в початку координат і його осі симетрії паралельні осям координат, $S_{ел}=2ab$. Для того, щоб знайти площу еліпса довільно розташованого по відношенню до системи координат треба перенести початок координат в центр еліпса, методом невизначених коефіцієнтів розв'язати рівняння перенесення, таким чином отримати ще два рівняння до системи лінійних рівнянь. Можна зробити підсумок, що існує алгоритм побудови найменшого еліпсу, що проходить через 3 точки.

Перевірка розташування точки відносно еліпса. Для цього достатньо підставити координати точки в рівняння еліпса. Якщо результат = 0 – точка належить еліпсу; якщо результат > 0 – точка лежить поза еліпсом; якщо результат < 0 – точка лежить всередині еліпса.

Твердження 2: *Найменший охоплюючий еліпс, побудований навколо множини точок співпадає з найменшим охоплюючим еліпсом, побудованим навколо опуклої оболонки.*

Доведення. Це твердження доводиться просто. Так, як опукла оболонка містить усі точки множини *S,* включаючи і вершини опуклої оболонки *CH(S)*, а опукла оболонка - це найменша опукла множина, яка містить множину *S*, то еліпс побудований для опуклої оболонки *CH(S)* буде НОЕ для множини *S*.

Алгоритм (покроково)

I етап: будується опукла оболонка для *N* заданих точок.

II етап: побудова найменшого охоплюючого еліпсу в свою чергу складається з декількох кроків:

Крок 1 : серед отриманих на етапі точок знаходимо трикутник з максимальною площею. Ці три точки записуємо до множини *Z*.

Крок 2 : Для поточної множини *Z* будуємо найменший охоплюючий еліпс. Ця процедура займає фіксований час. Перевіряємо решту точок многокутника. Якщо всі вони належать еліпсу – алгоритм завершує свою роботу, побудований еліпс є шуканим. Відкидаємо всі точки, що лежать всередині еліпса. Перевірка займає O(*N*). Якщо ні – переходимо до Кроку 3.

Крок 3 : Маємо найменший охоплюючий еліпс і множину точок *S*, яка містить точки, що лежать поза еліпсом. Множина *Z* містить не більше ніж 5 точок. Серед усіх точок, що лежать поза еліпсом знаходимо будь-яку і додаємо до множини *Z*. Перевіряємо множину *Z* на повноту (оскільки множина Z може містити 6 точок, то знайдеться точка, яка буде зайвою, тобто для поточного еліпсу вона буде лежати всередині) і переходимо на Крок 2.

Оцінка складності алгоритму

Теорема. 4.2. *Еліпс найменшої площі на множині S із n точок на площині можна побудувати розглянутим методом за час* $O(n^2)$ *з використанням* $O(n)$ *пам'яті.*

Доведення. Побудова опуклої оболонки - $O(n\log n)$. Пошук найбільшого за площею трикутника $O(n\log n)$. Крок 2 і Крок 3 повторюються ітеративно. Кількість ітерацій

для найгіршого випадку буде n, на кожному кроці виконується $O(n)$ операцій, тому загальна складність буде $O(n^2)$.

4.3 Алгоритм побудови з використанням опуклої оболонки (варіант3).

Під час дослідження можливих методів розв'язання задачі цікавим є формулювання даної проблеми ("Задача побудови опуклої оболонки та її покриття") та метод її розв'язання представлене у монографії Л.Ф. Тота [30]. Для покриття опуклої оболонки к Л.Ф. Тот пропонує ефективний та швидкий алгоритм, але одразу ж виникає запитання, що, в контексті даної задачі, вважати опуклою оболонкою, яку необхідно покрити, та як її сформувати. Якщо сформувати опуклу оболонку навколо точок в просторі, то покриття такої оболонки неодмінно утворювало б "прогалини", тобто круги, які не покривають жодної точки, але входять в покриття опуклої оболонки.

Алгоритм (основна ідея)

1. На першому кроці алгоритму будуємо на множині точок опуклу оболонку за допомогою метода Грехема.
2. Далі повертаємось до нашого алгоритму і находимо на опуклій оболонці пару точок з максимальною відстанню – це і буде велика вісь еліпса.
3. Знаходимо центр цієї вісі . Зсуваємо всі вісі і робимо відповідний поворот таким чином, щоб положення абсциси та ординат стали (0,0) .
4. Для кожної точки, окрім двох, що в результаті з'єднання утворили велику вісь еліпса, знаходимо максимальну відстань від центра вісі і з'єднуємо. Точка повинна лежати на еліпсі.

Оцінка складності алгоритму

Теорема. 4.3. *Еліпс найменшої площі на множині S із n точок на площині можна побудувати розглянутим методом за час $O(n\log n)$ з використанням $O(n)$ пам'яті.*

Доведення. В даному випадку, швидкість попередньої обробки визначається швидкість сортування точок за координатою x ($O(n\log n)$. Другий етап алгоритму можна виконати з витратами $O(\log n)$ часу. Третій та четвертий етапи виконуються за $O(n)$ часу. Таким чином отримаємо загальну складність алгоритму $O(n\log n)$.

Алгоритм (реалізація)

Аналіз середовища та мови реалізації. Оптимальним середовищем для реалізації даного проекту можна обрати JAVA (JRE), через його кросплатформність. Таким чином програмний модуль зможе виконуватись як на комп'ютерах з операційною системою Windows так і на машинах з операційною системою типу UNIX.
Можливості, програмне та технічне забезпечення реалізації. Для запуску програмного модуля, на комп'ютері користувача має бути встановлено віртуальну машину JAVA (JVM) версії не нижче 5.0. Комп'ютер має бути обладнаний не менш ніж 64 мб оперативної пам'яті, мати частоту процесора не нижче 700 мгц та частоту

шини не меншу за 333 мгц. На рис. 4.3 подано приклад реалізації, а у додатку 4 представлено варіант коду для реалізації.

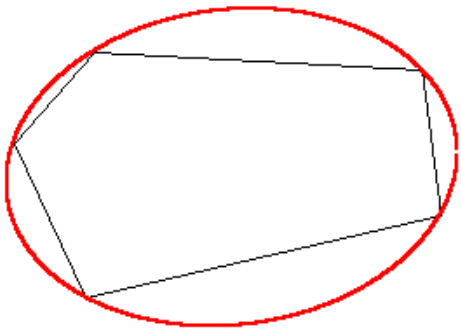


Рис. 4.3. Приклад реалізації

ЗАДАЧІ НА ВПИСАННЯ

5. Побудова кола найбільшого радіусу вписаного в опуклу оболонку (ВМКО)

На сьогоднішній час існують два основних підходи до розв'язання цієї задачі: використання методу подвійного тернарного пошуку [31] та метод «стискання сторін» [32, 33]. Більш ефективним є другий підхід так, як він не залежить від вибору системи координат та вимог до точності.

Постановка Задачі. *Для заданої N – вершинної опуклої оболонки на площині вписати коло найбільшого радіусу.*

Методи розв'язання

5.1 Алгоритм на основі методу «стискання сторін»

Розглянемо деякі основні означення.

Означення. 5.1. Під **бісектрисою** для двох ребер будемо розуміти пряму, кожна точка якої є рівновіддаленою від цих ребер.

Для випадку непаралельних ребер це відповідатиме бісектрисі кута між цими ребрами.

Означення 5.2. ***Лівою (правою) бісектрисою ребра*** називатимемо бісектрису між поточним ребром та наступним (попереднім) у списку ребер.

Означення 5.3. ***Висотою*** ребра будемо називати відстань від точки перетину його лівої та правої бісектрис до цього ребра.

Алгоритм (основна ідея)

Теорема 5.1. *Якщо у N – вершинному многокутнику видалити ребро з найменшою висотою, то розв'язок задачі не зміниться.*

Доведення. Будемо зсувати ребра многокутника на однакову відстань всередину, (рис. 5.1). Вершини будуть пересуватися по бісектрисам кутів многокутника. Ми отримаємо новий зменшений многокутник.

Доведемо, що при зсуві на h максимальне коло для нового многокутника матиме той самий центр і радіус менший на h. Припустимо, що ми побудували максимальне коло для нового многокутника з радіусом r . Збільшимо його радіус $R = r + h$. Якщо відстань до ребер від центру кола у новому многокутнику дорівнювала h_i, то відстань до відповідних ребер у старому буде h_i+h.

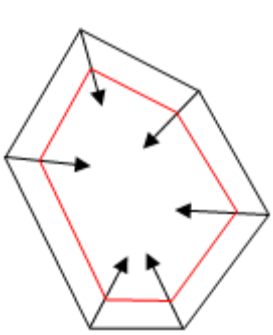


Рис. 5.1

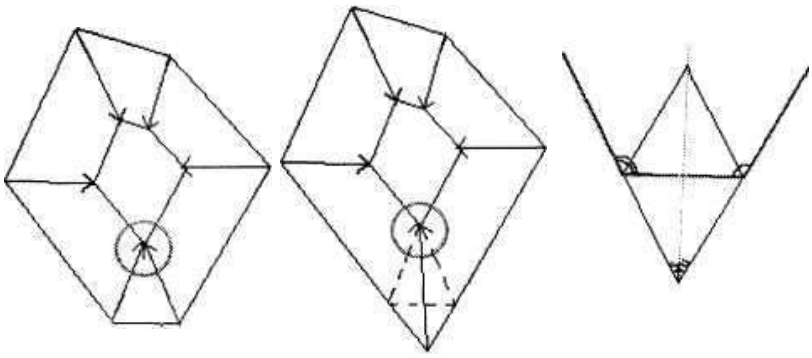


Рис. 5.2

Оскільки радіус також змінювався на h, то ті ребра які дотикалися у новому многокутнику будуть дотикатися до кола і в старому ($h_i = r \rightarrow h_i + h = r + h = R$). Очевидно також, що оскільки в новому многокутнику ребра не перетинали коло ($h_i < r$), то і в старому вони також не перетинатимуть його ($h_i + h < r + h = R$). Отже максимальне коло для зменшеного многокутника, буде вписаним колом для старого, якщо збільшити його радіус на h.

Доведемо тепер, що воно буде максимальним для початкового многокутника. Припустимо супротивне. Нехай існує інше коло вписане у початковий многокутник, з радіусом R' більшим за R. Тоді зменшимо його радіус на h. З аналогічних вищенаведеним міркувань отримане коло буде вписаним у зменшений многокутник. Але ж $r' = R' - h > r = R - h$. Це суперечить нашому припущенню, що ми побудували максимальне коло в зменшеному многокутнику з радіусом r. Отже при зсуві ребер на h всередину многокутника, центр максимального вписаного кола не зміниться, а радіус зменшиться на h. При зсуві на величину рівну мінімальній висоті ребра многокутника, відповідне ребро вироджується у точку (рис.5.2). Якщо ми посунемо ребра назад знову на h, то ми отримаємо старий многокутник з видаленим одним ребром. Таким чином ми звели задачу побудови максимального кола в многокутнику з N вершинами до задачі побудови кола для многокутника з N-1 вершинами. Другий факт, що використовувався у обчисленні, це те, що бісектриса новоутвореного кута при видаленні ребра, проходить через перетин бісектрис кутів між цим ребром та його сусідами.

Алгоритм (покроково)

1. Нехай побудована N – вершинна опукла оболонка на площині одним із відомих методів (наприклад методом Грехема).
2. Алгоритм побудови кола.
2.1. Ініціалізація лівих бісектрис для кожного ребра (вони будуть просто бісектрисами кутів многокутника).
2.2. Обрахування висот для кожного ребра.
2.3.Знаходження ребра з найменшою висотою.
2.4. Видалення ребра з найменшою висотою з множини.
2.5. Перерахування лівої бісектриси для правого сусіда цього ребра.

2.6. Перерахування висот для лівого та правого сусідів цього ребра.
2.7. Якщо кількість ребер >3 то перейти на пункт 2.4, інакше на 2.8.
2.8. Для трьох ребер, що залишилися будується коло вписане у трикутник, отриманий з прямих, що визначаються цими ребрами. Це коло і буде розв'язком.

Оцінка складності алгоритму

Теорема. 5.2. *Коло максимального радіусу можна вписати в* n *- вершинну опуклу оболонку методом стискання сторін за час* $O(n^2)$ *з використанням* $O(n)$ *пам'яті.*

Доведення. Початкова ініціалізація відбувається за лінійний час. Зменшення кількості вершин відбувається n-3 рази. Кожен раз нам потрібно знайти і видалити ребро з мінімальною висотою, потім перерахувати одну ліву бісектрису (для сусіднього ребра) і перерахувати висоти для обох сусідніх ребер. Можна встановити такі часові оцінки:
1) Пошук мінімальної висоти по списку ребер відбувається за час O(n).
2) Перерахування бісектриси, висот сусідів та сусідніх ребер за константний час.
3) Видалення ребра зі списку за час O(n).

Отже один цикл видалення проходить за час O(n). Оскільки таких циклів n-3, то загальна оцінка часу O(n^2). Оцінка пам'яті – O(n).

Алгоритм (реалізація)

На рис. 5.3 , 5.4 подано приклади програмної реалізації для різної кількості точок у двох форматах, відповідно.

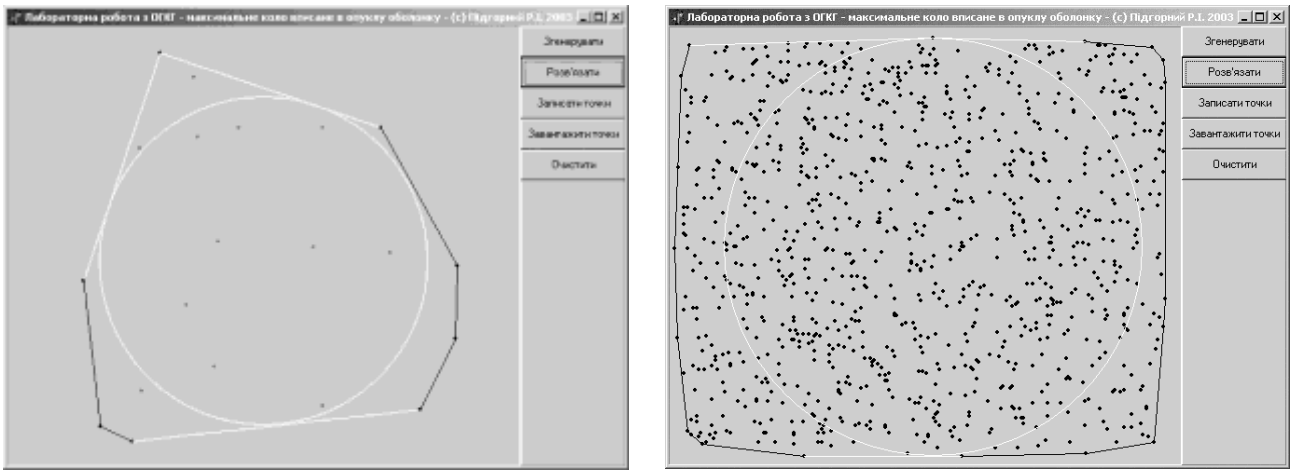
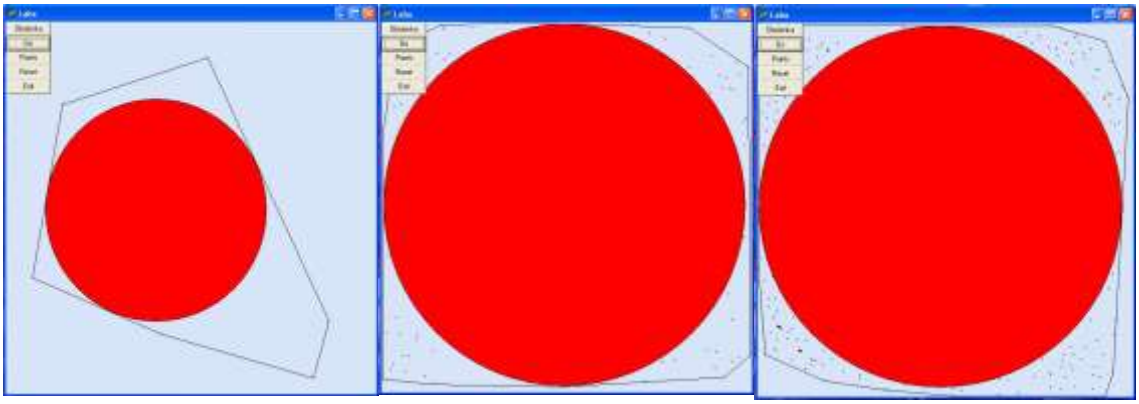


Рис.5. 3. Формат1. Приклади реалізації для різної кількості точок.



