МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

В. М. Терещенко

Аналіз методів
розв'язання оптимізаційних задач обчислювальної геометрії

Навчальний посібник

Видавничо-поліграфічний центр
"Київський університет"
Версія не для друку

КИЇВСЬКИЙ УНІВЕРСИТЕТ 2022

УДК 514.7:004(075.8)
Т35

Рецензенти:
д-р фіз.-мат. наук, проф. Д. Я. Хусаінов,
канд. фіз.-мат. наук, доц. Л. Л. Омельчук,
канд. фіз.-мат. наук, асист. Д. О. Іваненко
(Київський національний університет імені Тараса Шевченка)

Рекомендовано до друку вченою радою
факультету комп'ютерних наук та кібернетики
(протокол № 8 від 21 листопада 2020 року)

Ухвалено науково-методичною радою
Київського національного університету імені Тараса Шевченка
(протокол № 4-21 від 22 квітня 2021 року)

Терещенко В. М.

Т35 Аналіз методів розв'язання оптимізаційних задач обчислювальної геометрії : навч. посіб. / В. М. Терещенко. – К. : ВПЦ "Київський університет". – 2022. – 112 с.

Розглянуто структури даних і геометричні конструкції, а також основні класи оптимізаційних задач обчислювальної геометрії з описання, вписування та розташування геометричних об'єктів у евклідовому просторі E^2. Наведено деякі ефективні підходи, методи й алгоритмічні інструменти для їх розв'язання, для більшості задач подано приклади реалізації алгоритмів.

Для студентів, які вивчають дисципліну "Обчислювальна геометрія та комп'ютерна графіка", а також навчаються за освітніми програмами, пов'язаними з інформаційними та комп'ютерними технологіями.

УДК 514.7:004(075.8)

© Терещенко В. М., 2022
© Київський національний університет імені Тараса Шевченка, 2022
ВПЦ "Київський університет», 2022

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

ЗМІСТ

ВСТУП **5**
1. АЛГОРИТИМІЧНІ ІНСТРУМЕНТИ **9**
1.1. Аналіз задачі 10
1.2. Установлення межі ефективності розв'язання задачі 11
1.3. Визначення інструментів та стратегії розв'язання 14
1.4. Створення та використання структур даних 16
1.5. Типи структур даних 16
2. ЗАДАЧІ НА ОХОПЛЕННЯ **26**
2.1. Побудова найменшого охоплюючого кола навколо множини точок (ОМК) 26
2.1.1. Алгоритм із використанням діаграми Вороного 27
2.1.2. Алгоритм побудови базового трикутника 30
2.1.3 Алгоритм на основі пошуку трьох базових точок 32
2.2. Трикутник найменшої площі (ОМТ) 34
2.3. Побудова прямокутника найменшої площі, що містить задану множину точок (ОМП) 37
2.4. Побудова найменшого охоплюючого еліпса (ОМЕ) 46
2.4.1. Алгоритм побудови з використанням опуклої оболонки (варіант 1) 46
2.4.2. Алгоритм побудови з використанням опуклої оболонки (варіант 2) 51
2.4.3. Алгоритм побудови з використанням опуклої оболонки (варіант 3) 53
3. ЗАДАЧІ НА ВПИСУВАННЯ **55**
3.1. Побудова кола найбільшого радіуса, вписаного в опуклу оболонку (ВМКО) 55
3.1.1. Алгоритм на основі методу стискання сторін 55
3.1.2. Алгоритм на основі подвійного тернарного пошуку 59
3.2. Трикутник найбільшої площі, вписаний в опуклу оболонку (ВМТО (АО31)) 63
3.2.1. Алгоритм простого перебирання 63
3.2.2. Алгоритм бінарного пошуку вершин максимального трикутника 66

Видавничо-поліграфічний центр "Київський університет" Розсіяне для другу

3.3. Прямокутник найбільшої площі,
вписаний в опуклу оболонку (ВМПО (АО41)) 70
3.3.1. Алгоритм бінарного пошуку
вершин максимального трикутника 71
3.3.2. Алгоритм на основі методу січних прямих 73
3.3.3. Алгоритм на основі методу
prune-and-search для фіксованих точок 79
3.4. Еліпс найбільшої площі, вписаний
в опуклу оболонку (ВМЕО (ВО4)) 90
3.4.1. Алгоритм бінарного пошуку
вершин максимального трикутника (варіант 1) 90
3.4.2. Алгоритм бінарного пошуку
вершин максимального трикутника (варіант 2) 91
4. ЗАДАЧІ НА РОЗТАШУВАННЯ 92
4.1. Найбільше порожнє коло РМК (АО5) 92
4.2. Найбільший порожній прямокутник РМП (АО6) 96
4.2.1. Алгоритм пошуку найбільшого
порожнього прямокутника Орловського 98
4.2.2. Алгоритм на основі діаграми Вороного 100
4.2.3. Алгоритм з використанням тріангуляції 105
ЛІТЕРАТУРА 107

Видавничо-поліграфічний центр
"Київський університет"
Версія не для друку

ВСТУП

Інженери при розробленні сучасних інформаційних і комп'ютерних технологій досить часто стикаються з технічними чи технологічними проблемами, які так чи інакше зводяться до задач комбінаторної геометрії (computational geometry). Задачі обчислювальної геометрії зводяться до пошуку оптимальних розв'язків так, як вони визначені на великих множинах даних, елементи яких є складними геометричними об'єктами (точки, відрізки, полігони, багатогранники, розбиття простору тощо). Такий пошук не завжди приносить бажані результати, оскільки потрібен не просто розв'язок геометричної задачі, а щоб він задовольняв оптимізаційні вимоги, такі як час виконання та обсяг використаної пам'яті. Іншими словами, на відміну від класичної математики, необхідно знайти не лише розв'язок задачі загалом, а саме найефективніший розв'язок. У цьому полягає складність розв'язання задач обчислювальної геометрії. Тому ідеї, підходи, методи обчислювальної геометрії досить актуальні для фахівців у сфері інформаційно-комп'ютерних технологій.

У пропонованому посібнику викладено деякі ефективні підходи й алгоритми розв'язання для одного із практично важливих класів задач обчислювальної геометрії – задач оптимізація. Ці задачі пов'язані з оптимальним вписуванням і описанням різних фігур (коло, багатокутник, еліпс тощо). Список задач цього класу та їхніх позначень наведено в таблиці:

Список задач оптимізації

№	Позначення	Назва та постановка задачі	Оцінки складності
		A	
Задача лінійного програмування			
1	AO1	На заданій множині точок розв'язати задачу лінійного програмування	$O(n)$

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

Продовження табл.

№	Позначення	Назва та постановка задачі	Оцінки складності
Задачі на охоплення			
2	ОМК (АО2)	*Найменше коло* На заданій множині S із n точок на площині побудувати коло найменшого радіуса, яке б охоплювало S	$O(n^2)$
3	ОМТ (АО3)	*Трикутник найменшої площі* На заданій множині S із n точок на площині 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 вписування			
6	ВМКО (АО21)	*Найбільше коло, вписане в опуклу оболонку* На заданій множині S із n точок на площині побудувати опуклу оболонку і вписати в неї коло найбільшого радіуса	$O(n^2)$
7	ВМТО (АО31)	*Трикутник найбільшої площі, вписаний в опуклу оболонку* 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, вписаний в опуклу оболонку* На заданій множині S із n точок на площині побудувати опуклу оболонку і вписати в неї прямокутник найбільшої площі	$O(n^2)$

Видавничо-поліграфічний центр "Київський університет"
Вороніна Е.П. Друку не було

Продовження табл.

№	Позначення	Назва та постановка задачі	Оцінки складності
9	ВМЕО (ВО4)	*Найбільший еліпс, вписаний в опуклу оболонку* На заданій множині S із n точок на площині побудувати опуклу оболонку і вписати в неї еліпс максимальної площі	$O(n^2)$
10	ВМКЗ (ВО21)	*Найбільше коло, вписане в зірковий багатокутник* У заданий зірковий n-кутник вписати коло найбільшого радіуса	
11	ВМЕЗ (ВО21е)	*Найбільший еліпс, вписаний у зірковий багатокутник* У заданий зірковий n-кутник вписати еліпс найбільшої площі	
12	ВМПЗ (АО41п)	*Прямокутник найбільшої площі, вписаний у зірковий багатокутник* У 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* У задане 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 радіуса, яке б не містило всередині жодної точки множини S	$O(n^2)$

Видавничо-поліграфічний центр "Київський університет"
Вороніна Л.П. Пруцу

Закінчення табл.

№	Позначення	Назва та постановка задачі	Оцінки складності
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 усередині 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 усередині жодної точки множини S	$O(n^2)$

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

Видавничо-поліграфічний центр "Київський університет"
Версія не для друку

1. АЛГОРИТМІЧНІ ІНСТРУМЕНТИ

Спочатку розглянемо підходи й основні алгоритмічні інструменти (стратегії, структури даних та інтегральні геометричні структури), які дозволяють розробляти ефективні алгоритми розв'язання задач обчислювальної геометрії, зокрема й задач оптимізації, а передусім задач, результати яких застосовуються в інженерних чи наукових проєктах.

Як показує досвід співпраці з відомими IT-компаніями, при розробці алгоритмів найбільш поширений такий підхід: якщо немає обмежень за часом, то розв'язуємо напряму або за допомогою перебірного алгоритму; якщо є обмеження за часом, то шукаємо серед популярних методів розв'язання типових задач; якщо задача нетипова, то шукаємо ідеї розв'язання схожих задач за різними джерелами інформації (журнальні статті, конференції, інтернет).

Інколи постановку та розв'язання задачі можна знайти в літературі, але не завжди. У будь-якому випадку це буде не постановка потрібної нам задачі, а щось схоже. Окрім цього, інколи ідею розв'язання задачі можна знайти в інформаційних джерелах, але в більшості випадків деталі та проблемні моменти реалізації розробники приховують. Тому досить часто такий підхід розчаровує.

Інший підхід пов'язаний із застосуванням наявних узагальнених алгоритмічних стратегій, проте це теж не завжди дозволяє адаптувати ефективний алгоритм для розв'язання поставленої задачі.

Ми пропонуємо підхід, який дозволяє розробляти ефективні прямі чи узагальнені алгоритми і базується на:

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

2) установленні межі ефективності розв'язання задачі (нижні, верхні й оптимальні оцінки складності);

3) визначенні основних властивостей вхідних і вихідних даних задачі;

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

- розробленні прямого алгоритму розв'язання;

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

- застосуванні відомих узагальнених методів і підходів до розв'язання задачі;

5) створенні нових чи використанні наявних 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 1.1 (локалізації точки)*. Нехай задано n-вершинне розбиття G(V, E) евклідового простору E^d (у випадку двовимірного простору – планарний граф) і точку Z. Визначити область розбиття, яка містить точку Z* (*локалізувати точку Z на заданому розбитті*) (рис. 1.1)*.*

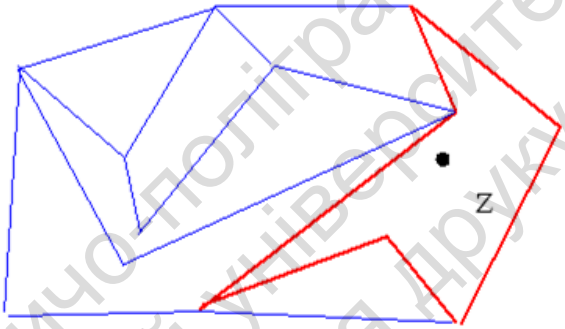


Рис. 1.1. Локалізація точки Z на планарному розбитті

1.1. АНАЛІЗ ЗАДАЧІ

1. З'ясуємо передусім клас задач, до якого належить наша задача. Це клас геометричного пошуку (ГП), підклас – задачі локалізації точки.

2. Аналіз входу: *n*-вершинний планарний граф (задається у вигляді *G*(*V*, *E*)) і точка *Z*. На виході має бути багатокутник (у вигляді списку вершин чи ребер), який містить *Z*.

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

3. Аналіз розв'язання задачі у випадку звичайного перебору. Підрахуємо кількість багатокутників (буде порядку $O(n)$) і кількість сторін багатокутника (буде $O(n)$).

4. Процедура перебору: $\forall$ i-го n-кутника перевірити належність Z. Загальний час пошуку становитиме порядку $O(n \times n)$ у найгіршому випадку.

Чи можна зменшити час розв'язання задачі? Щоб відповісти на це запитання, необхідно встановити межі ефективності розв'язання задачі, тобто теоретично найбільшу швидкість її розв'язання (або класу задач) для заданого розміру. Для відповіді на ці запитання скористаємося таким підходом.

1.2. УСТАНОВЛЕННЯ МЕЖІ ЕФЕКТИВНОСТІ РОЗВ'ЯЗАННЯ ЗАДАЧІ

Існує система оцінок складності алгоритмів розв'язання задач розмірністю N, запропонована Кнутом [1]. Згідно із системою Кнута ефективність задачі можна визначити за допомогою верхніх, нижніх і 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:

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

(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 передусім установити нижню оцінку складності так, як вона вказує найменшу можливу швидкість розв'язання задачі. Знаючи нижню оцінку складності, ми намагатимемось знайти або розробити стратегію, яка дозволить побудувати алгоритм, оцінка складності (верхня чи оптимальна) якого буде близькою до цієї

Видавничо-поліграфічний центр "Київський університет". Версія не для друку

оцінки складності в найгіршому випадку. У багатьох випадках установити нижню оцінку складності не просто, проте існує метод звідності задач [2–8], який дозволяє встановлювати нижні та верхні оцінки складності для деяких із них.

Метод перетворення (звідності) задач

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

*1) вхідні дані задачі **A** перетворюються на вхідні дані задачі **B** із часом $r_{in}(n)$;*

*2) задача **B** має розв'язок;*

*3) розв'язок задачі **B** перетворюється на правильний розв'язок задачі **A** із 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 це

$$\boldsymbol{A} \propto_{r(n)} \boldsymbol{B}.$$

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

Теорема 1.1 (*нижні оцінки методом перетворення*). *Якщо відомо, що задача **A** вимагає $T(n)$ часу і задача **A** $\propto_{r(n)}$ **B*** (***A** перетворена на **B**), то **B** можна розв'язати за час не менше $T(n) - O(r(n))$.*

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

Припустимо, що 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. Далі, застосовуючи наслідок 1.1, досить легко встановлюємо нижню оцінку складності задачі Z.

Теорема 1.2 (*верхні оцінки методом перетворення*). *Якщо задачу **B** можна розв'язати за час $T(n)$ і задача **A** $\propto_{r(n)}$ (**A***

Видавничо-поліграфічний центр "Київський університет" Вересневе видання друку

перетворена на ***B****), то* ***A*** *можна розв'язати за час, який не перевищує* $T(n)+O(r(n))$.