10 точок 100 точок 1000 точок

Рис.5. 4. Формат2. Приклади реалізації для різної кількості точок.

Висновки

Запропонований алгоритм розв’язує задачу пошуку максимального вписаного кола в опуклу оболонку за час $O(n^2)$. Алгоритм не є досконалим і має деякі вади: унаслідок перерахування нових бісектрис на основі старих бісектрис, можливе накопичення помилки обчислення; при наявності майже паралельних бісектрис можливе переповнення при обчисленні їх перетину.

5.2 Алгоритм на основі подвійного тернарного пошуку.

Розглянемо інший варіант ідеї стискання сторін границі опуклої оболонки для розв’язання задачі вписання в опуклу оболонку кола найбільшого радіусу.

Алгоритм (основна ідея)

Почнемо одночасно зміщувати сторони паралельно самим собі в середину, рис. 5.5.

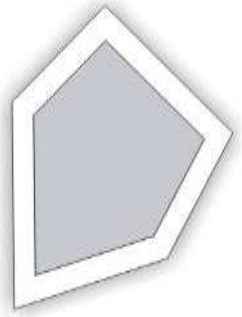


Рис.5.5.

Нехай для зручності ця швидкість дорівнює 1 координатній одиниці в секунду (тобто час рівний відстані в деякому розумінні). В процесі цього руху сторони цього многокутника будуть поступово зникати (перетворюватися в точки). Таким чином рано чи пізно весь наш многокутник стиснеться в точку чи відрізок і цей момент часу t і буде шуканим радіусом, а центр буде лежати на цьому відрізку. Справді, якщо ми стиснули наш многокутник на товщину t по всім напрямках, то існує точка,

що лежить на відстані t від усіх сторін, а для більших відстаней – такої точки не існує, бо тоді ми б мали можливість стиснути многокутник принаймні ще один раз.
Знайдемо час для кожної сторони, за який вона стискається в точку. Наші вершини заданого многокутника рухаються по бісектрисам відповідних кутів (це слідує з рівності відповідних трикутників). Тоді питання про час, через який сторона остаточно стиснеться, зводиться до питання знаходження висоти H трикутника, в якому відомі сторона L та прилеглі кути α та β. З теореми синусів маємо:

$$H = L \cdot \frac{sin\alpha \cdot sin\beta}{\sin(\alpha + \beta)} \qquad (5)$$

Таким чином, за O(1) ми можемо визначити час, за який сторона стиснеться в точку. Тепер створимо деяку структуру даних, за допомогою якої будемо визначати мінімум.

Алгоритм (покроково)

1. Знайдемо сторону з найменшим часом H – ця сторона першою стиснеться в точку.
2. Якщо многокутник ще не стиснувся в точку або відрізок, то необхідно видалити цю сторону, і продовжити перегляд для сторін, що залишилися. Причому, при видаленні сторони необхідно з'єднати правого та лівого сусіда, продовживши сторони до їх взаємного перетину, а потім перерахувати довжини двох нових сторін та час їх зникання і продовжити перегляд.
3. Якщо на деякому кроці після видалення сторони її сусіди-сторони паралельні, то це означає, що після цього стиснення многокутник виродиться в точку, а тому результатом (радіусом) є час, за який зникла остання сторона.
4. Інакше (відсутність паралельності сторін) на деякому етапі залишиться 2 сторони, і відповіддю буде час зникнення попередньої сторони.
5. Нарешті, знаходимо центр кола. Якщо многокутник стиснувся в точку, то ця точка і є його центром, якщо у відрізок – то будь-яка точка з цього відрізку, наприклад, будь-який з його кінців, якщо після видалення на останньому кроці сторони залишилося 2 непаралельні сторони – то це буде центр перетину бісектрис у трикутнику на передостанньому кроці (знаходимо за відомою формулою) [34], і, нарешті, при умові, що утворилися 2 паралельні сторони, то, як було сказано вище, на наступному кроці многокутник стиснеться в точку – відповідно вона і буде центром заданого кола.

Оцінка складності алгоритму

Теорема 5.3. Часова складність задачі знаходження найбільшого вписаного кола в опуклу оболонку за допомогою алгоритму подвійного теранарного пошуку складає $O(n\log n)$.

Доведення. Як вже було розглянуто вище часова складність алгоритму Грехема складає $O(n\log n)$. Таким чином, розглянемо задача про побудову кола, що вписане в опуклий многокутник. Даний алгоритм у загальному випадку складається з n кроків, на кожному з яких відбувається видалення однієї сторони – для цього виконуються декілька операцій в структурі даних для визначення мінімуму, на їх виконання необхідно $O(n\log n)$ часу. Оскільки зберігаємо лише подвійний список сусідів для

кожний вершини, то пам'ять – лінійна. Отже, загальна складність розглянутого алгоритму складе рівно $O(n\log n)$ часу у загальному випадку.

Алгоритм (реалізація)

Особливості програмної реалізації та вводу - виводу даних.
Програма реалізована мовою C++ з використанням стандартних бібліотек та потоків. У програмній реалізації використовуються два основних алгоритми, що описані вище: метод Грехема та метод «стискання сторін». Створена програма спочатку приймає на вхід задані точки (чи генерує задану кількість точок в певному діапазоні – для вибору кожного з режимів присвоєно перемикач). Далі виконується побудова лінійної оболонки за 3 вищезгаданих кроки: знаходимо точку з найменшою x-координатою, якщо декілька – найнижчу (та для спрощення реалізації знаходимо також з точку з найбільшою x – координатою (якщо декілька - найвищу), розбиваємо на дві півплощини прямою, що з'єднує ці точки і для кожної півплощини виконуємо дії, що написані далі – обидві точки, як початкові, включаємо в оболонку), сортуємо точки, порівнюючи їх між собою і, відповідно, перенумеровуючи їх, упорядковуючи їх відносно початкової точки, з'єднуємо їх почергово; нарешті виключаємо точки, що не входять в оболонку, перевіряючи почергово для трьох точок утворений кут за допомогою орієнтованої площі.
Далі ми вже маємо деякий опуклий многокутник і виконується метод «стиснення сторін». Алгоритм цього методу описаний вище. Основна ідея – поетапне видалення сторін з найменшим часом на видалення, продовження сусідів до перетину і перевірка на паралельність чи остаточне стиснення у відрізок чи точку – умови зупинки. Після цього у загальному вигляді розглядаємо всі можливі випадки остаточного стиснення і для нашого конкретного випадку скористаємось відповідною опцією для знаходження центру кола. Після цього програма видає результат користувачу у заданому форматі у файл output.txt (приклад у Додатку 2). Лістинг програми наведений нижче – у Додатку 1.

Опис основних функцій програмної реалізації.

Тут і надалі буде опис лише основних функцій(функції простої перевірки та взаємодії з користувачем описані детально в лістингу програми, що знаходиться в Додатку 1).
void convex_hull (vector<pt> & a) – для заданого набору точок виконує побудову опуклої оболонку методом Грехема.
double get_h (const pt & p1, const pt & p2, const pt & l1, const pt & l2, const pt & r1, const pt & r2) – визначає значення H для кожної сторони побудованої опуклої оболонки.
Обчислення значення радіусу відбувається вже в основній функції програми. Потім функція **void checkCenter(vector<pt> p),** що знаходить центр шуканого кола.

Висновки

У роботі було розглянуто алгоритм вписання кола найбільшого радіусу в опуклу оболонку. Даний алгоритм не має обмежень на кількість точок на вхід (залежить

лише від обчислювальної потужності відповідного пристрою) і тому він застосовується для розв'язування даної задачі у найзагальнішому випадку. Виявилось, що даний алгоритм може бути ефективно застосований для вирішення задач з обчислювальної геометрії. Щоб скористатися створеною програмою необхідно лише задати точки або відповідно перемкнути в режим автоматичної генерації (необхідно задати бажану кількість точок). Далі програма створить опуклу оболонку, а потім знайде вписане коло максимального радіуса, і на виході користувачу буде показано радіус та центр цього кола. Як було доведено вище, час роботи у загальному випадку складає O(nlogn).

6. Трикутник найбільшої площі вписаний в опуклу оболонку (ВМТО (АО31))

Постановка Задачі. *Для заданої n– вершинної опуклої оболонки на площині вписати трикутник найбільшої площі.*

Методи розв'язання

6.1 Алгоритм простого перебору

Алгоритм (основна ідея)

Теорема 6.1. *Вершини трикутника максимальної площі, вписаного в опуклу оболонку, співпадають з вершинами опуклої оболонки.*

Доведення. Нехай задана множина точок *S*, conv(*S*) – опукла оболонка множини S, CH(*S*) – границя опуклої оболонки множини S. Нехай трикутник ABC – трикутник максимальної площі, який вписаний в опуклу оболонку conv(*S*). Спочатку доведемо, що точки A, B, C$\in$CH(*S*).

1. Припустимо, що вершина A $\notin$CH(*S*), при цьому не важливо чи лежать вершини B та C на границі опуклої оболонки.

2. Продовжимо сторону AB до перетину з CH(*S*) і позначимо точку перетину A',рис.6.1. Отриманий трикутник A'BC має більшу площу, ніж трикутник ABC.

3. Якщо точки B$\notin$CH(*S*) та C$\notin$CH(*S*), то їх аналогічно можна перемістити в точки B'$\in$CH(*S*) та C'$\in$CH(*S*) відповідно. Отриманий трикутник A'B'C' має більшу площу, ніж початковий трикутник ABC.

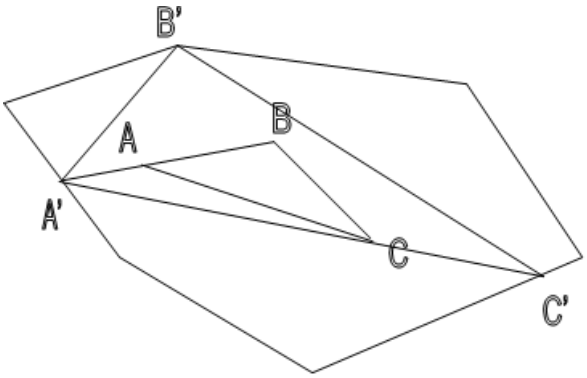


Рис.6.1

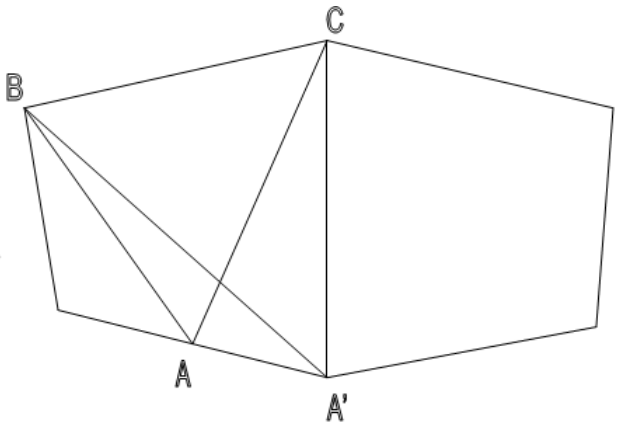


Рис.6.2.

Тепер доведемо, що точки A, B та C співпадають з вершинами опуклої оболонки множини *S*.

1. Припустимо, що точка A не співпадає з вершиною опуклої оболонки, рис.6.2. Якщо ребро, яке містить точку A, не паралельне відрізку BC, то точку A можна пересунути в вершину цього ребра так, щоб збільшилась висота, опущена з точки A на відрізок BC. Площа трикутника A'BC більша за площу трикутника ABC.
2. Якщо ребро, яке містить точку A, паралельне відрізку BC, то можна перемістити точку A у вершину опуклої оболонки. При цьому довжина висоти, опущеної з точки A на відрізок BC не зміниться, а тому площа трикутника ABC також не зміниться.
3. Аналогічно можна перемістити точки B та C у вершини опуклої оболонки, якщо вони там не лежать.

Алгоритм (покроково)

1. Побудуємо опуклу оболонку множини *S*.
2. Шукаємо трикутник найбільшої площі перебором всіх можливих трійок вершин опуклої оболонки.

Оцінка складності алгоритму

Теорема 6.2. *Вписання трикутника в n-вершинну опуклу оболонку запропонованим методом можна виконати за час* $O(n^3)$.

Доведення. Для пошук трикутника максимально площі, вписаного в опуклу оболонку реалізований методом перебору всіх можливих трикутників. Це вимагає $O(k^3)$ операцій, де k – кількість вершин опуклої оболонки множини S. Таким чином, складність алгоритму в найгіршому випадку $O(n^3)$. Але в середньому алгоритм працює швидше, оскільки в більшості випадків $k << n$.

Алгоритм (реалізація)

Основні функції програми:

1. **list *opukla_obolonka(list *p) -** функція отримує на вхід список точок множини S, на вихід видає впорядкований список точок, який утворює CH(S)

2. **void max_tr(list *l, int &i, int &j, int &k) -** функція отримує на вхід список точок множини S, який утворює CH(S). Після виконання цієї функції в змінних i, j, k зберігаються номери вершин шуканого трикутника.

3. **void drawOO(list *l) -** функція зображає опуклу оболонку множини S.

4. **void drawTr(list *l) -** функція зображає трикутник максимальної площі, який вписаний в опуклу оболонку.

В програмі передбачена можливість як ручного вводу точок, так і автома-тичного. Автоматичний спосіб дозволяє ввести 10000 точок.

Висновки

Недоліком алгоритму є велика часова складність роботи алгоритму. Однак це залишає можливість для підвищення ефективності програми. Один з можливих шляхів оптимізації – використання властивості опуклості опуклої оболонки, що дасть бінарний пошук. Якщо наперед відомо, що кількість вершин опуклої оболонки k набагато менша за кількість точок опуклої оболонки n, то для побудови опуклої оболонки краще використати метод Джарвіса, а не метод Грехема. Це дасть змогу пришвидшити побудову опуклої оболонки.

6.2 Алгоритм бінарного пошуку вершин максимального трикутника

Враховуючи теорему 6.1 можна спробувати покращити оцінку складності попереднього алгоритму до $O(n\log n)$ завдяки представленню вершин опуклої оболонки у вигляді відсортованого за полярним кутом списку вершин.

Алгоритм (основна ідея)

Нехай маємо n вершинну опуклу оболонку CH($S(n)$). Представимо вершини опуклої оболонки у вигляді центрально впорядкованого списку вершин за полярним кутом відносно центроїда (т. O) CH(S) = {1, 2, 3, 4, 5, 6, 7, 8, 9 }, рис.6.3. За центроїд можемо взяти перші вершини списку 1, 2, 3.

Знаходження трикутника розіб’ємо на 2 етапи. На першому етапі знаходяться всі локальні максимуми, тобто такі трикутники, для яких фіксуємо будь-які 2 їх вершини, а 3-ю ми не зможемо зсунути по списку вліво чи вправо, щоб збільшити площу. На другому етапі серед знайдених на першому етапі трикутників обираємо трикутник з найбільшою площею.

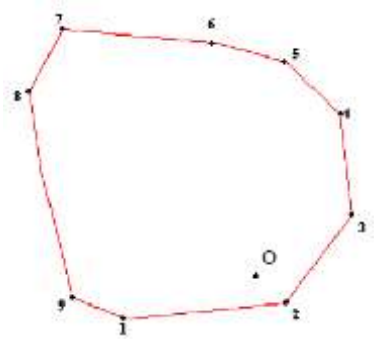


Рис. 6.3

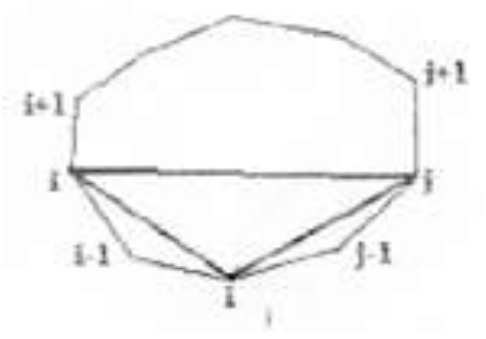


Рис. 6.4

Перший етап.

1. Нехай маємо фіксовану вершину, наприклад вершину 1, рис. 6.3.

2. Обираємо інші дві вершини як [(*n*-1)/4] і 3[(*n*-1)/4]. Ці вершини є серединами половини вершин опуклої оболонки, що лишилися.

3. Рухаємо точки *i* та *j* ліворуч та праворуч від фіксованої точки 1, рис. 6.4.

3.1. Для ребра (1, *i*) визначаємо відстань до точок *j*, *j*-1, *j*+1.

3.1.1 Якщо відстань у точці *j* більша за відстань у точках *j*-1 і *j*+1, то точка *i* залишається нерухомою.

3.1.2 Якщо відстань у точці *j*-1 більша за відстані в точках *j*, *j*+1 то рухаємося ліворуч (у напрямку *j*-1, 1).

3.1.3 Якщо ж відстань у точці *j*+1 більша за відстані в точках *j*, *j*-1 то рухаємося праворуч (у напрямку *i*+1, *j*+1).

3.2 Для ребра (1, *j*) визначаємо відстань до точок *i*, *i*-1, *i*+1.

3. 2.1 Якщо відстань у точці і більша за відстань у точках *i*-1 та *i*+1, то точка *j* залишається нерухомою.

3.2.2 Якщо відстань у точці *i*-1 більша за відстані в точках *i*, *i*+1 то рухаємося ліворуч (у напрямку *i*-1, 1).

3.2.3 Якщо ж відстань у точці і+1 більша за відстані в точках *i*, *i*-1 то рухаємося праворуч (у напрямку *j*+1, *i*+1).

Наприклад ми вибрали перші дві вершини якимось чином. Нехай це будуть 1 і 2. Тоді з вершин 3.9 шукаємо таку, щоб відстань від відрізка 1-2 були найбільшою. Оскільки площа трикутника – це ½ (основа*висоту), то це є важливим. Ця відстань шукається, якщо відомі координати точок.

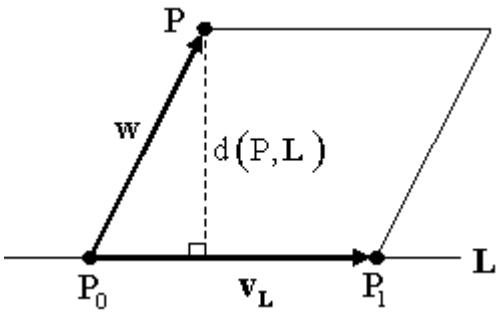


Рис. 6.5

$$\mathbf{d(P,L)} = \frac{(y_0 - y_1)x + (x_1 - x_0)y + (x_0 y_1 - x_1 y_0)}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}} \qquad (6)$$

Позначимо d(P,L) як відстань від очки P до відрізка L, координати x_0, y_0 початку і кінця відрізку x_1, y_1, і координати точки P(*x*, *y*) відповідно, рис. 6.5.

Рухаємо точки одночасно, наступні точки і та j будуть такими, рис. 6.6:

1. Точкі і та j обидві рухаємо праворуч, наступна ітерація буде вигляду: [(pri-i)/2] та [(j-prj)/2] - праворуч; [(i-pli)/2] та [(plj-j)/2] –ліворуч. Де pri та prj, pli та plj - попередні обмеження руху для точок і та j, відповідно праворуч та ліворуч. На першому кроці ці обмеження мають значення: pri = [(n-1)/2] та pli =plj =1.

2. Точку і рухаємо праворуч, а точку j - ліворуч, наступна ітерація буде вигляду: [(pri-i)/2] та [(plj-j)/2] –ліворуч.
3. Якась із точок фіксується , а якась рухається далі – виконується дія аналогічна попередній, лише змінюються plj або prj (в залежності від випадку)- попередні обмеження руху для точок і та j, відповідно праворуч та ліворуч. На першому кроці ці обмеження мають значення: pri = [(n-1)/2] та pli =plj =1.

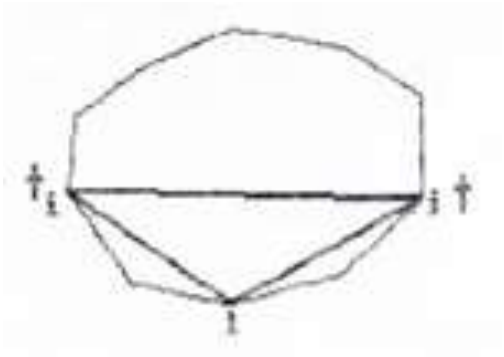


Рис.6.6.

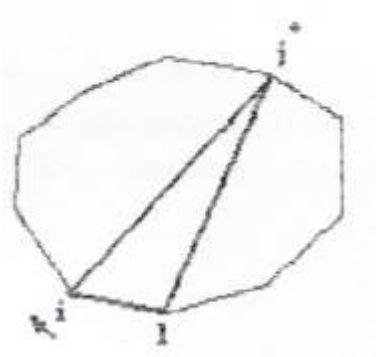


Рис.6.7.

4. На кожному кроці поновлюємо pri, prj, pli, plj присвоюючи *a* попередні значення *i* та j, рис. 6.7. Процес зупиняється за рахунок опуклості множини.

Другий етап.

На цьому етапі нам необхідно знайти всі такі локальні максимуми. Для цього беремо по черзі усі *n* точок опуклої оболонки і для кожної виконуємо процедуру пошуку максимального трикутника. Їх кількість буде не більше ніж *n*/3. Це лише в тому випадку, коли всі вершини границі опуклої оболонки будуть лежати на колі, або – оболонка нагадуватиме коло (рівносторонній многокутник). У результаті ми отримаємо для усіх вершин *n* трикутників. З поміж них будуть трикутники двох типів: максимальні трикутники та такі трикутники, для яких будь-які 2 фіксовані їх вершини, і 3-ю ми не зможемо зсунути по списку вліво чи вправо, щоб збільшити площу. Чим більш оболонка буде сплюснута за формою, тим менше в ній локальним максимумів. Щоб знайти їх всі досить обрати якусь вершину і виконати пошук локальних максимумів для основ з цією вершиною та всіма іншими, яких буде *n*-1. Цього буде досить, оскільки основа пройде наближено всі кути нахилу однієї з 3-х сторін локального максимуму, до якого процедура пошуку і приведе.

Оцінка складності алгоритму

Теорема 6.3. *Вписання трикутника в n-вершинну опуклу оболонку запропонованим методом можна виконати за час O(nlogn).*

Доведення. Пошук точки відбувається зі складністю $O(\log n)$, оскільки точки упорядковані то пошук дихотомічний, з деякою специфікою, що базується на наступному : якщо в одній з точок відстань більша ніж в другій, що є сусідньою, значить точка з найбільшою відстанню буди лежати праворуч від першої. Знайшовши таку точку, на нашому малюнку – відповідно – 7. фіксуємо її. Далі

фіксуємо або 1 або 2, нехай 2, тоді для точок 2 і 7 робимо ту ж процедуру. Ця процедура буде виконуватись не більше C раз. А отже складність C*O(log$n$) = O(log$n$).

Отже загальна складність алгоритму = O($n$log$n$) + O(log$n$)*(n-1) = O($n$log$n$)!

7. Прямокутник найбільшої площі вписаний в опуклу оболонку (ВМПО (АО41))

Проблема пошуку прямокутника найбільшої площі вписаного в опуклу оболонку часто виникає при спробі виправлення спотворення від зміщеного проектора. Щоб отримати найкращий ортогональний прямокутник на проекційній поверхні, ми повинні вибрати в межах проектованої площі. Моделюючи проектовану область як опуклий многокутник, ми можемо вирішити проблему, обчисливши найбільший вписаний ортогональний прямокутник. У той же час, якщо декілька проекторів об'єднані, щоб сформувати великий екран, многокутник, що є результатом об'єднання всіх проектованих областей, може не бути опуклим. Ми будемо розглядати саме вписання найбільшого прямокутника в опуклу оболонку на площині. Існує декілька алгоритмів, які дозволяють розв'язувати цю задачу з різною складністю від повного перебору до O(log$n$) [5, 29, 35-42]. Розглянемо деякі із них, які поряд із високою ефективністю дозволяють нескладну реалізацію. Зокрема розглянемо алгоритми із часовою складністю O($n$log$n$) та з часом O(log$n$) [38]. Варто звернути увагу також на алгоритм [39] з часом O(n) та алгоритм [41] з часом $O(\log^2 n)$.

Постановка Задачі. *Для заданої n– вершинної опуклої оболонки на площині вписати прямокутник найбільшої площі.*

Методи розв'язання

7.1 Алгоритм бінарного пошуку вершин максимального трикутника

У цьому алгоритмі пропонується ідея розв'язання задачі за допомогою стратегії «розділяй та володарюй» та діаграми Вороного [43].

Алгоритм (основна ідея)

Нехай маємо опуклу оболонку для заданої множини S із n точок задану границею CH(S) ={ P_1, P_2, …, P_n}. Оскільки прямокутник фактично задається однією точкою на границі CH(S) , ми пересуваємо цю точку по її ребрам згідно наступного алгоритму.

Алгоритм (покроково)

1. Розбиваємо ланку границі CH(S) від лівої до верхньої вершини на проміжки незмінності значення функції площі r_i. Ця функція буде квадратичною функцією від X.
2. Для кожного проміжку знаходимо значення площі.
3. Знаходимо її екстремум і, якщо він належить проміжку r_1r_2, враховуємо й точку де досягається максимум як можливого кандидата. Тобто головна ідея в тому що ліва точка шуканого прямокутника, буде або одним з кінців якогось з проміжків, або буде належати проміжку і при цьому буде максимумом функції площі на цьому проміжку, рис. 7.1.

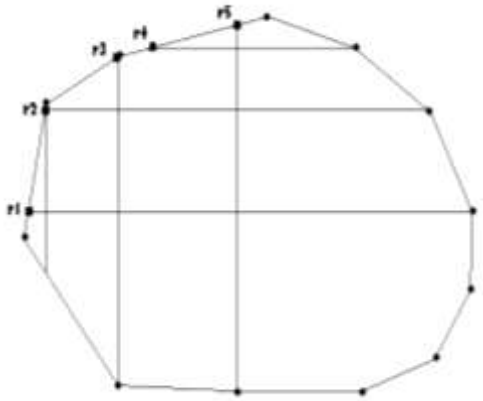


Рис. 7.1.

4. Двійковим пошуком знаходимо інші вершини прямокутника.
5. Порівнюємо його з найбільшим на даний момент прямокутником.
6. Відображаємо многокутник відносно осей координат, кожен раз запускаючи алгоритм, задля проходу не лише справа наліво по верхній ланці, а й усіма можливими способами.

Оцінка складності алгоритму

Теорема7.1. *Вписання прямокутника в n-вершинну опуклу оболонку запропонованим методом можна виконати за час O(nlogn).*

Доведення. Складність цього алгоритму O($n$log$n$), тому що для кожної вершини n запускається двійковий пошук за час O(log$n$).

Алгоритм (реалізація)

Основні функції програми, рис. 7.2 :

Функція пошуку опуклої оболонки
Функція пошуку вписаного прямокутника максимальної площі

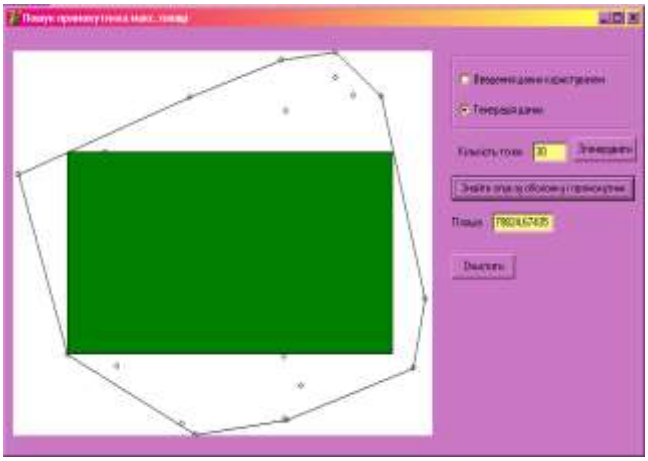


Рис.7.2

Ввід даних можна робити двома способами: вручну в поле, або згенерувати за допомогою програми. Вивід здійснюється на екран, при цьому значення площі записується в окреме поле.

Висновки

Запропоновано оптимальний алгоритм вписання прямокутника максимальної площі в опуклу оболонку на площині за оптимальний час O(nlogn). Прискорити час можливо лише на константу, сумістивши деякі кроки алгоритму. Інтерфейс виконано максимально просто, задля уникнення зайвих елементів і погіршення інтуїтивного розуміння програми. Більшість нюансів реалізації виникло на 2му етапі, оскільки на ньому розглядається чотири випадки розміщення прямокутника, для кожного з яких будується множина точок розбиття на інтервали. Оскільки можливо що 2 різні випадки розглядають один і той же прямокутник, можливо прискорити алгоритм, виключивши це, але знову ж це прискорення буде лише на константу.

7.2 Алгоритм на основі методу січних прямих

Перша частина алгоритму – це знаходження прямокутника найбільшої площі,сторони якого паралельні відповідним осям координат. Друга частина алгоритму полягає саме в знаходженні шуканого прямокутника. Тут використовується одна із властивостей опуклого многокутника, яка полягає у наступному.

Алгоритм (основна ідея)

Нехай знайдено оптимальний прямокутник для деякого напрямку α. (Прямокутник відповідає напрямку α , якщо одна з його сторін паралельна прямій, що містить промінь , який відповідає куту α на колі з центром в початку координат) Якщо зафіксувати напрямок $\alpha + \delta$, де $\delta \to 0$, то оптимальний для такого напрямку прямокутник отримується з початкового за допомогою зсуву вершин по ребрам многокутника в напрямку зміни кута.

Алгоритм (покроково)

Нехай маємо опуклу оболонку для множини *S* із *n* точок на площині CH(S) ={ P_1, P_2, …, P_n}, яка наприклад побудована методом Грехема. Тоді максимальний вписаний в CH(S) прямокутник знайдемо методом січних прямих. Січними прямими назвемо прямі, які паралельні осям OX і OY відповідно.

1. Оскільки у вписаного прямокутника принаймні три вершини належать сторонам многокутника то можливі два варіанти розташування прямокутника: точки верхньої сторони прямокутника належать сторонам многокутника або точки нижньої сторони прямокутника належать сторонам многокутника (варіанти не є взаємно виключаючими) .
2. Опишемо знаходження оптимального прямокутника першого типу (прямокутник другого типу шукаємо аналогічно). Зауважимо, що хоча б один з двох прямокутників обов'язково існує.
3. Січну пряму відносно OX починаємо рухати з найвищої точки многокутника. Січна пряма визначає чотири точки, рис. 7.3.