Наслідок 1.2. *Якщо задачу* ***B*** *можна розв'язати за час* $T(n)$ *і задача* $\boldsymbol{A} \propto_{r(n)} \boldsymbol{B}$*, то всі 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 вся множина алгоритмів задачі B згідно з теоремою 1.2 та наслідком 1.2 є підмножиною множини алгоритмів задачі Z, серед яких є алгоритм бажаної ефективності.

Множина класів звідності задач обчислювальної геометрії достатньо велика, формально її можна позначити як

$$KOG_Z = \left\{ K^{OG}_{f_1(N)}, K^{OG}_{f_2(N)}, ..., K^{OG}_{f_s(N)} \right\}.$$

Наведемо список задач, які входять до класів звідності порядку $K^{OG}_{\Omega(N \log N)}$.

1. Перевірка унікальності елементів (чи є на заданій множині два рівні елементи).
2. Перевірка рівності включення множин.
3. Перевірка перетинання множин.
4. Перевірка на значущість: задано множину із S точок на площині. Чи всі точки належать опуклій оболонці множини точок?
5. Перевірка ε-близькості: задано множину із S елементів. Чи є серед елементів цієї множини два, які лежать на відстані ε?
6. Об'єднання інтервалів.
7. Задача сортування: задано множину із S елементів. Необхідно їх відсортувати.

Отже, повертаючись до нашої задачі, бачимо, що 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 дорівнює* $\Omega(\log n)$.

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

Доведення наведено в монографії [5]. Нам необхідно розробити стратегію, яка б дозволяла будувати алгоритми локалізації точки на планарному розбитті $G(V, E)$ з оцінками складності $O(\log n)$ та $\theta(\log n)$.

1.3. ВИЗНАЧЕННЯ ІНСТРУМЕНТІВ ТА СТРАТЕГІЇ РОЗВ'ЯЗАННЯ

Метою стратегії розв'язання є досягнення оцінки складності $O(\log n)$ та $\theta(\log n)$.

Ми маємо $O(n)$ багатокутників і кількість сторін багатокутника в найгіршому випадку. Тому повний перебір дає $O(n^2)$. Нам необхідно знайти рішення, яке дозволить будувати алгоритми локалізації за час $O(\log n)$, тобто зменшити час перевірки належності багатокутника до $O(1)$. Це можна здійснити, якщо замість N-кутників ми візьмемо k-кутники (підрозбиття на k-кутники, напр. трикутники), при цьому $k << N$. Далі зобразимо k-кутники у вигляді структур даних, наприклад дерева пошуку. Тоді рішення полягає в такому:

1. Розбити на області задане планарне розбиття $G(V, E)$ так, щоб перевірка належності багатокутника давала $O(1)$. Варіанти можуть бути такими: розбиття на трикутники, смуги, чотирикутники та монотонні ланцюги (рис. 1.2–1.4).

2. Одержане підрозбиття подати у вигляді бінарного дерева пошуку.

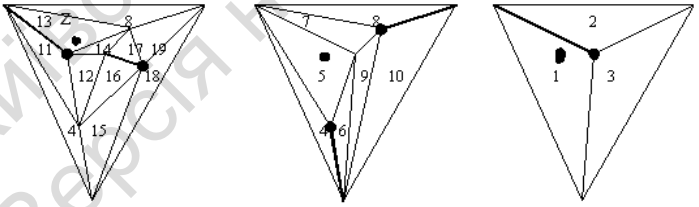


Рис. 1.2.(а) Розбиття на трикутники. Пошук $O(\log n)$

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

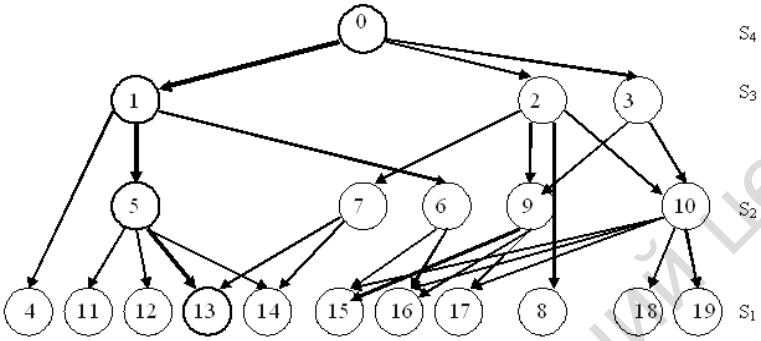


Рис. 1.2.(б) Розбиття на трикутники. Пошук $O(\log n)$

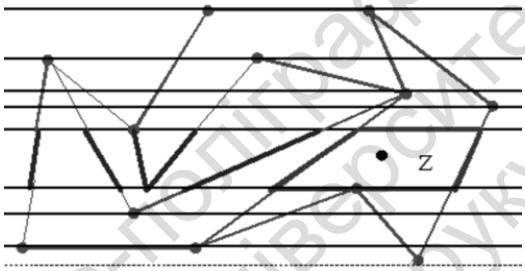


Рис. 1.3. Розбиття на смуги. Пошук $O(\log n)$

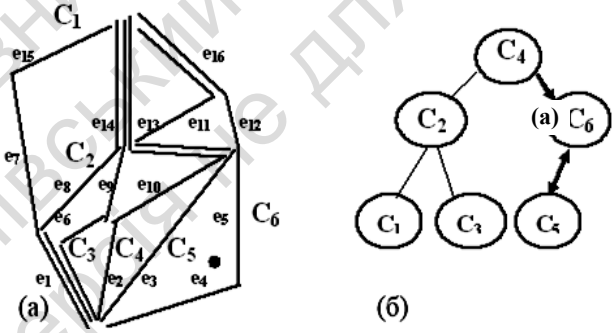


Рис. 1.4. Розбиття на ланцюги. Пошук $O(\log^2 n)$

Видавничо-поліграфічний центр "Київський університет". Верстання для друку

1.4. СТВОРЕННЯ ТА ВИКОРИСТАННЯ СТРУКТУР ДАНИХ

Вхідні дані в обчислювальній геометрії мають свою специфіку і зазвичай є множинами геометричних об'єктів складної структури (точок, ребер, багатокутників, багатогранників, об'єктів обертання), планарним розбиттям тощо. Для організації ефективної роботи алгоритмів над такими множинами даних останні треба подати у вигляді структур даних. Найпоширенішими зображеннями множин геометричних об'єктів в обчислювальній геометрії є списки (односторонні, одно- та двозв'язні кільцеві, стеки, черги, РСПЗ тощо), дерева (бінарні та *k*-арні, дерева відрізків, *k*-*d*- та АВЛ-дерева, зчеплені черги, червоно-чорні дерева тощо), хеш-таблиці [9–12]. Коротко охарактеризуємо основні структури даних і ефективність їх застосування на деяких прикладах задач.

Означення. *Опис складних об'єктів засобами простіших типів даних, які безпосередньо зображуються у машині, називається* ***структурами даних****.*

Структури даних складаються із: структури пам'яті для зберігання даних, способів її формування, модифікації та доступу до даних, набору операцій над елементами даних.

1.5. ТИПИ СТРУКТУР ДАНИХ

Основні типи структур даних детально представлено в роботах [5, 9–12], ми ж зупинимося на тих структурах даних, які часто використовуються в обчислювальній геометрії.

Списки

Означення

1. ***Списком*** *(довжиною n) називається впорядкована послідовність елементів* $a_1, a_2, ..., a_n$. *Список розміром 0 називається* ***порожнім****.*

Випавничо-поліграфічний центр "Київський університет" Версія не для друку

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. 1.5).

4. ***Двосторонній зв'язаний список (двозв'язний)** містить три поля: ключ і два вказівники – наступний і попередній. Кільцевий двосторонній зв'язаний список (кільцевий двозв'язний) – це список із вказівником з останнього елемента на перший і з першого на останній* (рис. 1.5).

Рис. 1.5. Односторонні та двосторонні списки

Прикладом застосування таких списків є зображення багатокутників (полігонів) (рис. 1.6).

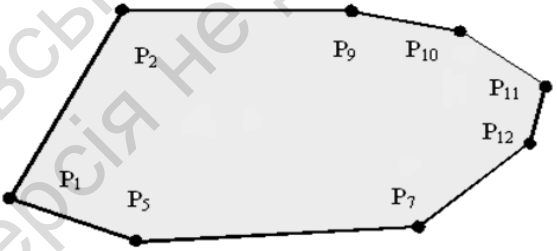


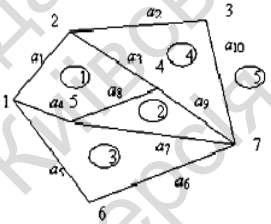
Рис. 1.6. Полігон

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

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

6. *Реберний список із подвійними зв'язками (РСПЗ)* [5]. Досить часто в задачах обчислювальної геометрії використовується в тій чи іншій формі спеціальний список для зображення планарного розбиття – реберний список із подвійними зв'язками. Він подається у вигляді шестивимірного масиву таким чином.

Нехай $V = \{v_1, v_2, \ldots, v_N\}$, а $E = \{e_1, e_2, \ldots, e_M\}$. Головна компонента РСПЗ для планарного графа (V, E) – це *реберний вузол* для ребра e, 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 e, поле V_2 – його кінець; у такий спосіб ребро e отримує умовну орієнтацію. Поля F_1 і F_2 містять імена граней, які лежать, відповідно, ліворуч і праворуч від ребра e, орієнтованого від V_1 до V_2. Вказівник P_1 задає реберний вузол, що містить перше інцидентне вершині V_1 вхідне ребро, яке ми отримуємо при повороті ребра e проти годинникової стрілки навколо V_1, а вказівник P_2 задає реберний вузол, що містить перше інцидентне вершині V_2 вихідне ребро, яке ми отримуємо при повороті ребра e проти годинникової стрілки навколо V_2. Імена граней і вершин можуть бути задані цілими числами. На рис. 1.7 наведено приклад зображення 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. 1.7. Зображення планарного розбиття за допомогою РСПЗ

Видавництво "Київський університет" Версія не для друку

Дерево відрізків. Використовується в задачах регіонального пошуку як самостійна структура або входить як складова до дерева регіонів. Дерево відрізків визначене на цілочисловому інтервалі [*L*, *N*] і будується рекурсивно як двійкове дерево (рис. 1.8). На дереві визначено операції вставлення та вилучення інтервалу. Розглянемо застосування цієї структури на прикладі задачі регіонального пошуку.

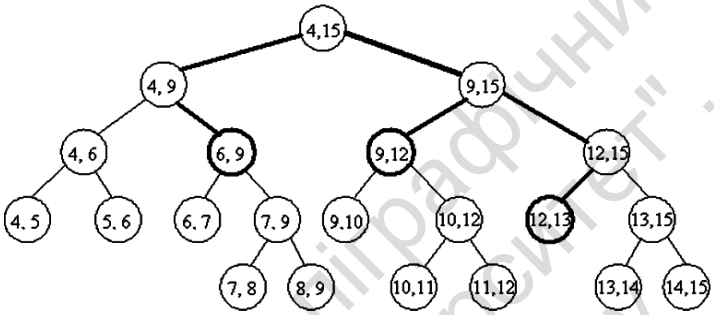


Рис. 1.8. Дерево відрізків $T(4, 15)$ для інтервалу [4, 15]. Вставлення інтервалу [6, 13] в $T(4, 15)$; [6, 13] = [6, 9] ∪ [9, 12] ∪ [12, 13]. Вузли віднесення: [6, 9], [9, 12], [12, 13]

Задача 1.2 (регіонального пошуку). *На заданій множині S із n точок запитуваного регіону R знайти підмножину точок множини S* (*або їхню кількість*)*, які містяться в регіоні R* (рис. 1.9).

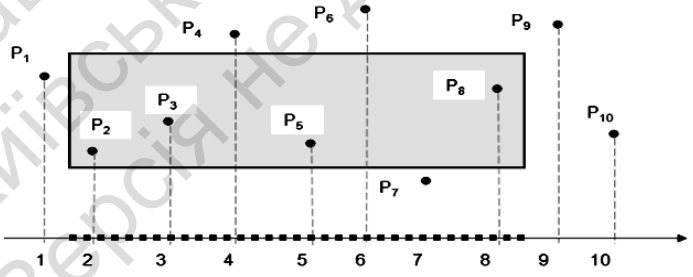


Рис. 1.9. Приклад задачі регіонального пошуку

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

Задача бінарного пошуку розмірністю N зводиться за час $O(1)$ до задачі регіонального пошуку, а тому нижня оцінка складності задачі регіонального пошуку становить $\Omega(\log n)$. Одним із методів, який дозволяє розробити алгоритм з оцінкою складності, близькою до $O(\log n)$, є метод дерева регіонів [5], який використовує структуру даних – дерево регіонів (рис. 1.10).

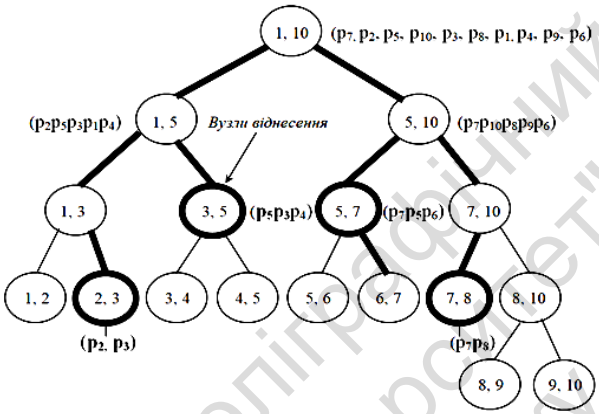


Рис. 1.10. Дерево регіонів

Ідея методу. Будується дворівнева структура даних (дерево регіонів), перший рівень якої – дерево відрізків по одній із координат (напр. x), а другий – вузли дерева відрізків, прошиті впорядкованими списками по іншій координаті (напр. y). На побудованому дереві регіонів виконується операція вставлення інтервалу (проєкція запитуваного 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 в комп'ютерних іграх, моделюванні, комп'ютерній графіці, задачах комп'ютерного зору, базах даних, а також інших алгоритмічних задачах. На рис. 1.11 показано приклад такого дерева для множини S із n точок на площині. Для його побудови застосовується рекурсивна процедура, яка на

Видавничо-поліграфічний центр "Київський університет" Всі права на друку

кожному кроці рекурсії почергово (по координатах 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 задану множину точок (рис. 1.11 а).