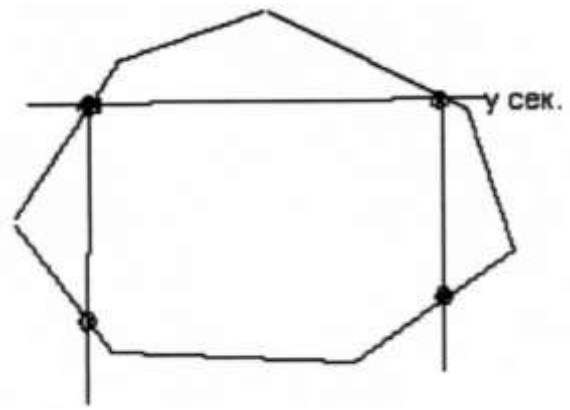


Рис. 7.3.

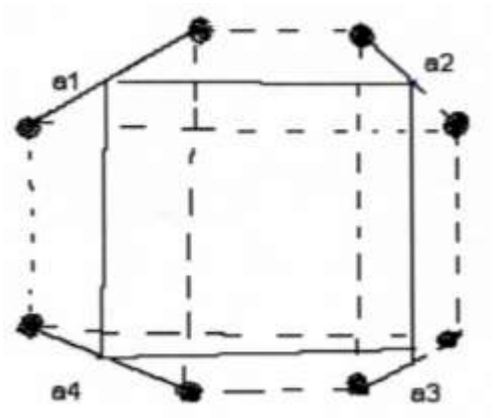


Рис. 7.4.

4. Розбиваємо рух січної прямої на етапи на кожному з яких жодна з чотирьох точок не змінює ребра. Пошук максимального прямокутника LR на кожному етапі являє собою мінімаксну задачу, розв’язок якої легко знаходиться за координатами відповідних ребер (можна легко отримати аналітичну формулу для знаходження оптимального положення січної прямої на етапі), а саме:
5. За відомими ребрами a_1, a_2, a_3, a_4 знаходимо максимальний прямокутник, рис. 7.4.
6. За результат беремо найбільший з побудованих на етапах прямокутників.
7. Зазначимо, що рух січної прямої продовжуємо до тих пір, доки по її положенню можна визначити згадані вище чотири точки (Іншими словами, доки обидва промені ,перпендикулярні до Ox і проведені через точки в яких січна пряма перетинає многокутник, перетинають деякі ребра нашого многокутника.)
8. Для знаходження найбільшого прямокутника січну пряму необхідно рухати знизу вгору.
9. Отже, після другого етапу ми маємо максимальний прямокутник P для напрямку $\alpha = 0$. Цей етап являє собою ітераційну процедуру. Опишемо її.

9.1. На нульовій ітерації $\alpha_0=\alpha_1=0$,оптимальним є прямокутник Р.
9.2. Нехай на останній ітерації ми знайшли оптимальний прямокутник для діапазону напрямків $[\alpha_0,\alpha_1]$. Нехай точка О центр цього прямокутника, а точки А, В, С — вершини прямокутника, що належать сторонам многокутника, рис. 7.5.

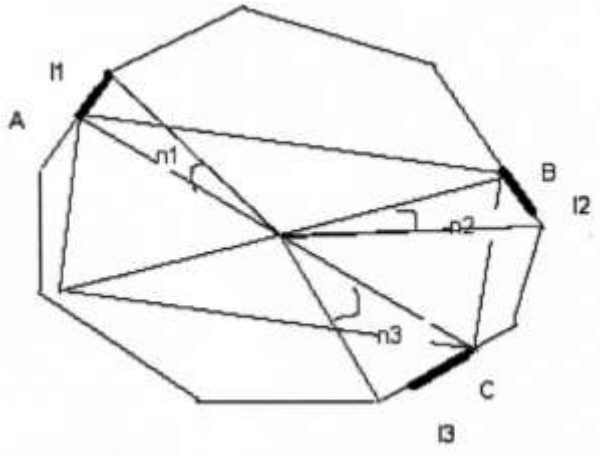


Рис. 7.5

9.3. Знаходимо для точок А, В та С мінімальні кути при повороті проти годинникової стрілки відносно точки О на які, вони перейдуть у вершину многокутника.
9.4. Мінімальний з трьох кутів позначимо як $\delta = \min(n_1, n_2, n_3)$.
9.5. Для наступної ітерації діапазон буде $[\alpha_l, \alpha_1+\delta]$. Знаходимо максимальний прямокутник для діапазону $[\alpha_l, \alpha_1+\delta]$.
9.6. Позначимо відрізки по яким рухаються точки А, В, С як 11, 12, 13, відповідно.
9.7. З властивості опуклого многокутника, яку було наведено вище випливає, що три вершини максимального прямокутника для даного напрямку будуть належати ребрам 11,12,13 відповідно.
9.8. Можна вивести формули, за координатами точок А,В,С та координатами відрізків 11,12,13 знаходиться максимальний прямокутник для поточного напрямку. Вигляд формул залежить від кутів нахилу відрізків.

9.9. Зауваження.

Може статися так що четверта вершина знайденого прямокутника для діапазону $[\alpha_0, \alpha_{II})$ виходить за межі многокутника. У цьому випадку максимальний прямокутник буде знайдено для напрямків від (α_0+180) до (α_1+180).
9.10. Для швидкої перевірки належності четвертої вершини многокутнику, після закінчення кожної ітерації зберігаємо ребро многокутника найближче до цієї вершини. На наступній ітерації найближче ребро або не зміниться , або перейде в наступне за годинниковою стрілкою.
9.11. Ітерації закінчуються ,коли якась з вершин зробить повний обхід, а значить переглянуті усі можливі напрямки від 0 до 360 градусів.
10. Результатом роботи третього етапу буде найбільший з всіх прямокутників які ми будували на ітераціях.

Оцінка складності алгоритму

Теорема 7.2. *Вписання прямокутника в n-вершинну опуклу оболонку методом січних прямих можна виконати за час O(nlogn).*

Доведення. На першому етапі ми виконуємо два рази алгоритм руху січної прямої. Кожен з кроків, на які ми розбиваємо рух січної прямої, обмежується положенням січної прямої при якому одна з чотирьох точок переходить в вершину многокутника. Отже, кількість кроків не може бути більшою за кількість точок , тобто їх не більше за *n*. На кожному кроці максимальний прямокутник знаходимо за формулами за константний час, тому неважко підрахувати, що складність алгоритму руху січної прямої є $O(n)$. Тому і весь перший етап має складність $O(n)$. На другому етапі для вершини прямокутника, яка не належить стороні многокутника знаходимо найближче ребро за час $O(n)$. Складність кожної ітерації є $O(1)$ складність переходу від ітерації до ітерації є також $O(1)$. Отже, нам залишилось оцінити найбільшу можливу кількість ітерацій на цьому етапі. Оскільки границя кожного етапу визначається ситуацією, коли одна з трьох вершин прямокутника суміщається з вершиною багатокутника, а ітерації закінчуються, коли деяка вершина зробить повний оберт , то з цього випливає ,що кількість етапів не більша за 3n. Тому складність другого етапу $O(n)$. Підсумовуючи складності всіх трьох етапів знаходимо, що складність всього алгоритму є $O(n\log n)$.

Алгоритм (реалізація)

Наведений алгоритм реалізовано в програмному продукті “Максимальний прямокутник”. Продукт розроблено в середовищі MS Visual C++ 6.0 з використанням мови C++. Програма будує на екрані опуклу оболонку та максимальний прямокутник вписаний в неї. Точки можна вводити мишкою та випадковим чином (максимальна кількість точок 30000). На рис. 7.6 показано приклад роботи програми.

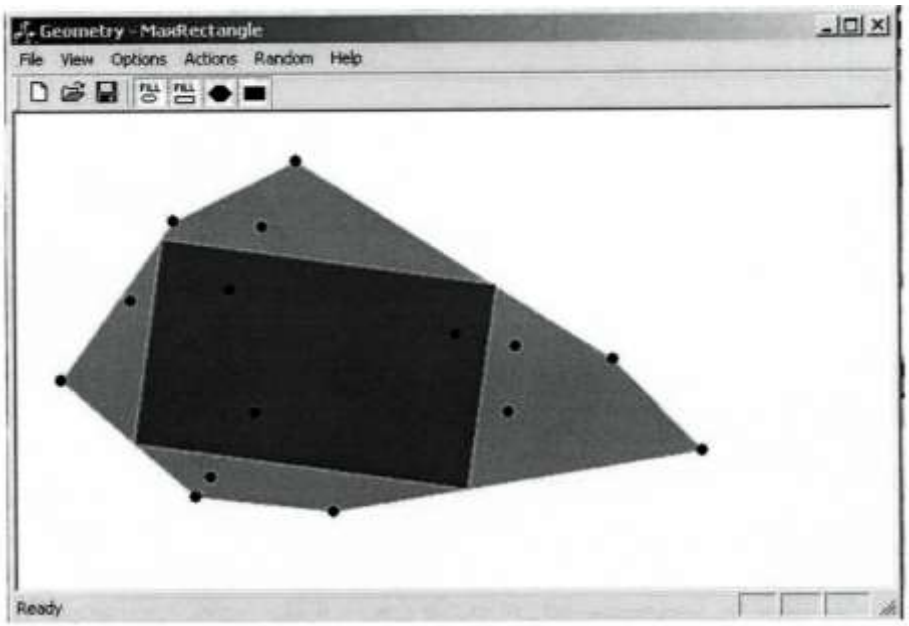


Рис.7.6.

7.3 Алгоритм на основі prune-and-search методу для фіксованих точок

Ми покажемо, випадок , коли найбільший прямокутник має дві, або три вершини на границі опуклої оболонки. Для розв'язання задачі можна скористатися алгоритмом Alt, Hsu та Snoeyink [38]. У цьому випадку задача зводиться до пошуку фіксованої точки композиції з двох або трьох функцій. Таким чином в основі алгоритму лежить ідея, що максимальна площа прямокутника визначається у фіксованій точці і ми можемо використати метод prune-and-search для нерухомої точки двох функцій або попередній (tentative prune-and-search) prune-and-search для нерухомих точок трьох функцій. Обидва методи мають дихотомію пошуку.

Алгоритм (основна ідея)

Розглянемо границю опуклої оболонки CH(S) як деякий многокутник P на площині. Розіб'ємо P на чотири ланцюги, розрізавши його в крайніх напрямках, як у напрямку *x*, так і *y*, як це показано на рис. 7.7.

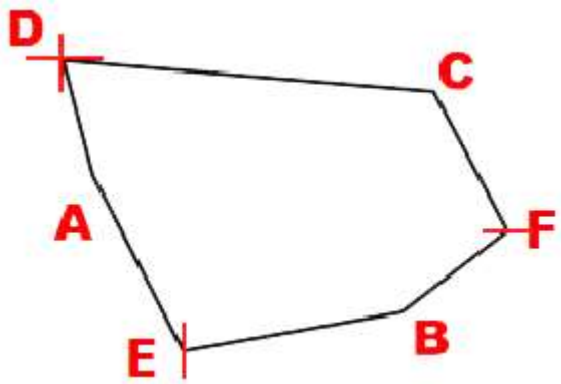


Рис. 7.7. Розділення на 4 ланцюги.

Визначимо обхід ланцюжків від A до D у напрямку проти годинникової стрілки, починаючи знизу ліворуч. Кожну вершину також потрібно розширити до нескінченно малого кола, щоб кожна можлива дотична до P у даній вершині мала унікальну точку дотику. Таким чином, коли ми будемо вилучати частину ланцюга з однієї сторони вершини, ми будемо також усувати деякі дотичні у цій вершині. Можливо, деякі ланцюжки спочатку складаються з однієї вершини. Тоді ми можемо негайно зробити висновок, що вони не будуть контактувати з кутом найбільшого прямокутника, оскільки як горизонтальні, так і вертикальні лінії, що проходять через вершину, лежать повністю поза P.

Випадок : 2 вершини на границі P.

Якщо найбільший прямокутник LR має рівно два кути на P, як на рис. 7.8, тоді ці кути будуть діагонально протилежними.

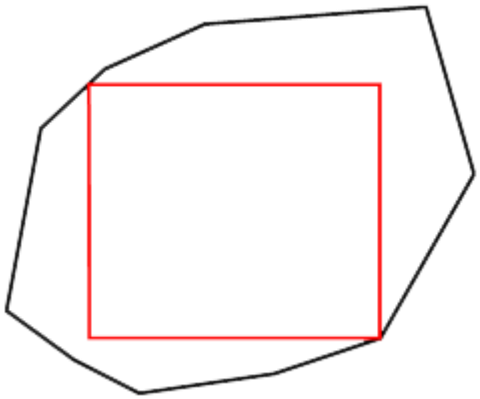


Рис. 7.8. Дві вершини прямокутника на границі многокутника.

Якби кути були суміжними, тоді протилежний кінець LR взагалі не торкався б границі P, і його можна було б відсунути, утворивши більший прямокутник. Подібним чином, якщо один кут LR знаходиться на границі P, два кінці LR можуть вільно рухатися і утворювати більший прямокутник. Діагонально протилежні кути LR повинні контактувати з двома вершинами P, або з однією вершиною та одним ребром. Згідно леми 7.1, якщо жоден з двох кутів LR не контактує з вершиною, тоді можна знайти більший прямокутник.

Лема 7.1 [44]: *Нехай e_1 і e_2 - не перетинаються відрізки ліній. Розглянемо набір порожніх паралельних осі прямокутників, які мають діагоналі протилежні кути на e_1 та e_2. У цьому наборі є прямокутник із найбільшою площею з принаймні одним кутом у кінцевій точці e_1 або e_2.*

Доведення: У роботі [44] стверджується, що функція площі з 2 параметрами може бути показана як поверхня сідла, і тому максимум лежить на границі. Більш інтуїтивно, якщо задовольняється припущення загального положення, два рядки, що містять відрізки, перетинатимуться з одного боку відрізків. Для будь-якого прямокутника з кутами, які ще не знаходяться на вершинах, можна буде зсунути його у зворотному напрямку, а потім розгорнути, оскільки один із кутів більше не буде контактувати з одним краєм многокутника. Це видно на рис. 7.9.

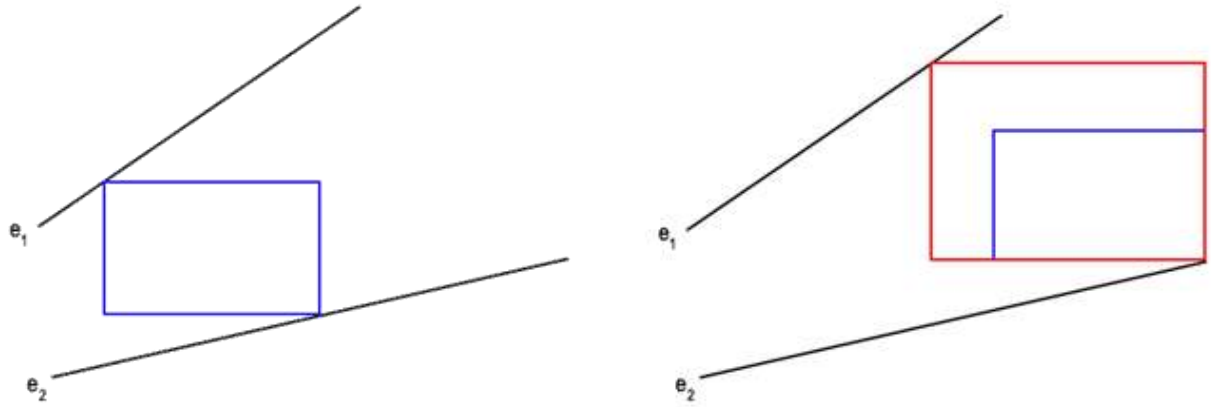


Рис.7.9. Двокутовий прямокутник, який не знаходиться на жодній вершині P, можна зробити більшим

Випадок : 3 вершини на границі P:

Найбільший за площею прямокутник LR може мати три вершини на границі P, як на рис. 7.10.

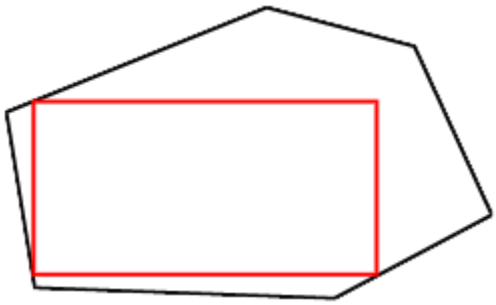


Рис. 7.10. Три вершини прямокутника LR лежать на границі P.

Якщо многокутник знаходиться в загальному положенні, максимум два кути LR будуть розташовані у вершинах P, оскільки сусідні кути не можуть знаходитися на вершині . Можливо, жоден з кутів не розташований на вершині. У всіх випадках, як тільки кожен ланцюг буде зведений до одного ребра або вершини найбільший прямокутник можна знайти за постійний час. Також можливий випадок, коли LR усі вершини його лежать на границі P. У цьому випадку процедура пошуку буде аналогічною як і в даному випадку.

Алгоритм (покроково)

Опишемо алгоритм, представлений в [38] для пошуку прямокутника найбільшої площі (LR). Якщо дві вершини LR лежать P, то вони будуть належати ланцюгу A і C або ланцюгу B і D. Ми отримаємо найбільший прямокутник знайшовши два максимуми. Робимо те ж саме для випадку з трьома вершинами. У цьому випадку існують чотири різні конфігурації: 1)A, B, C 2)A,B,D 3)A,C,D 4)B,C,D. Після того, коли ми знайшли всі можливі варіанти для двох і трьох кутів, їх максимум буде найбільшим ортогональним прямокутником.

Випадок: 2 вершини лежать на P:

1. **Лема 7.2.** *Припустимо, що прямокутник максимальної площі вписаний в P, має точно два кути на перетині з P, а саме a є A, c є C. Звідси слідує, що P має паралельні дотичні до a і c з нахилом m, де* $m = -m_{ac}m_{ac}$

Доведення. Якщо ми фіксуємо початок на *a* ми можемо отримати функцію постійної площі F через *c*, що можна записати як $F = xy$, де x та y – це відповідні координати *c*. Записавши $F = xy$, ми отримуємо m (нахилом до дотичних до c) беручи похідну $dy/dx = -F/x^2$. Нахил діагоналі від a до c m_{ac} є y/x або F/x^2 коли ми замінюємо y на F/x. Многокутник повинен мати дотичну до c з нахилом m, інакше ми можемо перемістити c на невелику відстань, щоб отримати прямокутник площею більше ніж F. Якщо многокутник має дотичну до c з нахилом m то перенесення c тільки

зменшить площу прямокутника. Такі ж результати ми отримаємо зафіксувавши *c* на початку і застосувавши функцію сталої площі через *a*. Поєднуючи ці результати ми отримуємо, що дотичні до *a* і *c* є паралельними. Рисунок 7.11. ілюструє доведення. В ньому $F = 6$, *c* розташоване на (3,2). Дотична – нахилом 2/3 належить, як функції площі так і с, нахил діагоналі складає 2/3.

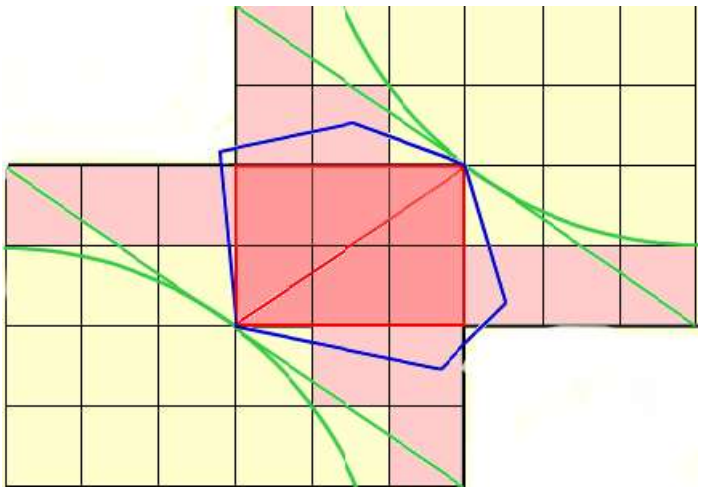


Рис. 7.11. Паралельні дотичні до многокутника і функція площі із нахилом $-m_{ac}$

2. З попередньої леми ми можемо визначити дві функції *f* та *g*, одна з яких буде зростаючою інша спадною, таким чином, щоб фіксована точка їх композиції забезпечила найбільший прямокутник.
3. Для точки *a є A*, нехай *f*(*a*) буде точкою на С, таким чином, що дотичні на *a і f(a),* будуть паралельними.
4. Для *a*, точка *c* є *C* є дотичною, яка має нахил *m*, нехай *g*(*c*) буде точкою на *A*. Таким чином, що лінія від *g*(*c*) до *c* має нахил *–m*.
5. При обході А і С проти годинникової стрілки *f* зростає, а *g* - спадає, так, що їхня композиція має фіксовану точку. В цій точці $a = g(f(a))$, так що $c = f(a)$ і $g(c) = a$. Це означає, що дотичні *a i c* паралельні, таким чином нахил діагоналі від'ємний по відношенню до нахилу дотичної до *c* . Ця умова задовольняє попередню лему і отже ми, таким чином, отримаємо прямокутник збільшеної площі

6. **Prune-and-search для знаходження фіксованої точки двох функцій**
Тепер, коли ми розмістили найбільший прямокутник у фіксованій точці поєднання двох функцій, необхідно описати яким чином знайдено цю фіксовану точку. Для цього розмістимо *a* і *c* на осях біля середини ланцюгів А і С та обираємо одне специфічне значення для дотичної з усіх можливих. Порівнюючи *f(a)* з *c* і *a* з *g(c)* ми можемо виключити половину А або С. Наприклад, якщо *f(a)* знаходиться після *c*,а *g(c)* перед *a*, то ми можемо виключити все після *a* , оскільки, ми знаємо, що фіксована точка не може бути розміщена на тій частині А. Функція f оцінена для будь-якої точки на виключеному ланцюзі завжди буде лежати поверх *c* на С. Оцінка *g* у цій новій точці буде завжди лежати нижче *g(c)* на *A* , яка сама знаходиться під *a*, тому ми ніколи не повернемось до нашої початкової точки. Усі можливі випадки, які

дозволяють нам виключити верхню або нижню частину A або C, представлені на рис.7.12.

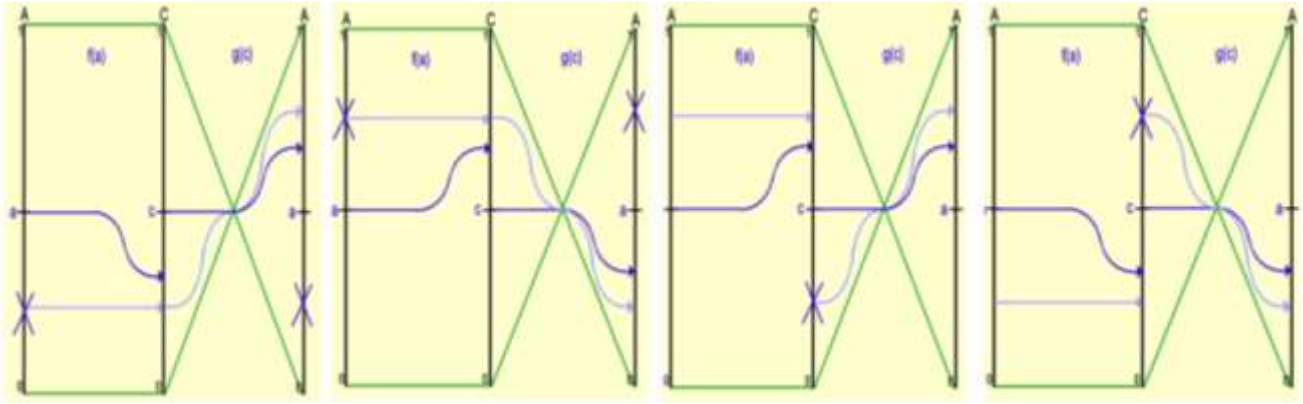


Рис.7.12. ***Prune-and-search*** для фіксованої точки композиції двох функцій

Цей процес може бути виконаний без оцінки *f(a)* або g(c). Замість цього, щоб порівняти *f(a)* і *c* нам потрібно порівняти дотичні до *a* і *c*, щоб порівняти g(c) і *a*, ми порівнюємо нахил до *c* з нахилом діагоналі з *a* до *c*. Коли частина ланцюга виключена ми вибираємо нову точку всередині частини, що залишилася. Після логарифмічної кількості кроків кожен ланцюг буде зведений до простого ребра чи вершини.

Як тільки обидва ланцюги звелися до ребра чи вершини, можна знайти найбільший прямокутник за допомогою простої алгебри:
1. Якщо A і C зведені до простої вершини, то найбільший прямокутник має діагональ, яка з'єднує дві вершини.
2. Якщо ребро залишилося на A і C ,то, згідно леми 7.1, то можна вважати що маємо чотири варіанти випадку з ребром і вершиною. У цьому випадку нахил дотичної - це нахил ребра і нам лише треба знайти лінію, яка проходить через вершину, що перетинає ребро. Це і буде діагональ найбільшого прямокутника.
3. Потім необхідно перевірити випадки, щоб пересвідчитись що вони вписані. Для цього необхідно перевірити, чи дві вершини що лишилися лежать у середині многокутника P. Це можна зробити за час $O(\log n)$, виконавши двійковий пошук по ланцюгах B і D (для нижнього і верхнього кутів, відповідно) щоб знайти відповідне ребро, яке має ту ж саму *x* або *y* координату.
4. Потім за константний час ми можемо перевірити чи знаходяться ці точки всередині.
5. Найбільший прямокутник з кутами на A і C є максимумом з усіх можливих прямокутників.
6. Тепер коли у нас є найбільший прямокутник з кутами на A і C ми можемо повторити ті ж самі кроки щоб знайти найбільший прямокутник з кутами на B і D.
7. Щоб знайти найбільший прямокутник в загальному випадку, нам необхідно також знайти найбільший прямокутник з 3 вершинами на P і взяти максимум. Рисунок 7.13 зображує весь процес знаходження прямокутника з двома кутами на A і C.

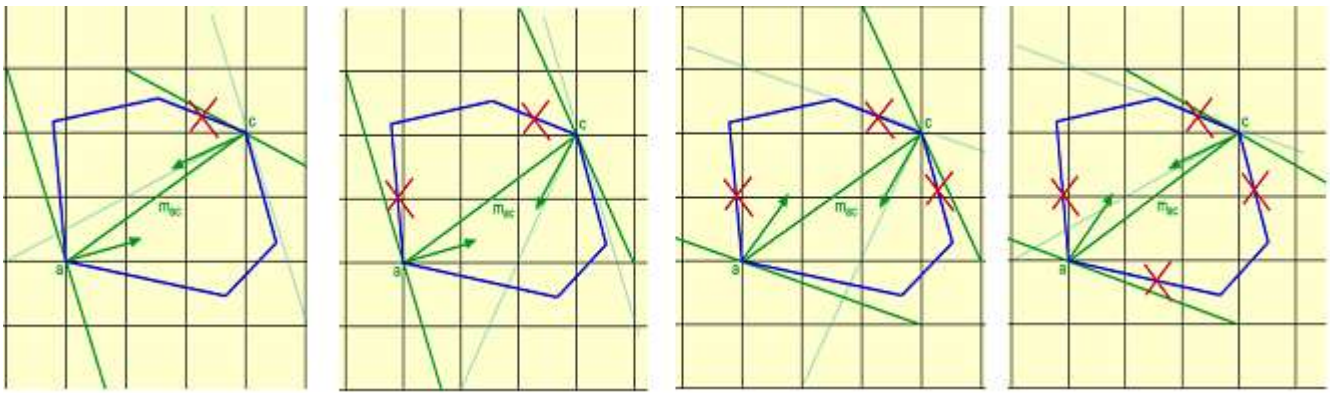


Рис. 7. 13.

Три кути на границі P:

Лема 7.3. *Припустимо, що максимальний прямокутник вписаний в P, і три його вершини лежать на границі P. Тоді існують дотичні* a *в A,* b *в B і* c *в C з нахилом* $m_a < 0$, $m_b > 0$, $m_c < 0$, *які відповідно задовольняють такі умови* $-m_a >= m >= -m_c$, *і* $m = -m_a * (m_c - m_b) / (m_a - m_b)$, *де* $m = m_{ac}$, *в тому випадку якщо найбільший прямокутник (LR) має максимальну площу.*

Доведення [38]. Для того щоб отримати рівняння для *m*, площа обчислюється як функція від *a*, *b* і *c*. Взявши похідну і прирівнявши її нулю отримаємо максимум і мінімум. Те ж саме можна сказати і тоді коли точки B і C співпадають. Якщо ми виключимо ці два випадки, то залишиться лише єдиний нуль, який відповідає максимуму. Зрештою це приводить до рівняння для *m*. З рис. 7.14 видно, що $-m_a$ більше за $-m_c$. Якщо це не так, то прямокутник може ковзати вздовж b, після чого буде розширений.

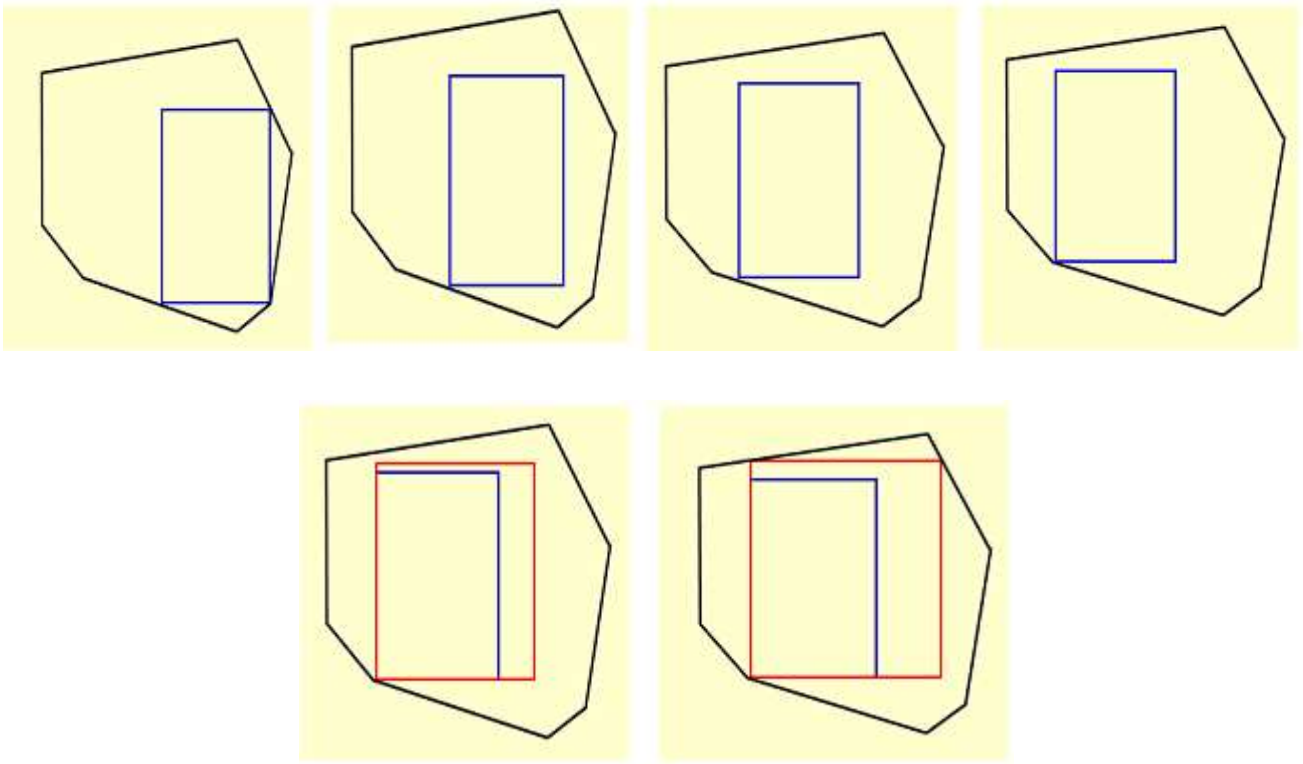


Рис. 7.14. Прямокутник може бути збільшений, якщо $-m_a < -m_c$

Попередній (tentative prune-and-search) prune-and-search для знаходження фіксованої точки трьох функцій.