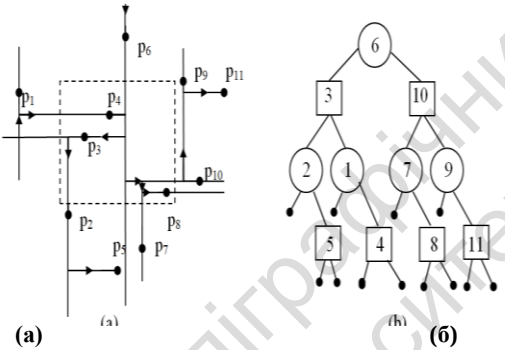


Рис. 1.11. 2-d-дерево: а) підрозбиття; б) дерево пошуку

Паралельно розбиттю формується бінарне дерево пошуку (рис. 1.11 б), яке називають *багатовимірним деревом пошуку* (*k-d-деревом пошуку*). Зокрема, ця структура даних ефективно використовується для розв'язання задачі 1.2 та запитуваного регіону прямокутної форми, сторони якого паралельні осям координат (рис. 1.12).

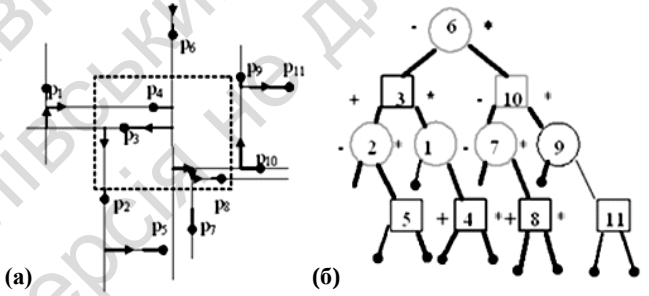


Рис. 1.12. Регіональний пошук методом 2-d дерева

Видавництво поліграфічний центр "Київський університет". Версія не для друку

Дерево інтервалів [9, 10]. Ця структура даних також використовується при розв'язанні задач регіонального пошуку.

Задача 1.3. *Задано N ортогональних відрізків на площині та запитуваний регіон R (прямокутник). Знайти множину відрізків, яка має непорожній перетин із запитуваним регіоном R* (рис. 1.13).

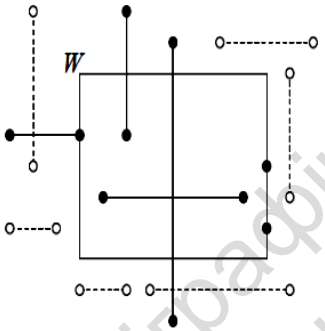


Рис. 1.13. Перетин ортогональних відрізків із запитуваним регіоном

Доведення. Структура даних *дерево інтервалів* (у поєднанні з деревом регіонів) дає час $O(\log^2 n)$. Для розв'язання задачі за кожною з координат застосовуємо дерево інтервалів. На рис. 1.14 наведено приклад дерева інтервалів для відрізків, паралельних осі *OX*.

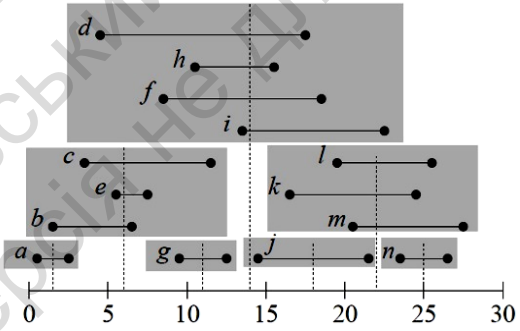


Рис. 1.14.(а) Дерево інтервалів для відрізків, що паралельні осі *OX*

Видавничо-поліграфічний центр "Київський університет". Версія не для друку

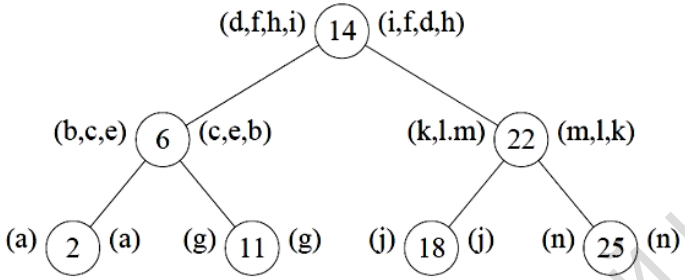


Рис. 1.14.(б) Дерево інтервалів для відрізків, що паралельні осі *OX*

Зчеплена черга [5]. Зчеплена черга – це бінарне дерево із вказівниками, вузлами якого є точки заданої множини. Опуклу оболонку можна зобразити у вигляді лінійного списку

$$CH(S) = \{P_5, P_1, P_2, P_9, P_{10}, P_{11}, P_{12}, P_7\}$$

або у вигляді зчепленої черги (рис. 1.15).

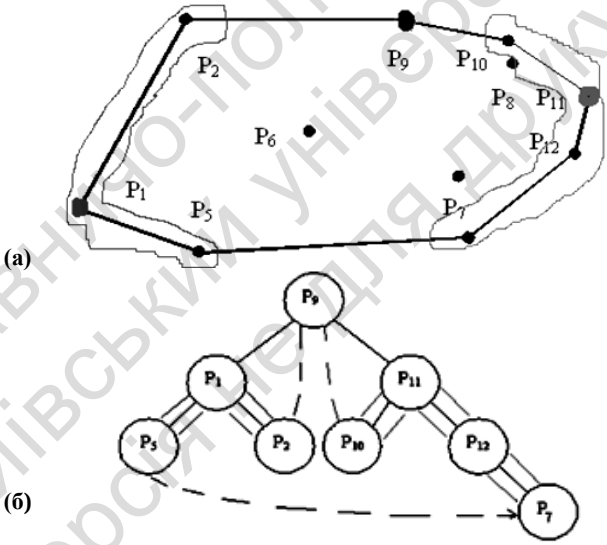


Рис. 1.15. Приклад опуклої оболонки (а) та зображення її границі у вигляді зчепленої черги (б)

Видавничо-поліграфічний центр "Київський університет" Верстка не для друку

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, зокрема динамічних задач. Розглянемо динамічну задачу пошуку опуклої оболонки на множині S із n точок на площині.

Задача 1.4. *Для множини точок S, яка формується додаванням точок із постійною затримкою надходження, побудувати структуру даних, яка підтримує поточну опуклу оболонку і дозволяє корегувати її при надходженні нової точки. Нижня оцінка корекції відома і становить* $\Omega(\log n)$.

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, що надійшла нова точка P (рис. 1.16).

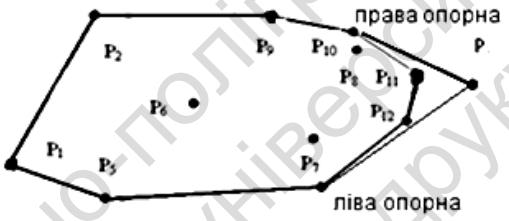


Рис. 1.16. Вставлення точки в опуклу оболонку

Для пошуку лівої і правої опорних точок для вставлення нової точки рухаємося лінійним списком $CH(S)$ (використовуючи відповідну класифікацію точки, рис. 1.17) і за час $O(n)$ одержуємо результат у найгіршому випадку (рис. 1.18).

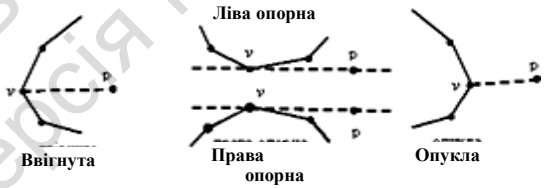


Рис. 1.17. Класифікація точки

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

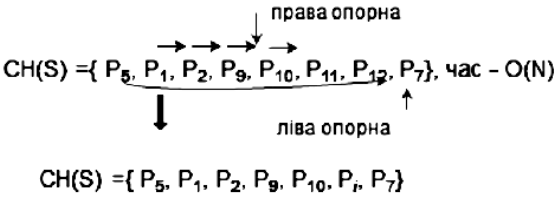


Рис. 1.18. Вставлення точки із використанням зображення границі опуклої оболонки у вигляді списку CH(S)

Якщо ж опуклу оболонку зобразити у вигляді зчепленої черги (рис. 1.19), то пошук опорних точок і вставлення нової точки ми виконаємо за час $\theta(\log n)$ у найгіршому випадку.

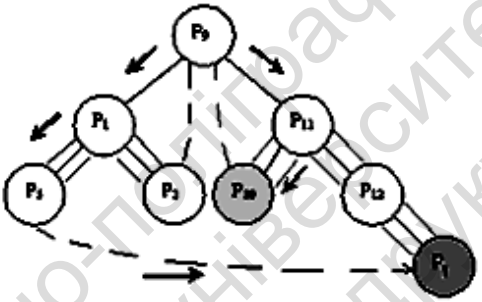


Рис. 1.19. Вставлення точки із використанням зображення границі опуклої оболонки у вигляді зчепленої черги

Видавничо-поліграфічний центр "Київський університет"
Версія не для друку

2. ЗАДАЧІ НА ОХОПЛЕННЯ

2.1. ПОБУДОВА НАЙМЕНШОГО ОХОПЛЮЮЧОГО КОЛА НАВКОЛО МНОЖИНИ ТОЧОК (ОМК)

Задача є класичною, їй присвячено багато наукової літератури [13–18]. Пошук ефективного алгоритму розв'язання цієї задачі почав ще у 1860 р. Сильвестр [13]. Теоретичну базу було закладено у праці Радамахера і Тепліца у 1957 р. [58]: найменше охоплююче коло – єдине й або має за діаметр відрізок, що сполучає деякі дві точки множини, або є описаним колом для деякої трійки точок із вихідної множини. Таким чином, із вищесказаного можна запропонувати простий перебірний алгоритм зі складністю $O(n^4)$. У 1972 р. Елзінг і Хірн [17] поліпшили цей алгоритм до $O(n^2)$.

У 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 якої найбільша з відстаней до точок заданої множини була б мінімальною, тобто задовольняла критерій

$$P_0 = \min \max \left\{ (x_0 - x_i)^2 + (y_0 - y_i)^2 \right\}. \qquad 2.1$$

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. Його також використовують для оптимального розміщення радіопередавача, що обслуговує N приймаючих приладів, щоб мінімізувати необхідну потужність приладу [59]. Деякі автори розглядали цей критерій як задачу неперервної квадратичної оптимізації [17]. Це спричинило появу цілої низки ітеративних алгоритмів (на кшталт градієнтних методів) розв'язання розглядуваної задачі, яка, безумовно, є дискретною.

У теорії розташування об'єктів ця задача відома як *задача про 1-центр*. Існує багато відомих методів її розв'язання [14]. Наприклад, із застосуванням діаграми Вороного це можна зробити за час

Видавничо-поліграфічний центр "Київський університет" Роздруковано для друзів

$O(n*\log(n))$ [5]. Використовуючи метод пошуку з відрізанням, Меджиддо [16] одержав алгоритм знаходження найменшої гіперсфери у m-вимірному евклідовому просторі, що оточує задану скінченну сукупність точок, який має складність $O(n)$.

Розглядувана задача має велике практичне застосування. Наприклад, за її допомогою можна знайти мінімальну допустиму потужність радіопередавача і його місцезнаходження для забезпечення зв'язком певного населеного пункту чи військової частини. З розвитком в Україні мобільних телекомунікацій актуальність задачі ще підвищилась, оскільки оптимальне розташування передаючих антен значно знижує енергетичні витрати передавача і поліпшує якість зв'язку. Саме тому її програмна реалізація, а точніше реалізація ефективного алгоритму її розв'язання, є гідним товаром на ринку ІТ-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 всю множину S.*

Нижче детально розглянемо методи її розв'язання.

2.1.1. Алгоритм із використанням діаграми Вороного

Щоб отримати розв'язок задачі ОМК, кращий ніж $O(n^2)$, пропонуємо підхід, в основі якого лежать побудова діаграми Вороного і пошук діаметра множини точок та діаметра опуклої оболонки [5]. Розглянемо цей підхід детально.

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, то шукаємо серед них найвіддаленішу від діаметра D точку.

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

Видавничо-поліграфічний центр "Київський університет" Відділення друку

Теорема 2.1 [18]. *Діаметр множини дорівнює діаметру її опуклої оболонки.*

Ця теорема дозволяє уникнути обчислення відстані для кожної пари точок.

Теорема 2.2 [19]. *Діаметр опуклої фігури дорівнює найбільшій із відстаней між двома паралельними опорними прямими до цієї фігури.*

Ця теорема дозволяє уникнути зайвих перевірок відстаней між парами точок і розглядати лише протилежні пари точок. Очевидно, що паралельні прямі можна провести не через кожну пару точок (рис. 2.1). Це означає, що не обов'язково перевіряти всі відрізки в опуклій оболонці на можливість 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 всі протилежні пари, не перебираючи всі можливі варіанти точок.

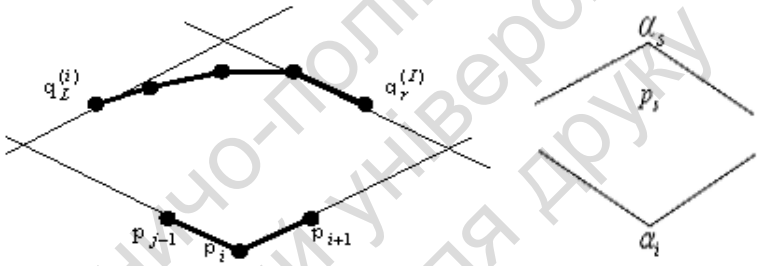


Рис. 2.1. Приклад розташування паралельних прямих

Розглянемо вершину p_i опуклого багатокутника P, вершини якого пронумеровані проти годинникової стрілки (рис. 2.1). Будемо рухатись границею багатокутника P проти годинникової стрілки. Почавши з вершини p_i, будемо рухатись до тих пір, поки не досягнемо вершини $q_R^{(i)}$, яка максимально віддалена від $p_{i-1}p_i$. Аналогічно визначаємо вершину $q_L^{(i)}$, 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, становить множину $C(p_i)$, кожна

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

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 відстань від відрізка до точки можна обчислювати як площу трикутника, побудованого на точках $p_i = (x, y)$, $p_{i+1} = (x', y')$, $p = (x'', y'')$, за формулою (2.2):

$$S = \begin{vmatrix} x & y & 1 \\ x' & y' & 1 \\ x'' & y'' & 1 \end{vmatrix} / 2. \qquad 2.2$$

Основна операція, яка для цього необхідна, – 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] наведено 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 $3n/2$, вона лінійна відносно $|CH(S)|$. Тоді визначення діаметра $CH(S)$ можна виконати за лінійний час, оскільки максимум множини визначається за час, пропорційний її потужності. За відомою теоремою 4.19 із [5] випливає таке:

Теорема 2.3 [5]. *Діаметр опуклого n-вершинного багатокутника може бути знайдений за O(n)-лінійний час.*

Під час роботи алгоритму вхідні дані (точки) можуть так розташуватися, що коло, яке ми будуємо на знайденому діаметрі, може містити не всі точки заданої множини. У цьому випадку проблему може розв'язати наступний крок алгоритму, який використовує діаграму Вороного для віддаленої точки.