1. Використовуючи лему 3, необхідно визначити три функції, щоб їх фіксована точка виявила найбільший прямокутник, три вершини якого лежать на P. Для цього необхідно, щоб усі три функції були спадними при обході P проти годинникової стрілки.
2. Очевидно, що ми можемо визначити $f(a)$ як точку на B з тією ж координатою y, що і на A.
3. Аналогічно, $g(b)$ - точку на C з такою ж координатою x, як b на B.
4. Нарешті, нехай $h(c)$ - точка на A така, що $m_{ac} = m$, як у лемі 3. Як і у випадку з двома вершинами, нам не потрібно безпосередньо визначати f, g або h. Ми отримаємо $f(a)$ і $g(b)$ виходячи з того що ми маємо y координату на a, і x координату на b.
5. Щоб оцінити $h(c)$, ми порівнюємо нахил лінії від a до c з m, яке знаходимо з відповідного рівняння (лема 3).

На рис. 7.15 показано, що у більшості випадків (шість з восьми), ми зможемо вилучити половину одного з трьох ланцюгів так, як ми робили у випадку з двома функціями.

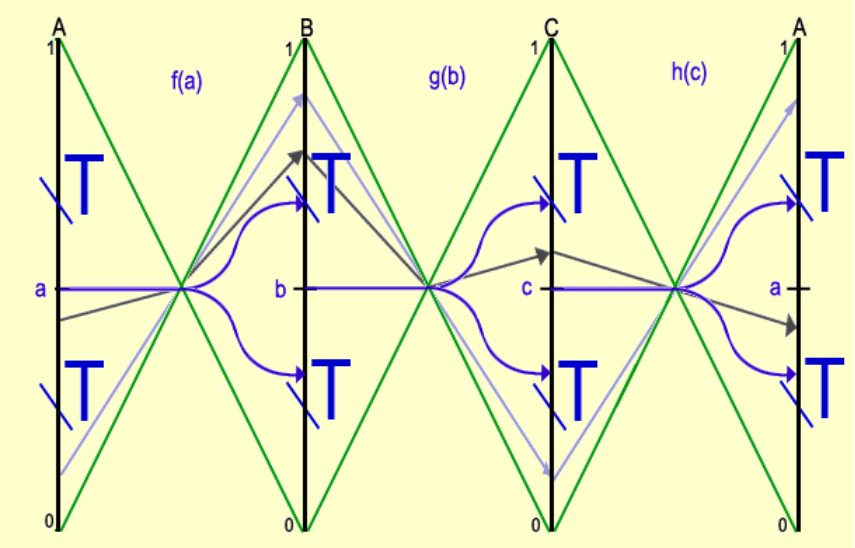


Рис. 7.15.

Тим не менш в інших випадках зображених на рис. 7.15 ми зможемо лише попередньо відкинути половину всіх ланцюгів. Коли ж попередніх вилучень не має, то одержимо випадок двох функцій. Коли ж є видалення, то ми входимо в попередній режим. В цьому режимі ми циклічно переміщуємо a, b і c. В результаті нічого не змінюється і ми продовжуємо попереднє відкидання, або ж ми можемо назавжди відкинути частину одного ланцюга. Коли це відбувається ми можемо скасувати інші попередні відкидання та повернутися до звичайного режиму.

Після того як кожен ланцюг був зведений до одного ребра чи вершини ми можемо знайти постійних кандидатів для найбільшого прямокутники за константний час.

1. По–перше може існувати рішення в якому не має вершин на P, у цьому випадку нам необхідно продовжити ребра, які містять *a, b, і c* до ліній, які будуть перетинатися і утворювати трикутник.
2. Якщо параметризувати область як функцію положення *b,* то отримаємо максимальну площу, коли b знаходиться точно посередині відповідного ребра трикутника.
3. Якщо b і a, c відповідно, розташовані одночасно на границі P, то це і буде шукане рішення.
4. Інші кандидати будуть містити принаймні одну вершину, тому ми можемо вивчити всі шість з них (по дві для кожного ребра в A, B і C) і зберегти лише вірні.
5. Для всіх вірних кандидатів ми повинні переконатися, що вони вписані точно так, як це було зроблено для двох вершин на P. Це дасть максимальний прямокутник з кутами на A, B і C.
6. Повторюючи кроки для трьох інших конфігурацій, ми отримаємо найбільший прямокутник з трьома вершинами на P.
7. Порівнюючи результат з випадком для двох вершин на P, одержимо остаточно найбільший прямокутник.
На рис. 7.16 показано приклад, коли найбільший прямокутник має вершини A, B і C на P.

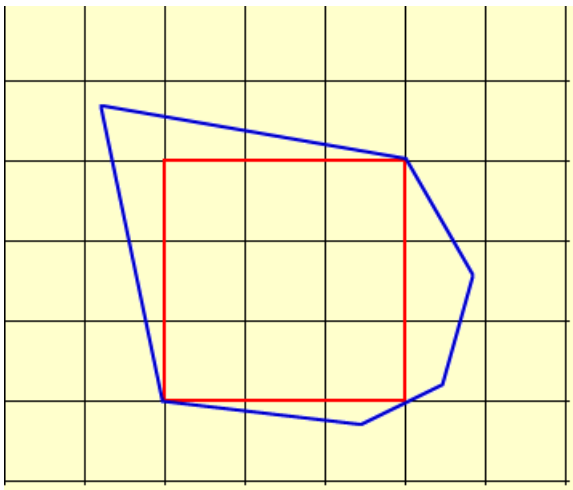


Рис.7.16. Приклад з трьома вершинами на P

Оцінка складності алгоритму

Теорема17. Вписання прямокутника в n-вершинну опуклу оболонку за допомогою prune-and-search методу для фіксованих точок можна виконати за час $O(n\log n)$.

Доведення. При пошуці прямокутника з двома вершинами A і C, які лежать на границі P і співпадають з його вершинами, ми виключаємо половину A або C на кожному кроці, щоб вони звелися до одного ребра або вершини за час $\log(A+C) < \log(n)$ [40]. Потім ми перевіряємо кандидатів на вписання за час $O(\log B) + O(\log D)$. Складність попереднього prune-and-search оцінена у [38] та [40] за допомогою потенційної функції, щоб показати, що ненульова постійна кількість вершин

відкидається на кожному попередньому кроці. В результаті досягається складність За час $O(\log n)$ знаходимо фіксовану точку трьох функцій. Перевірка приналежності виконується за час $O(\log D)$. Це повторюється чотири рази, щоб отримати найбільший прямокутник з трьома кутами. Як результат, загальна складність становить $O(\log n)$, а представлений вище алгоритм є найбільш ефективним із відомих алгоритмів пошуку найбільшого ортогонального прямокутника у опуклому многокутнику

Висновки

Алгоритм з [38], представлений вище, може бути використаний для вписання найбільшого прямокутника у опуклу n-вершинну оболонку за час $O(\log n)$. Алгоритм засновано на методі prune-and-search з фіксованими точками. Алгоритм є найбільш ефективним із відомих алгоритмів вписання найбільшого прямокутника у опуклу оболонку у найгіршому випадку.

8. Еліпс найбільшої площі вписаний в опуклу оболонку (BMEO (BO4))

Постановка Задачі. *Для заданої n – вершинної опуклої оболонки на площині вписати еліпс найбільшої площі.*

Методи розв'язання

Взагалі ідей та алгоритмів для розв'язання цієї задачі дуже мало. А тому ми розглянемо коротко ідеї можливих алгоритмів розв'язання, хоча не зовсім оптимальних.

8.1 Алгоритм бінарного пошуку вершин максимального трикутника

Існує гіпотеза, що еліпс однозначно задається 5 точками на границі. Для нашої задачі двоїстим поняттям буде те, що еліпс однозначно задається 5 прямими, що є дотичними до границі шуканого еліпсу. Скористаємося цією властивістю.
Нехай маємо опуклу оболонку для множини S із n точок на площині $CH(S) = \{ P_1, P_2, \ldots, P_n\}$, яка наприклад побудована методом Грехема.

Алгоритм (основна ідея)

1. На початку роботи алгоритму довільним чином вибираємо 5 ребер, щоб утворився початковий еліпс. Очевидно, що він буде вписаним, але не максимальним. Кожен наступний крок буде збільшувати площу нашого еліпса.
2. Обираємо деяке ребро зі всіх, що залишились, і яке не дотикається до еліпса.
3. Якщо ми можемо побудувати на основі цього ребра, і 4 основних більший еліпс – збільшуємо площу еліпса.
4. Кількість кроків для збільшення площі, очевидно, буде не більше n разів (за числом ребер в опуклій оболонці).

Оцінка складності алгоритму

Теорема 8.1. *Вписання еліпса максимальної площі в n-вершинну опуклу оболонку запропонованим алгоритмом можна виконати за час* $O(n^2)$.

Доведення. Сортування $O(n\log n)$. Вибір початкового еліпсу - за час $O(1)$. Ітерація збільшення площі вписаного еліпсу - за час $O(n)$ кожна. Отже, загальна складність $O(n^2)$.

Розглянемо ще одну із можливих ідей розв'язання задачі на вписання максимального еліпса в опуклу оболонку.

8.2 Алгоритм бінарного пошуку вершин максимального трикутника

Нехай маємо опуклу оболонку для множини S із n точок на площині $CH(S) = \{ P_1, P_2, \ldots, P_n\}$, яка побудована одним із відомих методів [5]. Позначимо її як многокутник P. Тоді має місце така ідея алгоритму.

Алгоритм (основна ідея)

1. Побудувати бісектриси всіх кутів многокутника P.
2. З поміж множини точок перетину бісектрис обираємо пару найбільш віддалених між собою точок A та B. Відрізок, що з'єднує ці дві точки визначає орієнтацію вписування.
3. Тепер треба знайти мінімальні розміри многокутника по орієнтації вписування. (діаметри вписаного еліпса не мають перевищувати їх). Для цього проведемо через знайдені точки A та B пряму. Той відрізок прямої, що лежатиме в середині многокутника буде більшим діаметром шуканого еліпса.
4. Перевірка на входження еліпса в многокутник. Для цього підійде властивість еліпса: якщо точка P належить еліпсі, то $\|F_1P\| + \|F_2P\| = 2a$, де F_1, F_2 - фокуси еліпса, а – більший радіус еліпса. Оцінка складності – $O(n\log n)$.

ЗАДАЧІ НА РОЗТАШУВАННЯ

9. Найбільше порожнє коло РМК (АО5)

Постановка Задачі. *На заданій множині S із n точок на площині побудувати коло найбільшого радіусу так, щоб внутрішня область оточена ним не містила б жодної точки множини S.*

Задачі такого роду можуть досить часто виникати в промисловості, коли об'єкт – шкідливе виробництво (наприклад, хімічний завод), атомна станція тощо – повинен бути розміщений якнайдалі від заданих об'єктів (населених пунктів, будинків тощо), щоб мінімізувати ефект від такого сусідства, або ж при створенні нового виробництва, якщо бажано уникнути конкуренції за вільний простір.
Також прикладом може бути задача вибору місця розташування магазину продажу тютюнових та алкогольних виробів, який не можна розміщувати, поблизу соціально побутових закладів (школи, дитсадки, лікарні, тощо). Також ця задача знайшла своє застосування у виробництві (задача розкрою): якщо область (прямокутник) - шматок тканини або листовий метал і точки - дефекти, необхідно оптимізувати витрати матеріалу. У військовій справі є задачі, які зводяться до задачі РМК, зокрема, для знаходження максимального радіусу дії радіолокаційних станцій на місцевості з врахуванням існуючих обмежень та особливостей рельєфу, які впливають на якість передач та прийому сигналу.

Методи розв'язання

Ранні алгоритми [45] вирішували цю задачу за $O(n^3)$ часу в гіршому випадку. Шеймос і Препарата [5] показали, що розв'язання може бути проведене за час $O(nlogn)$ з використанням діаграми Вороного [46], що й зумовило алгоритм розв'язання задачі,який являється найбільш оптимальним і простим в реалізації. Доведення цього факту описано у роботі [47].

9.1 Алгоритм пошуку найбільшого порожнього кола на основі Діаграми Вороного

Переформулювавши нашу задачу можна сказати так: треба знайти точку p_0 (як центр кола) на площині таку, що

$$\max_{p_0 \in Hull(s)} \min_i \left\{ (x_i - x_0)^2 + (y_i - y_0)^2 \right\}. \qquad (6)$$

Алгоритм (покроково)

Можливі два випадки розташування центра шуканого кола:

- Всередині опуклої оболонки. Нехай коло з таким центром "роздувається", поки не торкнеться якоїсь точки. Очевидно, радіус не буде максимальним, якщо

воно пройде тільки через одну точку (p_0 всередині комірки) або дві (p_0 на ребрі діаграми Вороного). Максимум можливий, якщо коло перетне три точки, тоді шукана p_0 – вершина діаграми Вороного (при цьому точки лежать більше ніж на одному півколі).

- На границі опуклої оболонки. При цьому шукана точка буде перетином ребра опуклої оболонки з ребром діаграми Вороного.

Таким чином, для розв'язання задачі потрібно здійснити такі кроки:

1. Побудувати діаграму Вороного.
2. Побудувати опуклу оболонку заданої множини точок.
3. Для кожної вершини діаграми Вороного перевірити, чи вона лежить всередині опуклої оболонки. Якщо так, то підрахувати радіус кола з центром в цій вершині, при потребі запам'ятати поточний результат як максимум.
4. Для кожного ребра діаграми Вороного підрахувати перетин його з опуклою оболонкою і знайти радіус кола з відповідним центром, при потребі запам'ятати поточний результат як максимум.
5. Повернути максимум у вигляді шуканого кола.

Оцінка складності алгоритму

Теорема19. *Пошук найбільшого порожнього кола на множині S із n точок на площині можна виконати методом на основі Діаграми Вороного за оптимальний час O(nlogn).*

Доведення. Як відомо, всі вершини діаграми Вороного можна отримати за час $O(n\ logn)$, таку ж складність може мати побудова опуклої оболонки (доведення в [5]). Нижче буде показано, що перетин ребер опуклої оболонки і ребер діаграми Вороного знаходиться за час $O(n)$.

Розглянемо спочатку два наступних факти.

I. Ребро діаграми Вороного перетинає не більше двох ребер опуклої оболонки (оскільки оболонка опукла, то перетин з нею будь-якої прямої є відрізком, можливо і порожнім).

II. Кожне ребро опуклої оболонки перетинає хоча б одне ребро діаграми Вороного (бо кожне ребро опуклої оболонки з'єднує дві різні точки із заданої множини, які належать двом різним многокутникам Вороного).

Уявимо, що кожен необмежений многокутник Вороного закритий відрізком на нескінченно віддаленій прямій, що лежить в площині. Нехай ребра e_i опуклої оболонки перелічені в порядку обходу проти стрілки годинника, рис.9.1. Згідно другого з наведених вище фактів, завжди існує точка u перетину довільного ребра e_j з ребром діаграми Вороного r.

Визначивши орієнтацію r в зовнішню відносно опуклої оболонки область, позначимо через $V(i)$ багатокутник діаграми Вороного, що знаходиться по лівий бік від r. Рухаючись з точки u по границі цього многокутника проти годинникової стрілки, потрапимо в деяку точку w перетину ребра опуклої оболонки з ребром діаграми. Така точка може належати або ребру e_j, або наступному e_{j+1}. Отже, кожне ребро $V(i)$ перевіряється на перетин з двома ребрами оболонки, що робиться за

константний час. Потім обхід повторюється для точки *w* і так далі, поки не повернемось у вихідну точку *u*.

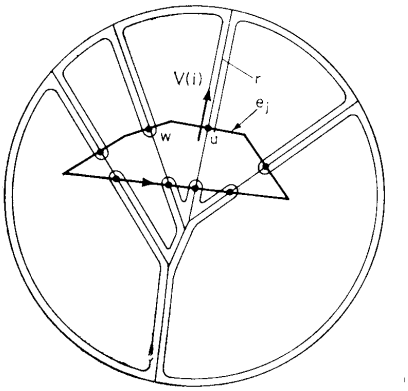


Рис. 9.1.

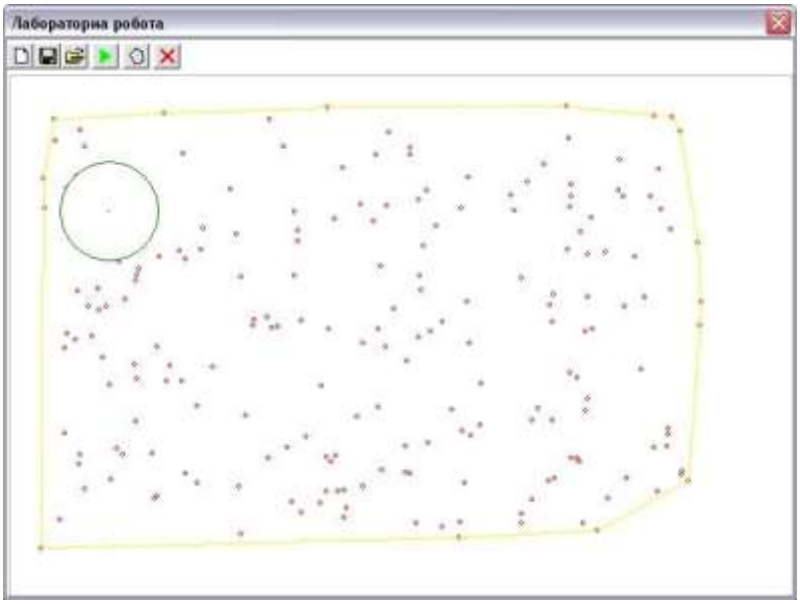


Рис.9. 2.

Таким чином, кожен відвідуваний відрізок діаграми Вороного проходиться не більше двох разів, а оскільки кожне ребро оболонки має не більше двох таких відрізків, то кожне ребро перевіряється в результаті максимум чотири рази, звідки отримуємо оцінку *O(n)*. Звідси очевидно отримується загальна оцінка часу виконання всього алгоритму *O(nlogn)*.

На рис. 9.2 подано результат програмної реалізації розв’язання задачі пошуку найбільшого порожнього кола на множині із 200 точок мовою java.

Висновки

У цьому розділі описано оптимальний алгоритм пошуку кола найбільшого радіусу на множині точок на площині на основі Діаграми Вороного з оцінкою складності оцінка часу *O(nlogn)*. Вдосконалення можуть бути пов’язані лише з певними особливостями програмної реалізації, але все це стосуватиметься вже конкретних випадків з конкретними завданнями, вимогами і обмеженнями. Слід сказати, що використання діаграми Вороного робить більш зручною реалізацію, і, крім того, це чудово вписується в сучасну концепцію повторного використання програмного забезпечення.

10. Найбільший порожній прямокутник РМП (АО6)

Постановка Задачі. *На заданій обмеженій (прямокутником чи опуклою оболонкою) множині S із n точок на площині побудувати прямокутник найбільшої площі, який би не містив всередині жодної точки множини S.*

Розглянемо задачу знаходження найбільшого порожнього прямокутника, яка часто виникає, наприклад, коли потрібно знайти найкраще розташування для магазину, чи стадіону, також ця задача виникає при роботі з базами даних. Для визначеності вхідної множини задачі будемо вважати, що задана множина обмежена деяким прямокутником, рис. 10.3.

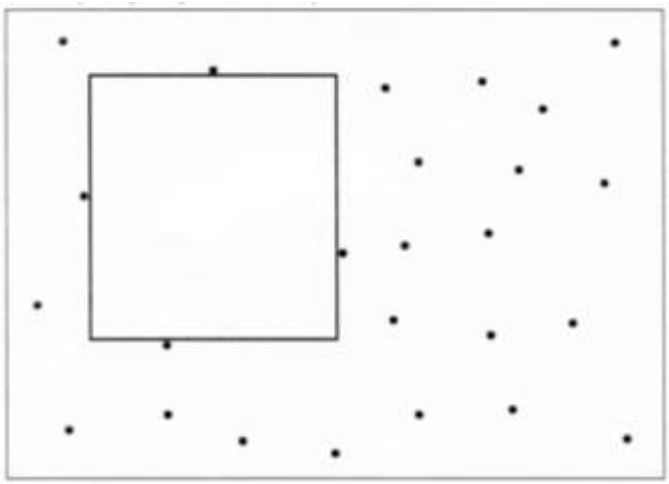
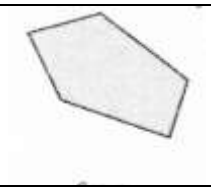
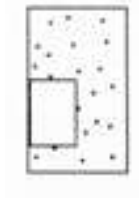


Рис. 10.3

Методи розв'язання

Незважаючи на свою практичну важливість, лише для простих областей (Convex, Constrained Staircase) були знайдені алгоритми з оптимальною оцінкою [48, 49, 50,51], таб. 10.1. Для більш складних та зокрема для нашої задачі РМП(найбільший порожній прямокутник) це виявилось не простим завданням. Так сьогодні можна виділити два найбільш перспективних шляхи її вирішення. Перший - це метод «розділяй та володарюй» [52] та другий метод «перебору всіх дотичних прямокутників». Chazelle, Drysdale, Lee [52] в 1986 році знайшли алгоритм для об'єднання для методу «розділяй та володарюй» з часовою складністю $O(\log n)$ і як результат отримали алгоритм для РМП з часовою складністю $O(n\log^3 n)$. Aggarwal A., Suri S. [53] покращили теоретичну оцінку до $O(\log n)$ та $O(n\log^2 n)$ відповідно, але зауважили, що константа виходить дуже великою, тому для реалізації краще використовувати попередній алгоритм.

	Форма області		Оцінка складності	автори
1	Найбільший опуклий многокутник		θ(n)	Amenta [49] (convex programing)
2	Найбільший ортогональний сходинковий многокутник 1		θ(n)	Aggarwal and Wein [48]
3	Найбільший ортогональний сходинковий многокутник 2		O(nlog^{2}n) O(na(n))	Maccena, O'Rourk, Suri [50] using LAECR
4	Найбільший ортогональний ізотеричний многокутник		O(nlog^{2}n)	Maccena, O'Rourk, Suri [50] using LAECR
5	Найбільший порожній прямокутник		O(nlog^{2}n)	Chazele, Drysdale, Lee [52] Aggarwal and Suri [53]

Таб. 10.1

10.1 Алгоритм Орловського пошуку найбільшого порожнього прямокутника

Для розв'язання задачі про РМП скористаємось методом «перебору всіх дотичних прямокутників», який був вперше запропонований Орловським [54]. Введемо поняття дотичного прямокутника.

Означення 10. 1. ***Дотичним прямокутником*** *називається прямокутник, який кожною стороною дотикається до точки або до сторони оточуючого прямокутника, проте не повністю міститься в оточуючому прямокутнику і не містить жодної точки всередині.*

Лема 10.1. *Розв'язком задачі про РМП буде дотичний прямокутник.*

Доведення. Припустимо, від противного: ми маємо розв'язком задачі РМП, який не є дотичним прямокутником, тоді існує хоча б одна сторона, в яку він може бути розширений, що не можливо. Отже, ми прийшли до протилежного.

Лема 10. 2. *Для n точок існує щонайбільше n^2 дотичних прямокутників.*

Доведення. Можна побачити, що дотичний прямокутник повністю визначається протилежними сторонами. Кожна сторона фіксується або точкою або стороною оточуючого прямокутника, отже таких прямокутників може бути щонайбільше n^2.

Отже, для того щоб вирішити задачу НПП необхідно перебрати всі можливі дотичні Прямокутники, як це вперше показав Орловський [54] за допомогою двох відсортованих списків: по x та y , відповідно. У цьому випадку можна скористатися інший підходом - використанням структури даних під назвою «висотне дерево». Дамо означення висотного дерева.

Означення 10. 2. *Висотним деревом називається геометрична структура даних, яка зберігає точки на площині та має наступні властивості:*
1) бінарне дерево.
2) лівий син знаходиться нижче (менша y координата) та лівіше (менша к координата).
3) правий син знаходиться не вище (y координата не більша) та не лівіше (x координата не менше).

Лема 10. 3. *Для будь якого набору точок висотне дерево може бути побудовано за O(nlog n).*

Доведення. Для доведення побудуємо алгоритм побудови висотного дерева:
1. Відсортувати точки по парі (x, y).
2. Будуємо висотне дерево, на кожному кроці додаючи по одній точці з відсортованого списку (час $O(\log n)$) до дерева. Для цього будемо використовувати «дерево відрізків».

Означення 10. 3. ***«Ліва вправо» гілка*** *називається список вершин, який складається з лівого сина та правого сина лівого сина, та правого сина правого сина лівого сина і так далі аж до кінця.*

Означення 10.4. ***«Права вліво» гілка*** *називається список вершин, який складається з правого сина та лівого сина правого сина, та лівого сина лівого сина правого сина і так далі аж до кінця.*

Лема 10. 4. *Всі дотичні прямокутники які верхньою стороною дотикаються до кореня висотного дерева іншими сторонами дотикаються або до сторін оточуючого прямокутника або до точок гілок «ліва вправо» та «право вліво».*

Доведення. Звернувши увагу на означення гілок висотного дерева можна побачити, що між лівою та правою гілками відсутні точки. Так як дотичні прямокутники не містять жодної точки всередині, то всі вони будуть формуватись лише точками гілок та сторін оточуючого прямокутника.
Корінь висотного дерева може бути видалений за лінійний час відносно суми точок гілок «ліва вправо» та «права вліво» за допомогою «зшивання віток». Отже, ми

можемо розглянути всі дотичні прямокутники, що зверху дотикаються до кореня висотного дерева, потім видалити корінь висотного дерева та повторити процес. В результаті, ми розглянемо всі дотичні прямокутники, окрім тих, що верхньою стороною спираються до верхньої сторони оточуючого прямокутника. Їх ми розглянемо окремо.

Алгоритм (покроково)

1. Розглянемо дотичні прямокутники, що дотикаються верхньою стороною до верхньої сторони оточуючого прямокутника.
2. Побудувати висотне дерево.
3. Перевірити дотичні прямокутники, що дотикаються верхньою стороною до кореня висотного дерева.
4. Видалити корінь висотного дерева.
5. Якщо дерево містить хоча б одну точку, перейти до пункту 3.

Оцінка складності алгоритму

Теорема20. *Пошук найбільшого порожнього прямокутника на множині S із n точок на площині можна виконати алгоритмом Орловського за час $O(n^2)$ у найгіршому та O(nlogn) у середньому випадках.*

Доведення. Пункт 1алгоритму, можна виконати за $O(n)$ при умові що на вхід надходять відсортовані по x точки. Це не впливає на час виконання всього алгоритму. Пункт 2 можна виконати (як було показано) за $O(n\log n)$. Інші пункти алгоритму перебирають дотичні прямокутники, витрачаючи на кожний константну кількість часу. Отже, запропонований алгоритм буде працювати за час рівний $O(n\log n+s)$, де s — кількість дотичних прямокутників, і як було показано в лемі 10.2, дотичних прямокутників буде щонайбільше n^2 в гіршому випадку, тобто оцінка складності алгоритму у найгіршому випадку буде $O(n^2)$ та $O(n\log n)$ в середньому. Алгоритм також потребує $O(n)$ пам'яті.

Висновки

Описано алгоритм, який дозволяє за час $O(n^2)$ в гіршому та $O(n\log n)$ у середньому випадку розв'язати задачу пошуку прямокутника максимальної площі на множині S із n точок на площині. Часова оцінка алгоритму в гіршому випарку, використовуючи метод «перебору всіх дотичних прямокутників» не може бути покращена, так як на розгляд одного дотичного прямокутника витрачається константна кількість часу.

10.2 Алгоритм на основі Діаграми Вороного

У розділі 9.1 описано ідею оптимального алгоритму пошуку найбільшого порожнього кола на множині S із n точок на площині за допомогою Діагарми Вороного. І тому постає логічне запитання, чи можна застувати цей же підхід до

пошуку найбільшого порожнього прямокутника щоб отримати оптимальний алгоритм. У цьому розділі описану ідею подібного алгоритму.

Алгоритм (ідея)

Розглянемо спочатку задачу для випадку, коли наша множина S складється із чотирьох точок. Очевидно, що для існування розв'язку необхідно, щоб ці чотири точки утворювали опуклий чотирикутник (надалі припустимо, що ми завжди маємо четвірку точок, що утворюють опуклий чотирикутник). Можна довести, що для прямокутника максимальної площі, сторони якого проходять через вершини даного опуклого чотирикутника, виконується наступне твердження [55].

Твердження 1. *Одна з пар протилежних сторін прямокутника перпендикулярна до діагоналі опуклого чотирикутника, яка визначається точками, через які проходять сторони прямокутника. Спираючись на цю властивість ми і побудуємо наш алгоритм.*

Як було показано в [55] якщо вписати прямокутник максимальної площі в 4 точки, то одна пара сторін цього прямокутника буде паралельна одній з діагоналей опуклого многокутника, утвореного з цих точок.

Перевіримо дію цього твердження для множини S із n точок. Візьмемо довільні дві точки p_1 та p_2 і розглянемо відрізок p_1p_2. Ми можемо вважати його діагоналлю деякого опуклого чотирикутника. Отож, побудуємо прямі l_1 та l_2, що проходять відповідно через точки p_1 та p_2 і є перпендикулярними до прямої p_1p_2. Далі знайдемо всередині смуги, яка визначається прямими l_1 та l_2, точки p_3 та p_4, які є найближчими до прямої p_1p_2 з різних сторін (для визначеності, можемо вважати, що p_3 є найближчою точкою "зліва" від p_1p_2, а p_4 - "справа"). Тепер, якщо ми проведемо через точки p_3 та p_4 прямі l_3 та l_4, які перпендикулярні до l_1 та l_2, то ми отримаємо в результаті, що чотири прямі l_1, l_2, l_3, l_4 визначають прямокутник, який є можливим розв'язком задачі. Слід зауважити, що точки p_3 та p_4 не завжди існують, але в тому випадку, коли одна (чи обидві) з цих точок не існують, пара точок p_1 та p_2 не може утворювати діагональ, що перпендикулярна до сторін найбільшого порожнього прямокутника.

Згідно цього підходу, необхідно перебрати всі пари точок і для кожної пари знайти кандидата на максимальний порожній прямокутник. Нажаль, такий підхід дає нам складність $O(n^3)$. Тому нам потрібно певним чином оптимізувати перебір пар точок. Тут нам на допомогу приходить діаграма Вороного. Припустимо ми побудували для нашої множини діаграму Вороного. Розглянемо точки p_1 та p_2, та відповідні їм многокутники Вороного V_1 та V_2. Можна показати [56], що точки p_3 та p_4, які ми визначили вище, будуть точками, що відповідають многокутникам Вороного V_3 та V_4, які є суміжними одночасно до V_1 та V_2. Більш того, Чазелле [57] показав, що якщо ми вже знаємо одну з точок (наприклад p_1), через яку проходить ребро максимального порожнього многокутника, то три інші точки повинні бути серед точок, що відповідають тим многокутникам Вороного, які є суміжними до V_1, а також тим, які є суміжними до цих, перших суміжних многокутників.