Ми знаємо, що мінімальне коло, яке охоплює множину, визначається або діаметром заданої множини, або трьома її точками. Саме тому, якщо коло не охоплює всю множину, то необхідно знайти новий центр кола найменшого радіуса, яке б охоплювало всю множину точок. Для цього варто скористатись діаграмою Вороного, одна із властивостей якої полягає в тому, що її вершини є центрами кіл, які описані навколо сусідніх точок, тобто центр шуканого кола збігатиметься з однією із вершин

Видавничо-поліграфічний центр "Київський університет" Власність бібліотеки

діаграми Вороного. Оскільки діаграма Вороного містить лише $O(n)$ точок, а радіуси кіл, що пов'язані з кожною з вершин діаграми, дорівнюють відстані від точки до будь-якої із сусідніх, відповідних до неї точок, то мінімальна по всіх вершинах відстань дає радіус нашого кола. Остаточний алгоритм буде таким:

Алгоритм

1. Побудова опуклої оболонки з використанням діаграми Вороного ($O(n)$).
2. Знаходження протилежних пар опуклої оболонки ($O(n)$).
3. Знаходження діаметра множини S ($O(n)$).
4. Перевірка охоплення діаметром D усієї множини S ($O(n)$).
5. Якщо після перевірки залишились точки за межами кола діаметром D, то будуємо діаграму Вороного множини точок ($O(n\log n)$).
6. Пошук вершини діаграми Вороного, що є центром кола, яке охоплює S ($O(n)$).

Оцінка складності алгоритму

Теорема 2.4. *Для заданої множини S із n точок на площині задачу про найменше охоплююче коло можна розв'язати на основі діаграми Вороного за час $O(n\log n)$ з використанням $O(n)$ пам'яті.*

Доведенням теореми є сам алгоритм, описаний покроково вище.

2.1.2. Алгоритм побудови базового трикутника

Розглянемо ідею алгоритму, схожого на попередній, з тією лише різницею, що замість діаграми Вороного застосуємо базовий трикутник (трикутник максимальної площі, вписаний в опуклу оболонку множини S). Для цього скористаємось такими теоремами:

Теорема 2.5. *Коло, описане навколо опуклої оболонки, проходить хоча б через одну точку, що належить кінцю відрізка, який є діаметром цієї опуклої оболонки.*

Видавничо-поліграфічний центр "Київський університет" Вороне не для друку

Теорема 2.6. *Трикутник, який є базою для побудови кола навколо опуклої оболонки, має хоча б одну вершину, що належить діаметру.*

Ураховуючи ці теореми, можна запропонувати алгоритм пошуку трикутника.

Алгоритм пошуку трикутника

1. Якщо в опуклій оболонці більше одного діаметра, то вибираємо перший з них.

2. Припускаємо спочатку, що трикутнику належать обидва кінці діаметра, V_1 та V_2. Вибираємо кінці діаметра (згідно з теоремою 1.6 вони можуть бути вершинами шуканого трикутника).

3. Визначаємо точку V_3 опуклої оболонки, яка утворює із цими двома вершинами трикутник найбільшої площі ($O(\log n)$).

4. Якщо коло, описане навколо цього трикутника, містить усю множину точок, то алгоритм завершено.

Інакше:

5. Припускаємо, що лише одна точка (кінець діаметра) V_1 належить шуканому трикутнику.

6. Зафіксувавши V_1 та V_3, шукаємо нове положення точки V_2 таке, щоб ці точки утворювали трикутник найбільшої площі ($O(\log n)$).

7. Якщо коло, описане навколо цього трикутника, містить усю множину точок, то алгоритм завершено.

Інакше:

8. Припускаємо, що лише друга точка (кінець діаметра) V_2 належить шуканому трикутнику. Повторивши попередні кроки, знаходимо центр кола, описаного навколо опуклої оболонки.

9. Аналогічні міркування повторюємо й для інших діаметрів, якщо для першого наші припущення не виконуються. Оскільки кількість діаметрів у найгіршому випадку може бути $O(n)$, то час на пошук трикутника становитиме $O(n\log n)$.

Отже, загальний алгоритм розв'язання задачі ОМК буде таким:

Загальний алгоритм побудови базового трикутника

1. Побудова опуклої оболонки (напр. методом "розділяй та володарюй") ($O(n\log n)$).

Видавничий відділ Навчально-методичного центру "Київський університет" Версія 1.0 для друку

2. Знаходження протилежних пар опуклої оболонки ($O(n)$).
3. Знаходження діаметра множини S ($O(n)$).
4. Перевірка охоплення діаметром усієї множини S $O(n)$.
5. Пошук базового трикутника ($O(n\log n)$).

Оцінка складності алгоритму

Теорема 2.7. *Для заданої множини S із n точок на площині задачу про найменше охоплююче коло можна розв'язати методом базового трикутника за час $O(n\log n)$ з використанням $O(n)$ пам'яті.*

Доведенням теореми є сам алгоритм, описаний покроково вище.

2.1.3. Алгоритм на основі пошуку трьох базових точок

Оскільки коло можна визначити трьома точками, то запропонуємо алгоритм, схожий на алгоритм із п. 2.1.2. На відміну від базового трикутника, можна шукати просто вершини трикутника. Отже, наступний алгоритм може бути охарактеризований такою властивістю:

Теорема 2.8. *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 випливає, що побудову мінімального охоплюючого кола можна здійснити через дві або три точки.

Алгоритм

1. Для заданої множини точок побудувати опуклу оболонку за методом Грехема.
2. Серед точок, які є вершинами побудованої в п. 1 опуклої оболонки, є найвіддаленіша пара. Назвемо ці дві точки базовими.
3. Для кожної з n-2 точок (дві виключені точки – базові) виконується пошук кута, що утворений базовими та поточною (однією з n-2) точками. Звісно, ідеться про кут при поточній вершині.
4. Серед знайдених у п. 3 кутів необхідно вибрати три найгостріші кути і відповідні до них точки (три, оскільки, як зазна-

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

чалося вище, для побудови мінімального кола достатньо саме трьох точок, а у випадку, коли це коло не проходитиме через одну або дві з базових точок, для його побудови будуть потрібні саме три точки, які шукаємо в цьому пункті).

5. Перебрати всі можливі комбінації базових точок і точок, знайдених у п. 4. Потужність множини точок для перебирання не буде залежати від n. До того ж можливих варіантів завжди буде 10. Для кожного варіанта набору точок будуємо коло.

6. Для побудованих у п. 4 кіл перевіряємо на приналежність їм усі n вхідних точок.

7. Серед кіл, що успішно пройшли перевірку, з п. 6 вибираємо коло найменшої площі.

Оцінка складності алгоритму

Теорема 2.9. *Мінімальне охоплююче коло для заданих N точок на площині можна визначити за допомогою трьох вершин, які описують коло, за час* $O(n\log n)$.

Доведення. На етапі попередньої обробки виконується побудова опуклої оболонки вхідної множини точок. Складність її буде дорівнювати $O(n*\log(n))$. Отже, пошук найвіддаленішої пари вимагатиме тієї самої складності. Кути з п. 3 алгоритму можна знайти за лінійний час, оскільки вимагається лише обрахування кута для кожної із n точок. Побудова кіл для кожного з десяти 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 три алгоритми мають однакову оцінку складності $O(n\log n)$ і зводяться до пошуку діаметра через опуклу оболонку і третьої точки кола. Саме пошуком третьої точки кола і відрізняються, по суті, ці підходи. У першому використовується діаграма Вороного, у другому – базовий трикутник, у третьому – звичайний перебірний пошук. Ураховуючи зазначене, можна узагальнити перший метод, побудувавши на етапі попередньої обробки діаграму Вороного ($O(n\log n)$). Тоді всі інші процедури, включаючи побудову опуклої оболонки, пошук діаметра та базового трикутника, будуть виконуватись за лінійний час.

Видавничо-поліграфічний центр "Київський університет"

Меджиддо [16] у 1983 р. поставив питання щодо необхідності мати повну інформацію про діаграму Вороного дальньої точки для знаходження центра найменшого охоплюючого кола, коли достатньо дослідити лише ті вершини діаграми, які містяться в певному околі від шуканого центра. Створення ним методу побудови лише потрібної частини діаграми дозволило досягти оптимальної лінійної складності алгоритму розв'язання.

2.2. ТРИКУТНИК НАЙМЕНШОЇ ПЛОЩІ (ОМТ)

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

Задачі, в основі яких лежить пошук найменшого описаного трикутника для множини точок, мають широке застосування в інженерних та дослідницьких проєктах, зокрема в робототехніці та моделюванні руху великих зграй пташок, риб, інших скупчень живих організмів і безпілотних літальних апаратів [60–64]. Клі та Ласковскі [65, 66] запропонували алгоритм зі складністю $O(n\log^2 n)$ для знаходження всіх локальних мінімумів (відносно площі) серед трикутників, що містять опуклий n-кутник P. У роботі [67] запропоновано оптимальний алгоритм. Розглянемо найліпші реалізації.

Алгоритм із використанням опуклої оболонки

Алгоритм (*головна ідея*)

1. Передусім будуємо за одним із популярних алгоритмів (напр. Грехема) опуклу оболонку $CH(S)$ як найменшу опуклу множину, яка містить усю множину точок S.

2. Наступним кроком є побудова ОМТ, який описує одержану опуклу оболонку. Для цього застосовуємо такі твердження [65]:

Теорема 2.10. *Якщо T – локальний мінімум серед трикутників, що містять деяку опуклу оболонку, то середина кожної сторони T дотикається до цієї опуклої оболонки.*

Видавничо-поліграфічний центр "Київський університет" Зразок не для друку

Теорема 2.11. *Якщо T – локальний мінімум серед трикутників, що містять деяку опуклу оболонку, то принаймні одна сторона T і ребро опуклої оболонки лежать на одній прямій.*

Твердження 2.1. *Середини сторін трикутника, який охоплює опуклу оболонку, мають до неї дотикатися* (рис. 2.2).

Існує локальний мінімум, що охоплює трикутник принаймні із двома сторонами, які лежать на ребрах опуклої оболонки.

Третя сторона трикутника може лежати на ребрі або дотикатися до опуклої оболонки (рис. 2.3).

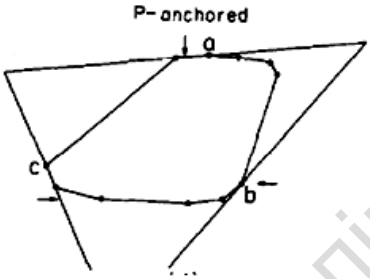


Рис. 2.2. Приклад того, як середини сторін трикутника дотикаються до ребер опуклої оболонки

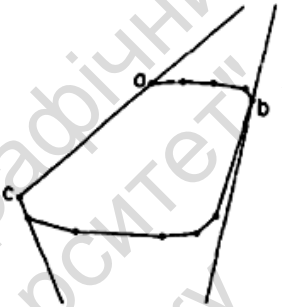


Рис. 2.3. Приклад того, як третя сторона трикутника дотикається до границі опуклої оболонки

Теорема 2.12. *Трикутник найменшої площі, який охоплює опуклу оболонку, існує тоді й тільки тоді, коли хоча б дві його сторони лежать на двох сторонах цієї опуклої оболонки CH*(*S*).

Алгоритм (*покроково*)

Медіана однієї зі сторін трикутника буде деякою точкою опуклої оболонки, а дві інші сторони лежатимуть на одній прямій із ребром опуклої оболонки або одній прямій і точці, яка належить опуклій оболонці. Ми беремо пряму і точку з опуклої оболонки або дві прямі з неї, знаходимо третю точку та визначаємо вершини трикутника, перебираючи їх. Далі знаходимо площу кожного трикутника за допомогою формули Герона, обираючи таким чином трикутник найменшої площі.

Видавничо-поліграфічний центр "Київський університет" Зразок не для друку

Отже, можна виділити такі основні етапи алгоритму:

1. Пошук опуклої оболонки множини заданих точок методом Грехема.

2. Перебирання двох сторін опуклої оболонки, які будуть на одному рівні зі сторонами результуючого трикутника.

3. Пошук точки перетину знайдених на попередньому кроці прямих.

4. Пошук третьої точки, тобто медіани третьої сторони результуючого трикутника.

5. Знаходження двох останніх точок результуючого трикутника.

6. Здійснюючи обхід списку ребер опуклої оболонки, визначимо трикутник і його площу.

7. Обираємо зі знайдених трикутників той, який має найменшу площу.

Оцінка складності алгоритму

Теорема 2.13. *Мінімальний охоплюючий трикутник для заданої множини S із n точок на площині можна визначити за допомогою побудови опуклої оболонки і теореми 2.12 за час O(nlogn)* (рис. 2.4).

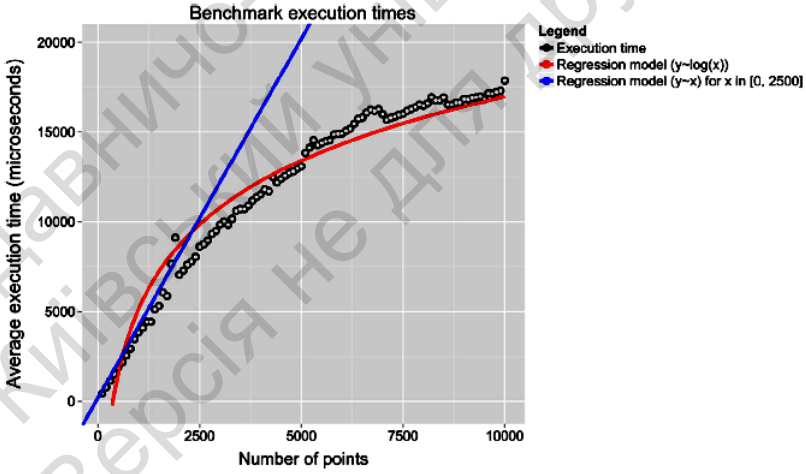


Рис. 2.4. Графік часової складності алгоритму

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

2.3. ПОБУДОВА ПРЯМОКУТНИКА НАЙМЕНШОЇ ПЛОЩІ, ЩО МІСТИТЬ ЗАДАНУ МНОЖИНУ ТОЧОК (ОМП)

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

Результати розв'язання задачі можуть бути застосовані до деяких випадків оптимізації задачі регіонального пошуку. Досить часто деяку множину точок розглядають як один об'єкт і вважають, що всі точки множини належать запитуваному регіону, якщо хоча б одна точка йому належить. У цих випадках із метою оптимізації можна розглядати перетин прямокутника, що містить таку множину, і запитуваного регіону. Якщо такий прямокутник не порожній, то вважатимемо, що точки належать запитуваному регіону. Оптимізація тут зрозуміла: якщо в нас є багато об'єктів, кожен з яких містить велику кількість точок, то заміна всіх точок об'єкта одним прямокутником дійсно зменшить час пошуку.