Алгоритм (покроково)

1. Будуємо діаграму Вороного. ЇЇ можна представити реберним списком з подвійними зв'язками (РСПЗ).
2. Перебираємо всі многокутники Вороного, і для точки, що відповідає кожному такому многокутнику, знаходимо інші три, які можуть разом визначати кандидата для максимального порожнього многокутника. Для цього нам треба лише перебрати всі суміжні (до другого порядку) многокутники, користуючись при цьому процедурою, описаною на початку. Подібно до того, як ми вже згадували, такі три точки не завжди існують. Якщо вони не існують, то ми просто переходимо до наступного многокутника Вороного, інакше порівнюємо площу новознайденого прямокутника, з попередньою максимальною площею. Якщо вона більша, то ми запам'ятовуємо новознайдений прямокутник, як максимальний порожній прямокутник. Далі переходимо до наступного многокутника Вороного.
3. Перебираємо всі пари суміжних точок і відповідно пари точок, які суміжні одночасно з двома цими точками. Якщо таких (одночасно сумісних точок) немає просто переходимо до наступної пари.
4. Знаходимо прямокутник вписаний у ці 4 точки, сторона якого паралельна одній із діагоналей многокутника описаного навколо цих чотирьох точок.
5. Порівнюємо площу S цього прямокутника із максимальною площею S_{max} прямокутників, отриманих на попередніх кроках. Якщо $S > S_{max}$ то покладаємо $S_{max} = S$ і запам'ятовуємо даних прямокутник.
6. Беремо наступну пару суміжних точок і повторюємо з кроку 4.

На рис. 10.7 показано фрагмент алгоритм пошуку максимального прямокутника.

```
 function MaxRect(set P)
   Smax = 0
   max = (0, 0, 0, 0)
   DiagrVoron = Voronoi(P)
   for indx V □ DiagrVoron do
      Rect = Rect4Point(V, DiagrVoron)
      if define(Rect) and (S(Rect) > Smax) then
         Smax = S(Rect)
         max = Rect
      endif
   enddo
   return Smax, max
end
```

Рис. 10.7

Оцінка складності алгоритму

Теорема 21. *Пошук найбільшого порожнього прямокутника на множині S із n точок на площині можна виконати алгоритмом на основі діаграми Вороного за час $O(n^2 \log n)$ у найгіршому.*

Доведення. Діаграму Вороного можна побудувати за час $O(n \log n)$.
Оскільки кожен многокутник Вороного може містити не більше ніж O(1) ребер, то для перегляду кожної групи точок необхідних лінійний час - $O(n)$.
Звідси отримуємо загальну складність алгоритму - $O(n^2 \log n)$.
На рис. 10.8. представлено приклад роботи програмної реалізації задачі.

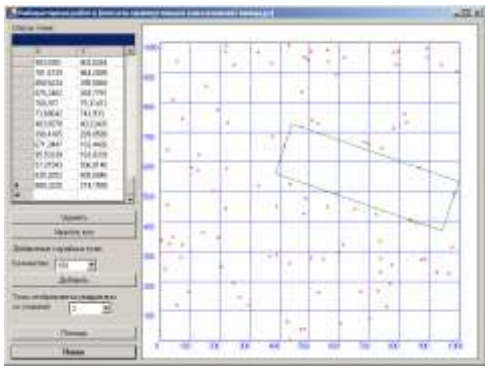


Рис. 10.8

Висновки

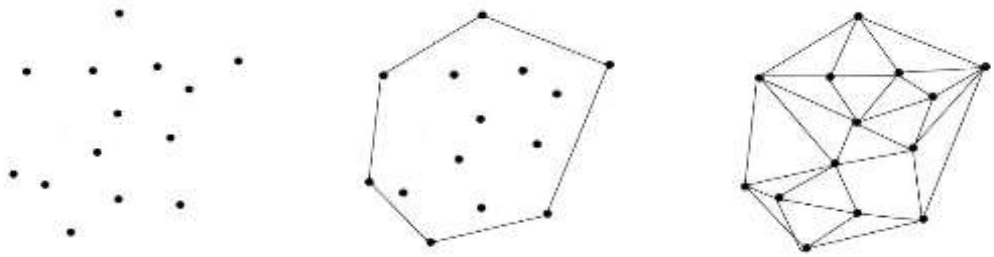
Представлений алгоритм розв'язує задачу знаходження найбільшого порожнього прямокутника для скінченної множини точок на площині. Проте більш популярною (і корисною) є задача про максимальний порожній прямокутник в дещо іншій постановці : на площині задана множина S з n точок, що міститься всередині деякого заданого прямокутника R. Знайти прямокутник максимальної площі, що міститься всередині R і сторони якого паралельні сторонам R, при цьому він не повинен містити всередині точок з множини S. Для цієї задачі також є алгоритм зі складністю $O(n \log n)$ [53] . Наш варіант задачі про найбільший порожній прямокутник розглядався також в [55], проте звичайно основний результат був отриманий в [57]. Також слід зауважити, що наш алгоритм ілюструє одне з багатьох застосувань діаграми Вороного.

10.3 Алгоритм з використанням триангуляції

Алгоритм (покроково)

1. Побудувати опуклу оболонку даної множини точок за допомогою алгоритму обходу Грехема,рис. 10.5 а, б.

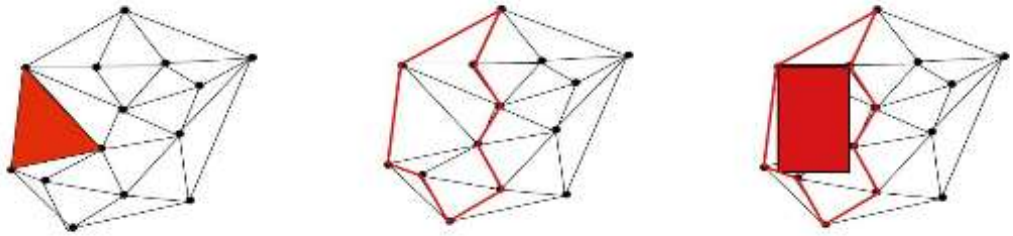
2. Триангулювати множину точок всередині опуклої оболонки, рис. 10.5 в.



а) б) в)

Рис. 10.5

3. За один прохід по тріангуляції знайти трикутник найбільшої площі, рис. 10.6 а.



а) б) в)

Рис. 10.6

4. Розширити трикутник таким чином, щоб утворилася максимально можлива множина, в основі якої лежить трикутник, без точок всередині, рис. 10.6 б.

5. З допомогою оптимізаційної функції, з можливих прямокутників вибрати прямокутник найбільшої площі, рис. 10.6 в.

Оцінка складності алгоритму

Теорема21. *Пошук найбільшого порожнього прямокутника на множині S із n точок на площині можна виконати алгоритмом з використанням тріангуляції за час O(nlogn) у найгіршому випадку.*

Доведення.

1. Опукла оболонка n точок на площині може бути знайдена за час O($n$log$n$).
2. Тріангуляцію можемо зробити за час O($n$log$n$).
3. За час O(n) можемо знайти трикутник найбільшої площі, множину навколо трикутника найбільшої площі та вписання прямокутника..

Література

1. D. E. Knuth. Big omicron and big omega and big theta. \ SIGAST News, 8(2). – P. 18 24.- 1976.

2. Терещенко В.М. Принцип зводимості в задачах обчислювальної геометрії // Вісник київського університету, серія фізико-математичні науки, випуск № 2, 2009, С. 149-154.

3. *Cook S.A.* The complexity of theorem-proving procedures, Proceedings Third Ann. ACM Symposium on the Theory of Computing, Shaker Heights, N.-Y. , 1971, 151-158.

4. *Карп Р. М.* Сводимость комбинаторных проблем. Кибернетический сборник, вып. 12.-М.: Мир, 1975, с. 16-38.

5. F. Preparata and M.I. Shamos. Computational Geometry: An introduction. Springer-Verlag, Berlin, 1985, -475 p.

6. *Гэри М., Джонсон Д.* Обчислювальні машини і трудно решаемые задачі. М.: Світ, 1982.

7. Peter Bürgisser: Completeness and Reduction in Algebraic Complexity Theory, Springer, 2000.

8. E.R. Griffor: Handbook of Computability Theory, North Holland, 1999, ISBN 978-0-444-89882-1.

9. Dave Mount. Data Structures.\ Lecture Notes. 2001. P. 123.
http://www.cs.umd.edu/users/mount/420/Lects/420lects.pdf

10. M.T. Goodrich, R. Tamassia, and D. Mount, Data Structures and Algorithms in C++, Second Edition, John Wiley and Sons, Inc., 2011.

11. M.T. Goodrich, R. Tamassia, and M. Goldwasser, Data Structures and Algorithms in Python, John Wiley and Sons, Inc., 2013.

12. M.T. Goodrich, R. Tamassia, and M. Goldwasser, Data Structures and Algorithms in Java, Sixth Edition, John Wiley and Sons, Inc., 2014.

13. Sylvester, J. J. *(1857), "A question in the geometry of situation",* Quarterly Journal of Mathematics, *1: 79.*

14. Shamos, M. I.*; Hoey, D. (1975), "Closest point problems",* Proceedings of 16th Annual IEEE Symposium on Foundations of Computer Science, *pp. 151–162,* doi:10.1109/SFCS.1975.8

15. Welzl, Emo *(1991), "Smallest enclosing disks (balls and ellipsoids)", in Maurer, H. (ed.), New Results and New Trends in Computer Science, Lecture Notes in Computer Science, 555, Springer-Verlag, pp. 359–370,* CiteSeerX 10.1.1.46.1450, doi:10.1007/BFb0038202, ISBN 978-3-540-54869-0.

16. Megiddo, Nimrod *(1983), "Linear-time algorithms for linear programming in* R3 *and related problems",* SIAM Journal on Computing, *12 (4): 759–776,* doi:10.1137/0212052, MR 0721011.

17. Elzinga, J.; Hearn, D. W. (1972), *"The minimum covering sphere problem",* Management Science, *19: 96–104,* doi:10.1287/mnsc.19.1.96

18. J. G. Hocking, G. S. Young. Topology, Addison-Wesley, Reading, MA. -1961.

19. I. M. Yaglom, V. G. Boltyanski. Convex Figures, Holt, Rinehart and Winston,New York, 1961.

20. Ильясова, Н.Ю. Измерение биомеханических характеристик сосудов для ранней диагностики сосудистой патологии глазного дна / Н.Ю. Ильясова, А.В. Куприянов, М.А. Ананьин // Компьютерная оптика. -2005. - № 27. - С. 165-170.

21. N. Moshtagh.Minimum volume enclosing ellipsoid. GRASP Lab, University of Pennsylvania, 2005 http://www.seas.upenn.edu/nima/papers/Mim ˜ vol ellipse.pdf.

22. Б.В. Рубльов Вісник Київського університету. Серія: фіз.-мат. науки. - 1999. - 1. - с. 223-228

23. http://articles.org.ru/cfaq/index.php?qid=653&frommostrecent=yes

24. http://www.springerlink.com/index/M8X71U7884X572R7.pdf

25. http://www.cgal.org/

26. Акопян А.В., Заславский А.А. Геометрические свойства кривых второго порядка. -2-е изд., дополн. — М.: МЦНМО, 2011. — 152 с.

27 . Ф.Препарата, М.Шеймос. Обчислювальна геометрія: Вступ. - М.: Мир, 1989.

28. Gernot Hoffmann. Ellipse Through Four Points.\http://docs-hoffmann.de/ellipse08032004.pdf

29. А.В. Анісімов, В. М. Терещенко, І. В. Кравченко. Основні алгоритми обчислювальної геометрії.\ Навчально-методичний посібник. – ВПЦ «Київський університет». – 2002.- С. 82.

30. Л.Ф.Тот «Расположение на плоскости, на сфере и в пространстве» Гос Изд Физ-Мат Лит М. 1958г., 383с

31. http://e-maxx.ru/algo/inscribed_circle_ternary

32. http://e-maxx.ru/algo/inscribed_circle

33. Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars. Computational Geometry:Algorithms and Applications.Berlin Heidelberg: Springer-Verlag,- 2nd, revised edition.- 2008. – 386 p.

34. http://mathcentral.uregina.ca/QQ/database/QQ.09.06/h/matthew2.html

35 Т.Кормен, Ч. Лейзерсон, Р. Ривест «Алгоритмы. Построение и Анализ»

36 Ахо Х., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов.

37 Роджерс Д. Алгоритмические основы машинной графики.

38 H. Alt, D. Hsu, and J. Snoeyink. Computing the largest inscribed isothetic rectangle. In Proc. 7th Canadian Conf. Comput. Geom., Universit'e Laval, Qu'ebec, August 1995, pp. 67--72.

39. N. Amenta. Bounded boxes, Hausdorff distance, and a new proof of an interesting Helly-type theorem. \\ Proceedings of the 10th Annual ACM Symposium on Computational Geometry (1994) pages 340-347.

40 D. Kirkpatrick and J. Snoeyink, Tentative prune-and-search for computing fixed-points with applications to geometric computation, Fundamental Informatic, 22 (1995), 353—37

41. P. Fischer and K. U. Hoffgen. Computing a maximum axis-aligned rectangle in a convex polygon. In Information Processing Letters, 51, pages 189-194, 1994.

42. http://cgm.cs.mcgill.ca/~athens/cs507/Projects/2003/DanielSud.

43. www.williamspublishing.com/PDF/5-8459-0772-1/part.pdf

44. K. Daniels, V. Milenkovic, and D. Roth. Finding the largest area axis-parallel rectangle in a polygon. Computational Geometry: Theory and Applications, 7:125--148, 1997.

45. B. Dasarathy and L.J. White. On some maximum location and classifer problems, Computer Science , Washington , D.C., (1975).

46. S. J. Fortune, A sweep line algorithm for Voronoi diagrams, Algorithmica 2 (1987), 153-174.

47. U. Manber and M. Tompa. Propabilistic, nondeterministic and alternating decision trees, Tech. Rep. N. 82-03-01 , Univ, Washington, 1982.

48. A. Aggarwal and J. Wein. Computational Geometry Lecture Notes for MIT 18.409. 1988.

49. N. Amenta. Largest Volume Box is Convex Programming. Personal communication, 1992.

50. M. McKenna, J. O'Rourke, and S. Suri. Finding the largest rectangle in an orthogonal polygon. In Proceedings of the 23rd Al lerton Conference on Communication, Control, and Computing, pages 486{ 495, 1985

51. D. Wood and C. K. Yap. The Orthogonal Convex Skull Problem. Discrete and Computational Geometry, 3:349{365, 1988.

52. B. Chazelle, R. L. Drysdale III, and D. T. Lee. Computing the largest empty rectangle. SIAM Journal on Computing, 15:300{315, 1986.

53. A. Aggarwal and S. Suri. Fast Algorithms for Computing the Largest Empty Rectangle. In Proceedings of the 3rd Annual ACM Symposium on Computational Geometry, pages 278 { 290, 1987.

54. M. Orlowski. A new algorithm for the largest empty rectangle problem. Algorithmica, 5:67-73. 1990.

55. Jeet Chaudhuri, Subhas C. Nandy. Largest Empty Rectangle among a Point Set. LNCS 1738, p. 34

56. D.T. Lee, D. Dobkin. Voronoi diagram : revisiting. Discrete and Computational Geometry, 1993, vol. 4 No. 3, pp. 123-136

57. B. Chaselle, D.T. Lee. Largest empty rectangle and Voronoi diagram, IJCGA Vol 7, No 2 pp. 112-129

58. H. Radamacher and O. Toeplitz. The Enjoyment of Mathamathics, Princenton University Press, Princeton, , № 3, 1957.

59. C. Toregas, R. Swain, C. Revelle and L. Bergman. The location of emergency service facilities \ Operations Research, 19. –P 1363 -1373. – 1971.