Картографічні системи також містять багато багатокутників складної форми. Вони зображують будинки, райони, річки, дороги тощо. Кількість таких об'єктів може досягати десятків тисяч, а виведення їх на екран ЕОМ займатиме багато часу, оскільки є повільною операцією. До того ж під час виведення кожного об'єкта можуть кілька разів змінюватися стилі екрана, такі як колір, тип лінії, що також сильно уповільнює виведення. Отже, бажаним було б виводити тільки ті об'єкти, які будуть відображатися хоча б частково. Очевидною є задача регіонального пошуку, де запитуваний регіон – прямокутник на карті, що відповідає екрану ЕОМ. Якщо ж шукати не сам об'єкт, а відповідний прямокутник, то отримаємо значну оптимізацію.

Прямокутники можна використовувати також у статистиці для зменшення кількості даних, що зберігаються. Наприклад, часто множину точок заміняють прямокутником і числом –

Видавничий центр "Київський університет" Вовчанецький О.Д.

кількістю точок, що входять до нього. У такому 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 всю множину S.*

Алгоритм побудови з використанням опуклої оболонки

Ідея розв'язання задачі про ОПМ у цьому підході полягає в побудові опуклої оболонки заданої множини точок. Отже, у першій частині алгоритму будуємо опуклу оболонку. Якщо всі точки мають належати шуканому прямокутнику, то достатньо знайти прямокутник, якому належать усі точки опуклої оболонки. Усі інші точки містяться 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 у знаходженні самого прямокутника. Тут використовуємо припущення, що хоча б одна зі сторін прямокутника містить ребро опуклої оболонки. Це припущення легко обґрунтувати.

Припустимо від супротивного, що на кожній із сторін прямокутника лежить лише одна точка. Розглянемо лише чотири точки, якими опукла оболонка не спирається на багатокутник, які просто обмежують кут повороту до упору і більше ніякої ролі не відіграють (рис. 2.5). У цьому випадку (нижня точка лівіша за верхню, права нижче за ліву) наш прямокутник не є оптимальним варіантом побудови прямокутника найменшої площі, оскільки при повороті за годинниковою стрілкою ми досягнемо кращого результату, зменшивши і ширину, і висоту.

Розглянемо випадок на рис. 2.6. Чи можна бути впевненими, що роблячи поворот так, як на рис. 2.5, ми отримаємо кращий результат? І чому сторона опуклої оболонки збігатиметься зі стороною прямокутника?

Відповідаючи на ці запитання, припустимо, що розміри початкового прямокутника $h = d_1 \sin\alpha$, $w = d_2 \cos\beta$. При повороті пря-

Видавничо-поліграфічний центр "Київський університет" Вилучено для друку

мокутника на кут φ (за годинниковою стрілкою кут від'ємний, проти – додатний) h та w змінять свої значення: $h^* = d_1 \sin(\alpha + \phi)$, $w^* = d_2 \cos(\beta + \phi)$. Тоді площа прямокутника становитиме

$$S^* = h^* w^* = d_1 d_2 \sin(\alpha + \phi)\cos(\beta + \phi),$$

звідки

$$dS^*/d\phi = d_1 d_2[\cos(\alpha + \beta)\cos(\beta + \phi) - \sin(\alpha + \phi)\sin(\beta + \phi)] =$$
$$= d_1 d_2 \cos(\alpha + \phi + \beta + \phi).$$

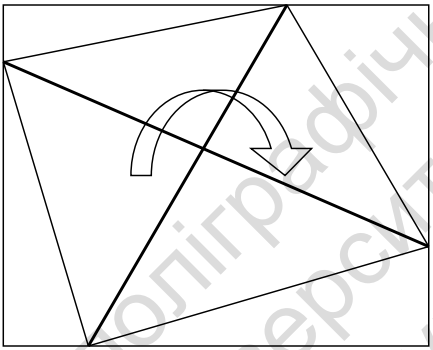


Рис. 2.5. Випадок, коли нижня точка лівіша за верхню, а права нижча за ліву

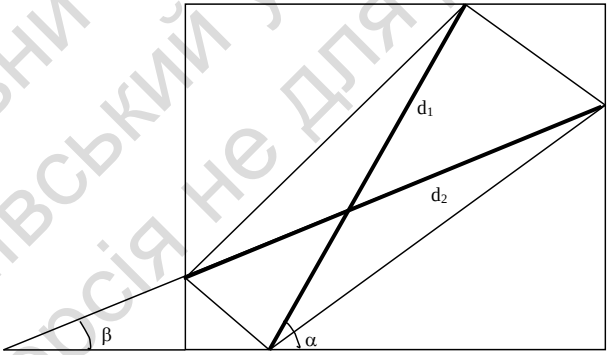


Рис. 2.6. Випадок, коли нижня точка лівіша за верхню, а права вища за ліву

Видавничо-поліграфічний центр "Київський університет". Версія не для друку

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 його площа буде монотонно зменшуватись, а значить, ми одержимо випадок з рис. 2.5, коли будуть монотонно зменшуватися ширина та висота.

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

Для кожного ребра оболонки шукаємо найвищу, найлівішу та найправішу точки оболонки відносно зазначеного ребра. Вони будуть задавати координати кутів прямокутника. Найменший серед таких прямокутників і буде шуканим. Алгоритм складається із двох етапів.

1. Будуємо опуклу оболонку одним із відомих методів (напр. Грехема або методом *загортання подарунка* (gift wrapping)). Вона буде задана послідовністю точок, упорядкованою відносно найвищої точки за кутом проти годинникової стрілки. Верхня точка буде першою у списку вершин оболонки.

2. Шукаємо мінімальний прямокутник.

Для кожного відрізка опуклої оболонки будуємо прямокутник, що містить найвищу, найлівішу та найправішу точки відносно прямої, заданої цим відрізком. Для кожного прямокутника знаходимо площу та шукаємо найменшу площу.

Розглянемо процес детальніше.

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 задаватиме напрямок осі OX. Вісь OY повертатиметься на $\pi/2$ проти годинникової стрілки відносно OX (тобто система координат стандартна).

2.2. Найвища точка матиме найбільшу ординату в такій системі. Її ми знаходимо за допомогою двійкового пошуку, використовуючи впорядкованість точок оболонки.

2.3. Найправіша точка матиме найбільшу абсцису. Її також знаходимо двійковим пошуком, проте логічно розглядати не всю оболонку, а ту її частину, що міститься між i-ю точкою та найвищою. Аналогічно шукаємо найлівішу точку (з найменшою абсцисою).

Видавничо-поліграфічний центр "Київський університет"

2.4. Прямокутник також шукаємо спочатку в системі координат ребра. Ординати двох його нижніх кутів дорівнюють нулю, а двох верхніх – ординаті знайденої найвищої точки. Абсциси лівих і правих кутів дорівнюють абсцисам відповідно найлівішої та найправішої точок оболонки.

Зрозуміло, що такому прямокутнику належать усі точки з множини. Площу його також легко знайти, оскільки сторони його паралельні осям системи координат ребра. Знайдену площу порівнюємо із площею прямокутника, який вважали найменшим до розглядання зазначеного ребра. Обираємо найкращий і його координати перетворюємо на екранні. Найменший після обробки всіх ребер прямокутник матиме найменшу можливу площу. Опишемо тепер структури даних і сам алгоритм детальніше.

Алгоритм (*реалізація*)

Спочатку розглянемо опис констант і 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` міститься поточна кількість точок, які зберігаються у двох масивах: `PX` та `PY`. Для знаходження координат *X* та *Y* *i*-ї змінної потрібно взяти *i*-й 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 у масивах `PX`, `PY` та змінною `Top` (кількість елементів у стеку). Для подальшої роботи елементи, що відповідають номерам зі стеку, копіюються у відповідні масиви `OpX` та `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 (найви-

Видавничо-поліграфічний центр "Київський університет" Росія НЕ для друку

щий) елемент опуклої оболонки зустрічатиметься в масивах двічі (OpX, OpY) – як перший і останній. У такий спосіб опукла оболонка замикається, що спрощує подальший бінарний пошук.

Опишемо тепер пошук прямокутника. Для зберігання прямокутників використовується 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 уже зазначалося, пошук відбувається в циклі послідовно за ребрами опуклої оболонки. Проілюструємо порядок обходу ребер на рис. 2.7.

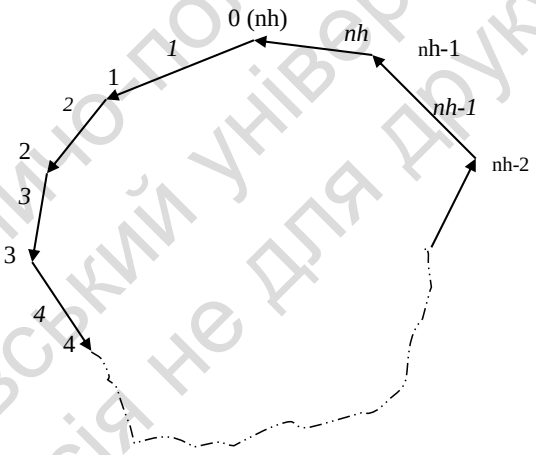


Рис. 2.7. Обхід ребер опуклої оболонки

Тут суцільною лінією зображено номери точок (у масивах), переривчастою – номери ребер (під час обходу). Змінна *i* є

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

лічильником циклу. Перед циклом площа найменшого прямокутника ініціалізується великим числом (10^{100}). Під час перебирання прямокутників воно зменшиться.

Розглянемо одну з ітерацій такого циклу. Спочатку знаходимо sin(α) та cos(α) кута між напрямленим ребром (віссю *OX* своєї системи координат) і 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 зазначеного ребра.

Розглянемо алгоритм пошуку на прикладі пошуку найвищої вершини. Спочатку зауважимо, що варто позначити вершини оболонки таким чином, щоб *i*-та вершина (кінець ребра) мала номер 0, а далі вершини нумерувалися проти годинникової стрілки та *i*-1-ша вершина (початок ребра) мала номер *nh*-1. Якщо цього не зробити, то нумерація нашої оболонки відносно ребра буде розірваною, що ускладнить бінарний пошук. Наведена нижче функція шукає координату *y* точки за її номером 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 у тілі функції шукає номер точки в масиві оболонки (варто нагадати, що *j* – номер відносно *i*-го кінця поточного ребра). Другий перетворює координату з екранної системи на реберну.

Тепер продемонструємо пошук найвіддаленішої за координатою *y* точки (рис. 2.8).

```
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;
```

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

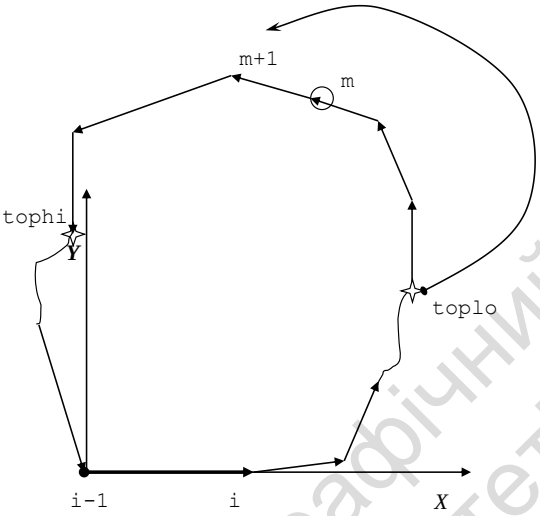


Рис. 2.8. Пошук найвіддаленішої за координатою *y* точки (*i*-та ітерація циклу)

Шукатимемо точку на інтервалі між точкою `toplo` (спочатку це кінець ребра) та `tophi` (початок ребра). На кожній ітерації будемо знаходити точку, що лежить посередині такого інтервалу, і наступну за нею точку та порівнювати їхні ординати. Відповідно до результатів порівняння точку буде переміщено в середину, початок або кінець відрізка. Зрозуміло, що не більше ніж через $O(\log n)$ буде виконана умова виходу з циклу `while` після знаходження відповідної точки. З рис. 2.8 стає зрозумілим алгоритм пошуку. Після відшукання крайніх точок легко знаходимо площу прямокутника, оскільки з координат крайніх точок у реберній системі координат легко знайти довжини його сторін. Залишилось лише перетворити координати прямокутника на екранні за допомогою такої процедури:

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

Видавничо-поліграфічний центр "Київський університет". Версія для друку

Оцінка складності алгоритму

Теорема 2.14. *Пошук прямокутника найменшої площі на множині S із N точок на площині можна виконати за час O*(n log*n*) *із використанням O*(*n*) *пам'яті.*

Доведення. Перший етап алгоритму – побудова опуклої оболонки методом Грехема. Він має складність *O*(*n*log*n*). На другому етапі отримуємо цикл довжиною *n* для всіх ребер опуклої оболонки. На кожній ітерації здійснюємо двійковий пошук у впорядкованій множині трьох точок, що потребує *O*(log*n*) часу. Отже, загальна складність пошуку прямокутника становить *O*(*n*log*n*). Таким чином, загальна складність усього алгоритму становитиме *O*(*n*log*n*). На рис. 2.9 показано приклад роботи алгоритму.

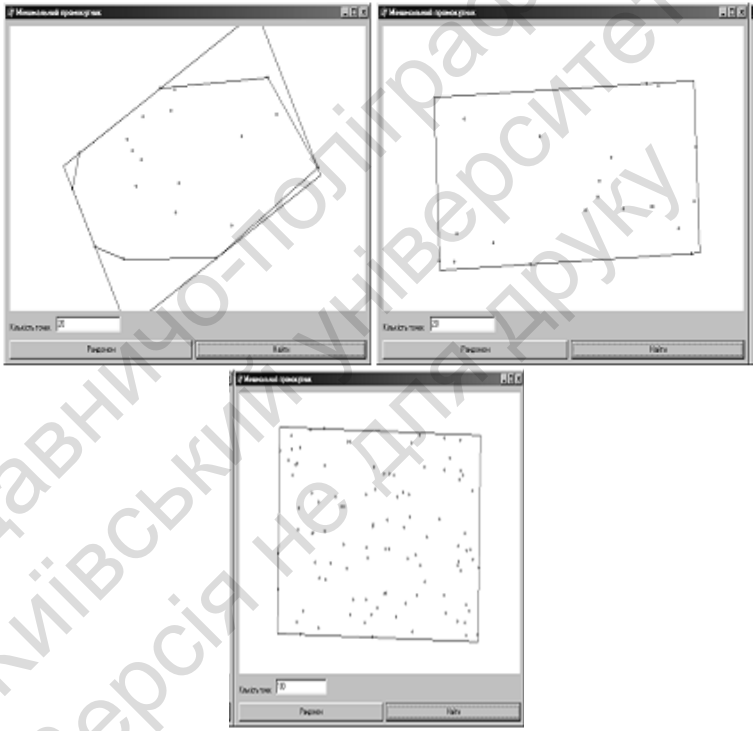


Рис. 2.9. Приклад програмної реалізації пошуку прямокутника найменшої площі

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

Висновки. З того, що було описано вище, можна зробити висновок, що розглядувана програма дійсно шукає прямокутник найменшої площі, який містить задані точки. На рис. 2.9 наведено приклад програмної реалізації задачі.

2.4. ПОБУДОВА НАЙМЕНШОГО ОХОПЛЮЮЧОГО ЕЛІПСА (ОМЕ)

Результати розв'язання задачі побудови найменшого охоплюючого еліпса застосовують у різних галузях, зокрема медицині. Еліпс мінімальної площі використовують при пошуку області диска зорового нерва [20], для виділення контурів розмитого зображення [21] тощо. Також розв'язки задачі застосовують у будівництві при визначенні місця під забудову об'єктів різної форми.

Щодо пошуку ефективних методів розв'язання задачі, то цій тематиці присвячено достатню кількість публікацій [21–24]. У роботах [22, 24] розглянуто і досліджено шість методів побудови еліпса (варіації з побудовою опуклої оболонки, пошуком трикутника найбільшої площі тощо). На жаль, дуже мала кількість матеріалів є у відкритому доступі. Проте є вже 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.*

2.4.1. Алгоритм побудови з використанням опуклої оболонки (варіант 1)

Алгоритм (основна ідея)

Перший варіант алгоритму був такий: серед заданих точок знайти три, що утворюють трикутник максимальної площі. Через ці точки провести еліпс Штейнера [26] (еліпс мінімальної площі, що проходить через три точки). Однак у ході досліджень було знайдено контрприклад, для якого зазначений алгоритм не підходив. Якщо побудувати еліпс Штейнера через три точки із чотирьох, що зада-

Видавничий центр "Київський університет" Версія не для друку

ють прямокутник, то побудований еліпс може не захоплювати четверту точку. Таким чином було встановлено, що для побудови потрібного еліпса необхідно як мінімум чотири точки. Отже, необхідно знайти дві найвіддаленіші точки, потім дві точки, які утворюють трикутники найбільшої площі, за умови, що вони містяться по різні боки відрізка, який сполучає перші дві знайдені точки. Для реалізації цієї ідеї та пошуку таких точок потрібно побудувати опуклу оболонку до заданої множини точок і знайти шукані точки поміж вершин опуклої оболонки. У цьому випадку дві найвіддаленіші точки можна знайти за лінійний час, оскільки потрібно перевіряти лише протилежні пари точок, які знаходимо теж за лінійний час [27]. Щодо побудови еліпса, то через задані точки здійснюємо афінне перетворення [28]. Якщо еліпс має проходити через чотири точки, то робимо їхнє афінне перетворення на точки квадрата з координатами (1,1), (1, –1), (–1, –1), (–1, 1). Тоді еліпс буде проєкцією кола, описаного навколо квадрата. У варіанті з трьома точками використовуємо правильний трикутник з координатами (1,1), $(-\sqrt{3}/2,-1/2)$, $(\sqrt{3}/2,-1/2)$ і центр (0,0).

Проєкція кола, описаного навколо трикутника, і буде шуканим еліпсом Штейнера [26].

Алгоритм (реалізація)

Структури даних і функції, задіяні для виконання алгоритму, наведено у табл. 2.1.

***Таблиця* 2.1**

`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 із 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

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

Закінчення табл. **2.1**

`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

Побудова еліпса

Еліпс у координатах xy будуємо як проєкцію кола в координатах uv. Нижче наведено математичні формули для перетворення (2.3), (2.4):

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

$$x=\frac{a_0+\mathbf{a}^T\mathbf{u}}{1+\mathbf{c}^T\mathbf{u}},\ \ y=\frac{b_0+\mathbf{b}^T\mathbf{u}}{1+\mathbf{c}^T\mathbf{u}}. \qquad 2.3$$

$$a_0+\mathbf{a}^T\mathbf{u}+0+0+0-x\mathbf{c}^T\mathbf{u}=x,\ \ 0+0+0+0+b_0+\mathbf{b}^T\mathbf{u}-y\mathbf{c}^T\mathbf{u}=y,$$

$$\mathbf{p}=(a_0,a_u,a_v,b_0,b_u,b_v,c_u,c_v)^T,\ \ \mathbf{r}=(x_1,x_2,x_3,x_4,y_1,y_2,y_3,y_4)^T,$$

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

$$\mathbf{A}=\begin{bmatrix}1 & u_1 & v_1 & 0 & 0 & 0 & -x_1u_1 & -x_1v_1\\ 1 & u_2 & v_2 & 0 & 0 & 0 & -x_2u_2 & -x_2v_2\\ 1 & u_3 & v_3 & 0 & 0 & 0 & -x_3u_3 & -x_3v_3\\ 1 & u_4 & v_4 & 0 & 0 & 0 & -x_4u_4 & -x_4v_4\\ 0 & 0 & 0 & 1 & u_1 & v_1 & -y_1u_1 & -y_1v_1\\ 0 & 0 & 0 & 1 & u_2 & v_2 & -y_2u_2 & -y_2v_2\\ 0 & 0 & 0 & 1 & u_3 & v_3 & -y_3u_3 & -y_3v_3\\ 0 & 0 & 0 & 1 & u_4 & v_4 & -y_4u_4 & -y_4v_4\end{bmatrix} \qquad 2.4$$

Видавничо-поліграфічний центр "Київський університет". Версія не для друку

Основну частину вікна становить декартова система координат. Використовуючи формули перетворення [23], можна масштабувати та рухати видиму частину системи паралельно осям координат. На рис. 2.10, 2.11 наведено приклад програмної реалізації алгоритму пошуку найменшого охоплюючого еліпса для ручного введення вхідної множини точок та генерації довільної множини точок.

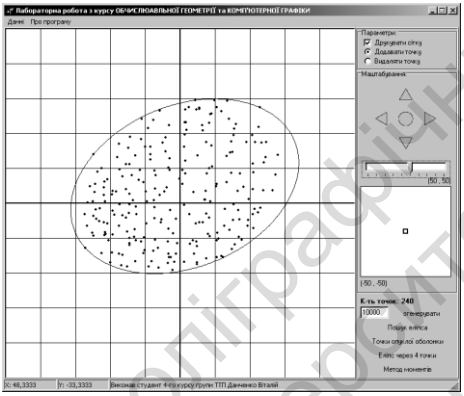


Рис. 2.10. Випадок ручного введення вхідної множини точок (існує можливість видалення точок)

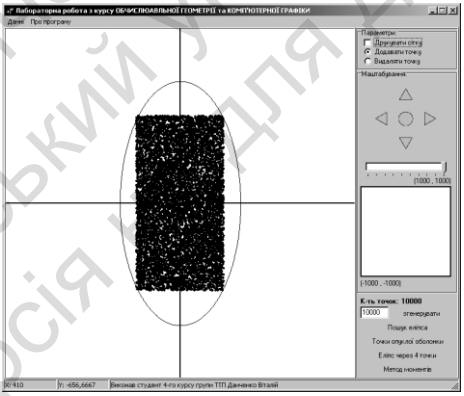


Рис. 2.11. Випадок генерації довільної кількості точок (кількість указується)

Видавничо-поліграфічний центр "Київський університет". Версія для друку

Для отримання результату потрібно натиснути на "Пошук еліпса". Також окремо можна будувати опуклу оболонку. Точки, що входять до оболонки, відображаються іншим кольором. На екрані весь час відображається загальна кількість точок. При змінюванні розмірів форми декартова система координат автоматично масштабується для зручного відображення даних. Є можливість побудувати еліпс методом моментів за лінійний час, але розглядуваний алгоритм не дає оптимального розв'язку.

Оцінка складності алгоритму

Теорема 2.15. *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. Для пошуку точок опуклої оболонки застосуємо метод Грехема [29]. Як відомо, складність цього методу становить *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*). Отже, навіть у найгіршому випадку, коли всі задані точки будуть точками опуклої множини, час виконання алгоритму становитиме *O*(*n*log*n*) (табл. 2.2).

***Таблиця* 2.2**

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)

Висновки. Запропоновано та реалізовано алгоритм побудови еліпса мінімальної площі, описаного навколо заданої на площині множини точок. Алгоритм виконується за оптимальний час *O*(*n*log*n*).

Видавничо-поліграфічний центр "Київський університет" Версія на 1 друк.

2.4.2. Алгоритм побудови з використанням опуклої оболонки (варіант 2)

Алгоритм (основна ідея)

У загальному випадку еліпс задається загальним рівнянням кривої другого порядку $ax^2 + bxy + cy^2 + dx + ey + f = 0$. Однак для еліпса $f = 1$. Отже, маємо п'ять невідомих, і для побудови еліпса потрібно п'ять точок.

Теорема 2.16. *Навколо опуклого п'ятикутника завжди можна описати еліпс такий, що точки п'ятикутника належатимуть еліпсу. Такий еліпс буде єдиним і найменшим охоплюючим еліпсом для цього п'ятикутника.*

Доведення випливає з єдиного розв'язку системи лінійних рівнянь, де замість незалежних змінних підставляємо координати вершин п'ятикутника.

Побудова найменшого охоплюючого еліпса навколо трьох точок. Еліпсів, які описані навколо трьох точок, існує нескінченна кількість, але серед них є лише єдиний із найменшою площею. Для того щоб його знайти, треба розв'язати систему лінійних рівнянь, обмеживши її умовою мінімізації площі. Для випадку, коли центр еліпса лежить на початку координат і його осі симетрії паралельні осям координат, $S_{ел} = 2ab$. Щоб знайти площу еліпса, довільно розташованого відносно системи координат, треба перенести початок координат у центр еліпса, методом невизначених коефіцієнтів розв'язати рівняння перенесення і у такий спосіб отримати ще два рівняння до системи лінійних рівнянь. Можна зробити висновок, що існує алгоритм побудови найменшого еліпса, який проходить через три точки.

Перевірка розташування точки відносно еліпса. Для цього достатньо підставити координати точки в рівняння еліпса. Якщо результат дорівнює нулю, то точка належить еліпсу; якщо результат більше нуля, то точка лежить поза еліпсом; якщо результат менше нуля, то точка лежить усередині еліпса.

Теорема 2.17. *Найменший охоплюючий еліпс, побудований навколо множини точок, збігається з найменшим охоплюючим еліпсом, побудованим навколо опуклої оболонки.*

Видавничий поліграфічний центр "Київський університет"
Вовчок Я.І. ВПЦ КНУ

Доведення. Це твердження доводиться просто. Оскільки 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 ОМЕ для множини *S*.

Алгоритм (покроково)

Етап I. Будуємо опуклу оболонку для *N* заданих точок.

Етап II. Будуємо найменший охоплюючий еліпс, що, у свою чергу, складається з кількох кроків.

Крок 1. Серед отриманих на етапі I точок знаходимо трикутник із максимальною площею. Ці три точки записуємо до множини *Z*.

Крок 2. Для поточної множини *Z* будуємо найменший охоплюючий еліпс. Ця процедура триває фіксований час. Перевіряємо решту точок багатокутника. Якщо всі вони належать еліпсу, то алгоритм завершує роботу, а побудований еліпс є шуканим. Відкидаємо всі точки, що лежать усередині еліпса. Перевірка займає час O(*N*). Інакше 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 п'ять 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 шість точок, то знайдеться точка, яка буде зайвою, тобто для поточного еліпса вона буде лежати всередині) і переходимо на крок 2.

Оцінка складності алгоритму

Теорема 2.18. *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. Будуємо опуклу оболонку $O(n\log n)$. Шукаємо найбільший за площею трикутник $O(n\log n)$. Кроки 2 і 3 повторюються ітеративно. Кількість ітерацій для найгіршого випадку буде *n*, на кожному кроці виконується $O(n)$ операцій, тому загальна складність становитиме $O(n^2)$.

Видавничий центр "Київський університет"
Версія не для друку

2.4.3. Алгоритм побудови з використанням опуклої оболонки (варіант 3)

Серед досліджень можливих методів розв'язання задачі привертають увагу її формулювання (*задача побудови опуклої оболонки та її покриття*) і метод розв'язання, представлені у монографії Л. Ф. Тота [30]. Для покриття опуклої оболонки Тот запропонував ефективний і швидкий алгоритм, однак виникло запитання, що саме в контексті розглядуваної задачі можна вважати опуклою оболонкою, яку необхідно покрити, і як її сформувати. Якщо сформувати опуклу оболонку навколо точок у просторі, то покриття такої оболонки неодмінно утворюватиме "прогалини", тобто круги, які не покривають жодної точки, але входять у покриття опуклої оболонки.

***Алгоритм* (*основна ідея*)**

1. На першому кроці алгоритму будуємо на множині точок опуклу оболонку за допомогою методу Грехема.

2. Далі повертаємось до нашого алгоритму і знаходимо на опуклій оболонці пару точок з максимальною відстанню – це і буде велика вісь еліпса.

3. Знаходимо центр цієї осі. Зсуваємо всі осі й робимо поворот таким чином, щоб положення абсциси й ординат стали (0, 0).

4. Для кожної точки, крім двох, що в результаті з'єднання утворили велику вісь еліпса, знаходимо максимальну відстань від центра осі та з'єднуємо їх. Точка має лежати на еліпсі.

Оцінка складності алгоритму

Теорема 2.19. *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. У нашому випадку швидкість попередньої обробки визначається як швидкість сортування точок за координатою *x* (*O*(*n*log*n*)). Другий етап алгоритму можна виконати з витратами *O*(log*n*) часу, третій і четвертий – за *O*(*n*) часу. Таким чином, загальна складність алгоритму становить *O*(*n*log*n*).

Видавничий поліграфічний центр "Київський університет" Видання для друзів

Алгоритм (реалізація)

Аналіз середовища та мови реалізації. Оптимальним середовищем для реалізації цього проєкту може бути JAVA (JRE) через її кросплатформність. Тоді програмний модуль зможе виконуватись як на комп'ютерах із операційною системою Windows, так і на машинах із операційною системою типу UNIX.

Можливості, програмне й технічне забезпечення реалізації. Для запуску програмного модуля на комп'ютері користувача має бути встановлена віртуальна машина JAVA (JVM) версії не нижче 5.0. Комп'ютер має бути обладнаний не менш ніж 64 Мбайт оперативної пам'яті, мати частоту процесора не нижче 700 мгц і частоту шини не менше 333 мгц. На рис. 2.12 наведено приклад реалізації алгоритму.

Рис. 2.12. Приклад реалізації алгоритму

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

3. ЗАДАЧІ НА ВПИСУВАННЯ

3.1. ПОБУДОВА КОЛА НАЙБІЛЬШОГО РАДІУСА, ВПИСАНОГО В ОПУКЛУ ОБОЛОНКУ (ВМКО)

Нині існують два основні підходи до розв'язання задачі побудови кола найбільшого радіуса, вписаного в опуклу оболонку: з використанням методу подвійного тернарного пошуку [31] і методу стискання сторін [32, 33]. Більш ефективним є другий метод, оскільки він не залежить від вибору системи координат і вимог до точності.

Постановка задачі. *У задану N-вершинну опуклу оболонку на площині вписати коло найбільшого радіуса.*

3.1.1. Алгоритм на основі методу стискання сторін

Розглянемо деякі основні означення.

Означення 3.1. *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 3.2. ***Лівою (правою) бісектрисою ребра** називатимемо бісектрису між поточним ребром і наступним (попереднім) у списку ребер.*

Означення 3.3. ***Висотою** ребра будемо називати відстань від точки перетину його лівої та правої бісектрис до цього ребра.*

Алгоритм (*основна ідея*)

Теорема 3.1. *Якщо у N-вершинному багатокутнику видалити ребро з найменшою висотою, то розв'язок задачі не зміниться.*

Доведення. Будемо зміщувати ребра багатокутника на однакову відстань усередину (рис. 3.1). Вершини будуть пересуватися по бісектрисах кутів багатокутника. Отримаємо новий зменшений багатокутник.

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

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

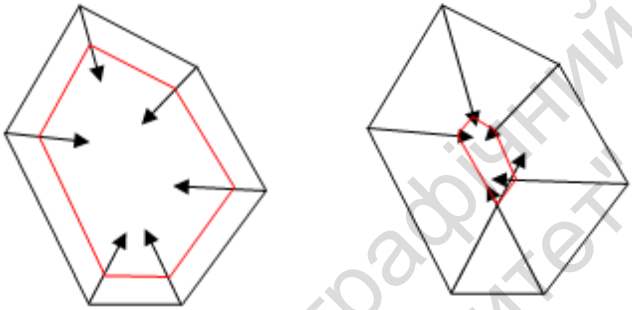


Рис. 3.1. Зміщення ребер багатокутника всередину

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

Припустимо супротивне. Нехай існує інше коло, вписане в початковий багатокутник, з радіусом R', більшим за R. Тоді зменшимо його радіус на h. З аналогічних вищенаведеним міркувань отримане коло буде вписаним у зменшений багатокутник. Однак $r' = R' - h > r = R - h$. Це суперечить нашому припущенню, що ми побудували максимальне коло у зменшеному багатокутнику з радіусом r. Отже, при зміщенні ребер на h усередину багатокутника центр максимального вписаного кола не зміниться, а радіус зменшиться на h. При зміщенні на величину, що дорівнює мінімальній висоті ребра багатокутника,

Видавничо-поліграфічний центр "Київський університет"
Керується ДП ДАУ

відповідне ребро вироджується у точку (рис. 3.2). Якщо ми посунемо ребра назад знову на h, то отримаємо старий багатокутник з одним видаленим ребром. У такий спосіб ми звели задачу побудови максимального кола в багатокутнику з N вершинами до задачі побудови кола для багатокутника з N-1 вершинами. Другий факт, що використовувався в обчисленні: бісектриса новоутвореного кута при видаленні ребра проходить через перетин бісектрис кутів між цим ребром і його сусідами.

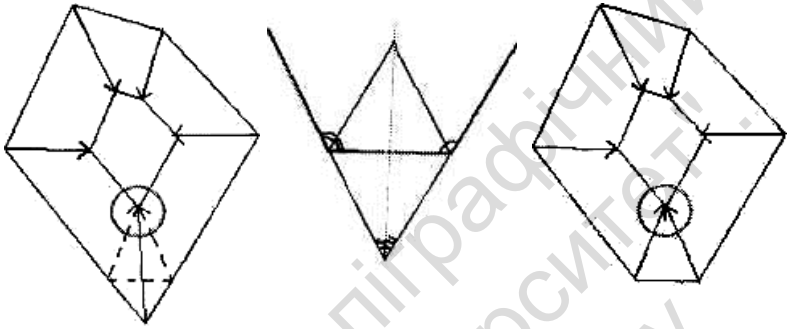


Рис. 3.2. Вироджения ребра в точку

Алгоритм (*покроково*)

1. Нехай побудовано N-вершинну опуклу оболонку на площині одним із відомих методів (напр. методом Грехема).

2. Алгоритм побудови кола:

2.1. Ініціалізація лівих бісектрис для кожного ребра (вони стануть просто бісектрисами кутів багатокутника).

2.2. Обрахування висот для кожного ребра.

2.3. Знаходження ребра з найменшою висотою.

2.4. Видалення ребра з найменшою висотою із 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 п. 2.4, інакше – на п. 2.8.

Видавничо-поліграфічний центр "Київський університет" Версія тільки для друку

2.8. Для трьох ребер, що залишилися, будуємо коло, вписане у трикутник, отриманий із прямих, що визначаються цими ребрами. Побудоване коло і буде розв'язком.

Оцінка складності алгоритму

Теорема 3.2. *Коло максимального радіуса 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 висоти для обох сусідніх ребер. Можна встановити такі часові оцінки:

1) пошук мінімальної висоти за списком ребер за час $O(n)$;

2) перерахування бісектриси, висот сусідів і сусідніх ребер за постійний 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 становить $O(n^2)$, а оцінка пам'яті – $O(n)$.

Алгоритм (реалізація)

На рис. 3.3, 3.4 наведено приклади програмної реалізації для різної кількості точок у двох форматах, відповідно.

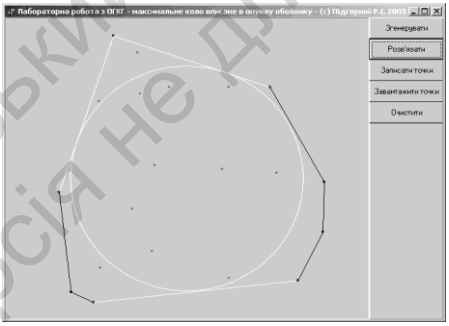


Рис. 3.3. Приклад реалізації для 20 точок

Видавничо-поліграфічний центр "Київський університет"
Версія не для друку

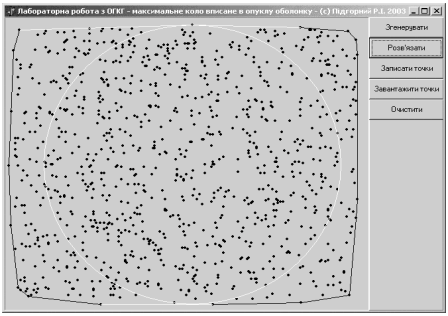


Рис. 3.4. Приклад реалізації для 1000 точок

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

3.1.2. Алгоритм на основі подвійного тернарного пошуку

Розглянемо інший варіант ідеї стискання ребер границі опуклої оболонки для розв'язання задачі вписування в опуклу оболонку кола найбільшого радіуса.

***Алгоритм* (*основна ідея*)**

Почнемо одночасно зміщувати ребра паралельно самим собі всередину (рис. 3.5).

Нехай для зручності швидкість зміщування дорівнює одній координатній одиниці в секунду (тобто час дорівнює відстані в деякому розумінні). У процесі руху сторони багатокутника будуть поступово зникати (перетворюватися на точки). Таким чином, рано чи пізно весь наш багатокутник стиснеться в точку чи відрізок, і цей момент часу t буде шуканим радіусом, а центр лежатиме на отриманому відрізку.

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

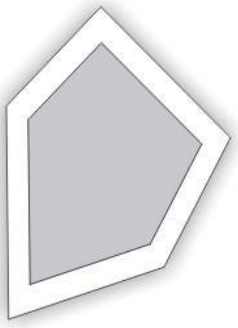


Рис. 3.5. Одночасне зміщення ребер усередину

Дійсно, якщо ми стиснули багатокутник на товщину t по всіх напрямках, то існує точка, що лежить на відстані t від усіх сторін, а для більших відстаней такої точки не існує, оскільки тоді ми б мали можливість стиснути багатокутник принаймні ще один раз.

Знайдемо час для кожної сторони, за який вона стискається в точку. Вершини заданого багатокутника рухаються по бісектрисах відповідних кутів (це випливає з рівності відповідних трикутників). Тоді питання про час, за який сторона остаточно стиснеться, зводиться до питання знаходження висоти H трикутника, у якому відомі сторона L і прилеглі кути α та β. З теореми синусів маємо:

$$H = L\frac{\sin\alpha\sin\beta}{\sin(\alpha+\beta)}. \qquad 3.1$$

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 багатокутник 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 сусідів, продовживши сторони до їхнього

Видавничо-поліграфічний центр "Київський університет". Версія не для друку

взаємного перетину, потім перерахувати довжини двох нових сторін і 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 багатокутник виродиться в точку, а тому результатом (радіусом) буде час, за який зникла остання сторона.

4. Інакше (відсутність паралельності сторін) на деякому етапі залишаться дві сторони, а розв'язком буде час зникання попередньої сторони.

5. Нарешті, знаходимо центр кола. Якщо багатокутник стиснувся в точку, то ця точка і буде його центром; якщо у відрізок – то будь-яка точка із цього відрізка, наприклад будь-який з його кінців; якщо після видалення на останньому кроці сторони залишилися дві непаралельні сторони – то центр перетину бісектрис у трикутнику на передостанньому кроці (знаходимо за відомою формулою) [34]. За умови, що утворилися дві паралельні сторони, на наступному кроці багатокутник стиснеться в точку, яка і буде центром заданого кола.

Оцінка складності алгоритму

Теорема 3.3. *Часова складність задачі знаходження найбільшого вписаного кола в опуклу оболонку за допомогою алгоритму подвійного тернарного пошуку становить* $O(n\log n)$.

Доведення. Як було розглянуто вище, часова складність алгоритму Грехема становить $O(n\log n)$. Розглянемо задачу побудови кола, що вписане в опуклий багатокутник. Потрібний алгоритм у загальному випадку складається із n кроків, на кожному з яких відбувається видалення однієї сторони. Для цього виконуються кілька операцій у структурі даних для визначення мінімуму, на їх виконання необхідно $O(n\log n)$ часу. Оскільки ми зберігаємо лише подвійний список сусідів для кожної вершини, то пам'ять тут лінійна. Отже, загальна складність розглянутого алгоритму становитиме $O(n\log n)$ часу в загальному випадку.

Алгоритм (реалізація)

Особливості програмної реалізації та введення-виведення даних. Програма реалізована мовою C++ із використанням стандартних бібліотек і потоків. У програмній реалізації

Видавничо-поліграфічний центр "Київський університет" Електронна версія друку

застосовуються два основні алгоритми, що описані вище: метод Грехема та метод стискання сторін. Створена програма спочатку приймає на вхід задані точки чи генерує задану кількість точок у певному діапазоні (для вибору кожного з режимів є перемикач). Далі виконується побудова лінійної оболонки за три вищезгадані кроки: знаходимо точку з найменшою *x*-координатою, якщо кілька – то найнижчу; для спрощення реалізації знаходимо також точку з найбільшою *x*-координатою, якщо кілька – то 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 їх (порівнюємо між собою, перенумеровуємо, упорядковуємо відносно початкової точки, з'єднуємо почергово); нарешті виключаємо точки, що не входять до оболонки, перевіряючи почергово для трьох точок утворений кут за допомогою орієнтованої площі.

Далі ми вже маємо деякий опуклий багатокутник і виконується метод стиснення сторін. Алгоритм цього методу описаний вище. Основна ідея – поетапне видалення сторін із найменшим часом, 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. Для нашого конкретного випадку скористаємось відповідною опцією для знаходження центра кола. Після цього програма видає результат користувачу в заданому форматі у файл output.txt.

Опис основних функцій програмної реалізації. Опишемо лише основні функції:

Функція `void convex_hull (vector<pt> & a)` для заданого набору точок виконує побудову опуклої оболонки методом Грехема.

Функція `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 радіуса відбувається вже в основній функції програми. Потім функція `void checkCenter (vector<pt> p)` знаходить центр шуканого кола.

Видавничий інформаційний центр "Київський університет" Розсилка не для продажу

Висновки. Алгоритм вписування кола найбільшого радіуса в опуклу оболонку не має обмежень на кількість точок на вхід (залежить лише від обчислювальної потужності відповідного пристрою), тому його застосовують для розв'язання розглядуваної вище задачі у найзагальнішому випадку. Також він може бути ефективним для розв'язання задач із обчислювальної геометрії. Щоб скористатися створеною програмою, необхідно лише задати точки або перейти в режим автоматичної генерації (потрібно задати бажану кількість точок). Далі програма створить опуклу оболонку, а потім знайде вписане коло максимального радіуса. На виході користувачу буде показано радіус і центр цього кола. Як було доведено вище, час роботи алгоритму в загальному випадку становить $O(n\log n)$.

3.2. ТРИКУТНИК НАЙБІЛЬШОЇ ПЛОЩІ, ВПИСАНИЙ В ОПУКЛУ ОБОЛОНКУ (ВМТО (АО31))

Постановка задачі. *У задану n-вершинну опуклу оболонку на площині вписати трикутник найбільшої площі.*

3.2.1. Алгоритм простого перебирання

***Алгоритм* (*основна ідея*)**

Теорема 3.4. *Вершини трикутника максимальної площі, вписаного в опуклу оболонку, збігаються з вершинами опуклої оболонки.*

Доведення. Нехай задано множину точок S, $\mathrm{con}v(S)$ – опукла оболонка множини S, $CH(S)$ – границя опуклої оболонки множини S. Нехай ABC – трикутник максимальної площі, вписаний в опуклу оболонку $\mathrm{con}v(S)$. Спочатку доведемо, що точки $A, B, C \in CH(S)$.

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

Видавничо-поліграфічний центр "Київський університет" Вересня не для друку

2. Продовжимо сторону AB до перетину із $CH(S)$ і позначимо точку перетину A' (рис. 3.6). 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.

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

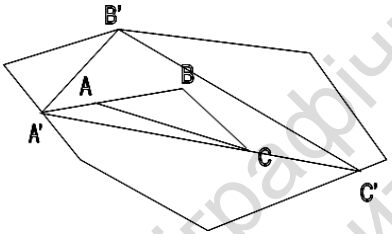


Рис. 3.6. Ілюстрація доведення п. 2 теореми 3.4

1. Припустимо, що точка A не збігається з вершиною опуклої оболонки (рис. 3.7). Якщо ребро, яке містить точку A, не паралельне відрізку BC, то точку A можна перемістити у вершину цього ребра так, щоб збільшилась висота, опущена з точки A на відрізок BC. Площа трикутника $A'BC$ більша за площу трикутника ABC.

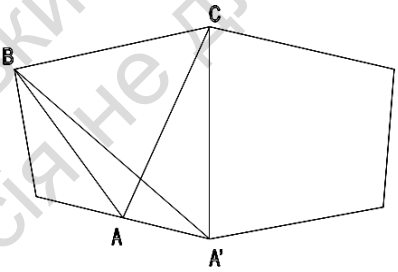


Рис. 3.7. Ілюстрація доведення збігу точок A, B та C з вершинами опуклої оболонки множини S

Видавничо-поліграфічний центр "Київський університет". Версія не для друку

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. Знайдемо трикутник найбільшої площі перебиранням усіх можливих трійок вершин опуклої оболонки.

Оцінка складності алгоритму

Теорема 3.5. *Вписування трикутника у n-вершинну опуклу оболонку запропонованим методом можна виконати за час* $O(n^3)$.

Доведення. Пошук трикутника максимальної площі, вписаного в опуклу оболонку, реалізовано методом перебирання всіх можливих трикутників. Це вимагає $O(k^3)$ операцій, де k – кількість вершин опуклої оболонки множини S. Таким чином, складність алгоритму в найгіршому випадку становитиме $O(n^3)$. Однак у середньому алгоритм працює швидше, оскільки в більшості випадків $k << n$.

Алгоритм (*реалізація*)

Основні функції програми

1. `list *opukla_obolonka(list *p)` – функція отримує на вході список точок множини S, на виході видає впорядкований список точок, який утворює $CH(S)$.

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

3. `void drawOO(list *l)` – функція зображує опуклу оболонку множини S.

4. `void drawTr(list *l)` – функція зображує трикутник максимальної площі, вписаний в опуклу оболонку.

Видавничо-поліграфічний центр "Київський університет" Вдома не для друку

У програмі передбачена можливість як ручного, так і автоматичного введення точок. Автоматичний спосіб дозволяє ввести 10000 точок.

Висновки. Недоліком алгоритму є велика часова складність його роботи. Однак це залишає можливість для підвищення ефективності програми. Один із можливих шляхів оптимізації – використання властивості опуклості 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 Грехема. Це дозволить пришвидшити побудову опуклої оболонки.

3.2.2. Алгоритм бінарного пошуку вершин максимального трикутника

Ураховуючи теорему 3.4, можна спробувати поліпшити оцінку складності попереднього алгоритму до $O(n\log n)$ завдяки зображенню вершин опуклої оболонки у вигляді їхнього відсортованого за полярним кутом списку.

***Алгоритм* (*основна ідея*)**

Нехай маємо n-вершинну опуклу оболонку $CH(S(n))$. Зобразимо її вершини у вигляді центрально-впорядкованого списку за полярним кутом відносно центроїда (т. O)

$CH(S) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ (рис. 3.8).

За центроїд можемо взяти перші вершини списку 1, 2, 3.

Пошук трикутника розіб'ємо на два етапи. На першому етапі знаходимо всі локальні максимуми, тобто такі трикутники, для яких фіксуємо будь-які дві їхні вершини, а третю не можемо зсунути за списком ліворуч чи праворуч для збільшення площі. На другому етапі серед знайдених на першому етапі трикутників обираємо трикутник з найбільшою площею.

Видавничо-поліграфічний центр "Київський університет". Відповідає для друку.

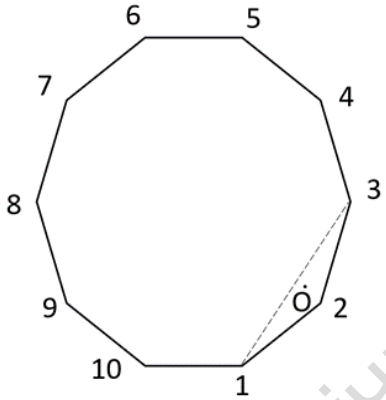


Рис. 3.8. Опукла оболонка $CH(S) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

Перший етап

1. Нехай маємо фіксовану вершину, наприклад вершину 1 (рис. 3.9).

2. Обираємо інші дві вершини: $[(n\text{-}1)/4]$ і $3[(n\text{-}1)/4]$. Ці вершини є серединами половини вершин опуклої оболонки, що залишилися.

3. Переміщуємо точки i та j ліворуч і праворуч від фіксованої точки 1 (рис. 3.9).

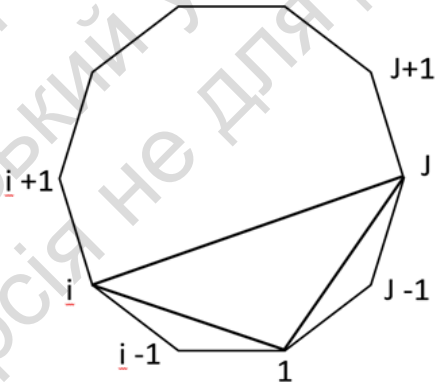


Рис. 3.9. Пошук максимумів на $CH(S)$

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

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

Наприклад, ми вибрали перші дві вершини довільно. Нехай це будуть 1 і 2. Тоді з вершин на рис. 3.9 шукаємо таку, щоб відстань від відрізка 1–2 була найбільшою. Цю відстань можна визначити, якщо відомі координати точок (формула 3.2).

$$d(P,\mathbf{L}) = \frac{(y_0 - y_1)x + (x_1 - x_0)y + (x_0 y_1 - x_1 y_0)}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}}, \qquad 3.2$$

де $d(P, L)$ – відстань від точки P до відрізка L; x_0, y_0, x_1, y_1 – координати початку і кінця відрізка; x, y – координати точки P (рис. 3.10).

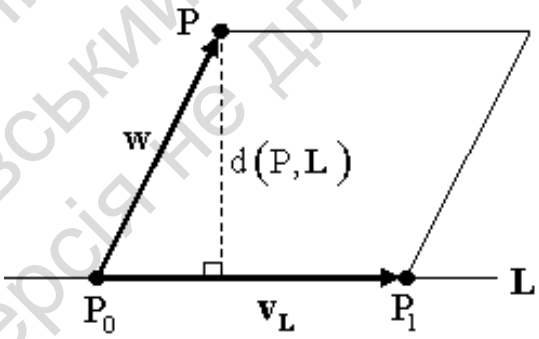


Рис. 3.10. Ілюстрація до формули 3.2

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

Переміщуємо точки одночасно, тоді наступні точки *i* та *j* будуть такими, як зображено на рис. 3.9.

1. Точки *i* та *j* (обидві) рухаємо праворуч, тоді наступна ітерація матиме вигляд:

[(*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 *i* та *j* (відповідно праворуч і ліворуч). На першому кроці ці обмеження мають значення *pri* = [(*n* – 1)/2] та *pli* = *plj* = 1.

2. Точку *i* рухаємо праворуч, а точку *j* – ліворуч, тоді наступна ітерація матиме вигляд:

[(*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* (залежно від випадку) – попередні обмеження руху для точок *i* та *j* (відповідно праворуч і ліворуч). На першому кроці ці обмеження мають значення *pri* = [(*n*-1)/2] та *pli* = *plj* =1.

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

Другий етап. На цьому етапі нам необхідно знайти всі локальні максимуми. Для цього беремо по черзі всі *n* точок опуклої оболонки і для кожної виконуємо процедуру пошуку максимального трикутника. Їхня кількість становитиме не більше ніж *n*/3, однак лише в тому випадку, коли всі вершини границі опуклої оболонки лежатимуть на колі або оболонка нагадуватиме коло (рівносторонній багатокутник). У результаті отримаємо для всіх вершин *n* трикутників. Поміж них будуть трикутники двох типів: максимальні й такі, для яких будь-які дві їхні вершини фіксовані, а третю ми не можемо зсунути за списком ліворуч чи праворуч для збільшення площі. Чим більше оболонка сплюснута за формою, тим менше в ній локальних максимумів. Щоб знайти їх усі, достатньо обрати якусь вершину і виконати пошук локальних максимумів для основ із цією вершиною та всіма іншими, яких буде *n*-1. Цього буде достатньо, оскільки основа пройде наближено всі кути нахилу

Видавничий центр "Київський університет"
Вилучено електронну версію

однієї із трьох сторін локального максимуму, який знаходимо за допомогою процедури пошуку.

Оцінка складності алгоритму

Теорема 3.6. *Вписування трикутника у n-вершинну опуклу оболонку запропонованим методом можна виконати за час* $O(n\log n)$.

Доведення. Пошук точки відбувається зі складністю $O(\log n)$. Оскільки точки впорядковані, то пошук дихотомічний, з деякою специфікою, що базується на такому: якщо до однієї з точок відстань більша, ніж до іншої (сусідньої), то точка з найбільшою відстанню буде лежати праворуч від першої. Знайшовши таку точку, фіксуємо її. Далі фіксуємо точку або 1, або 2. Нехай 2, тоді для точок 2 і 7 виконуємо ту саму процедуру. Процедура виконуватиметься не більше C разів, тобто маємо складність $C*O(\log n) = O(\log n)$. Отже, загальна складність алгоритму становить

$$O(n\log n) + O(\log n)*(n\text{-}1) = O(n\log n)!.$$

3.3. ПРЯМОКУТНИК НАЙБІЛЬШОЇ ПЛОЩІ, ВПИСАНИЙ В ОПУКЛУ ОБОЛОНКУ (ВМПО (АО41))

Проблема пошуку прямокутника найбільшої площі, вписаного в опуклу оболонку, часто виникає за спроби виправлення спотворення від зміщеного проєктора. Щоб отримати найкращий ортогональний прямокутник на проєкційній поверхні, треба вибрати його в межах проєктованої площі. Моделюючи проєктовану область як опуклий багатокутник, можна розв'язати проблему, обчисливши найбільший вписаний ортогональний прямокутник. Утім, якщо кілька проєкторів об'єднані, щоб сформувати великий екран, то багатокутник, який є результатом об'єднання всіх проєктованих областей, може не бути опуклим.

Ми розглядатимемо саме вписування найбільшого прямокутника в опуклу оболонку на площині. Існує кілька алгоритмів, які дозволяють розв'язати цю задачу з різною складністю, від

Видавничо-поліграфічний центр "Київський університет" Вступ до ІТ для друку

повного перебору до $O(\log n)$ [5, 29, 35-42]. Розглянемо деякі з них, які поряд з високою ефективністю дозволяють нескладну реалізацію. Зокрема, розглянемо алгоритми із часовою складністю $O(n\log n)$ та із часом $O(\log n)$ [38]. Варто звернути увагу також на алгоритм [39] із часом $O(n)$ і алгоритм [41] із часом $O(\log^2 n)$.

Постановка задачі. *У задану n-вершинну опуклу оболонку на площині вписати прямокутник найбільшої площі.*

3.3.1. Алгоритм бінарного пошуку вершин максимального трикутника

У цьому алгоритмі запропоновано ідею розв'язання задачі за допомогою стратегії "розділяй та володарюй" і діаграми Вороного [43].

Алгоритм (основна ідея)

Нехай маємо опуклу оболонку для заданої множини S із n точок, задану границею $CH(S) = \{P_1, P_2, \ldots, P_n\}$. Оскільки прямокутник фактично задається однією точкою на границі $CH(S)$, то пересуваємо цю точку по її ребрах згідно з наведеним нижче алгоритмом.

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, то враховуємо й точку, де досягається максимум, як можливого кандидата. Головна ідея полягає в тому, що ліва точка шуканого прямокутника буде або одним із кінців якогось із проміжків, або належатиме проміжку, причому буде максимумом функції площі на ньому (рис. 3.11).

4. Двійковим пошуком знаходимо інші вершини прямокутника.

5. Порівнюємо його з найбільшим на поточний момент прямокутником.

Видавничо-поліграфічний центр "Київський університет" Безоплатно для друку

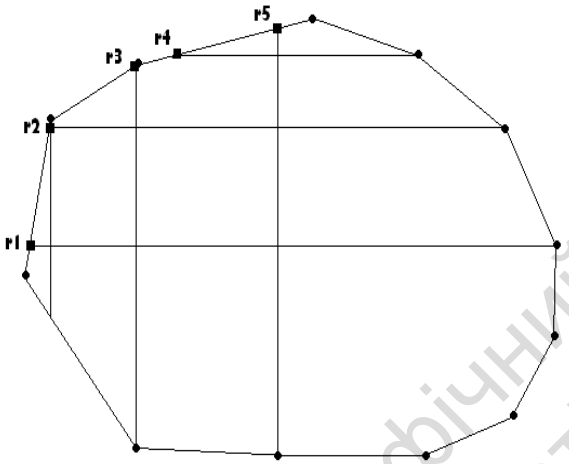


Рис. 3.11. Пошук екстремумів площ прямокутників

6. Відображуємо багатокутник відносно осей координат, кожного разу запускаючи алгоритм для проходження не лише справа наліво верхньою ланкою, а й усіма можливими способами.

Оцінка складності алгоритму

Теорема 3.7. *Вписування прямокутника у n-вершинну опуклу оболонку запропонованим методом можна виконати за час* $O(n\log n)$.

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

Алгоритм (*реалізація*)

Основні функції програми (рис. 3.12): функція пошуку опуклої оболонки; функція пошуку вписаного прямокутника максимальної площі.

Введення даних можна робити двома способами: вручну в поле або згенерувати за допомогою програми. Виведення здійснюється на екран, причому значення площі записується в окреме поле.

Видавничо-поліграфічний центр "Київський університет". Версія не для друку

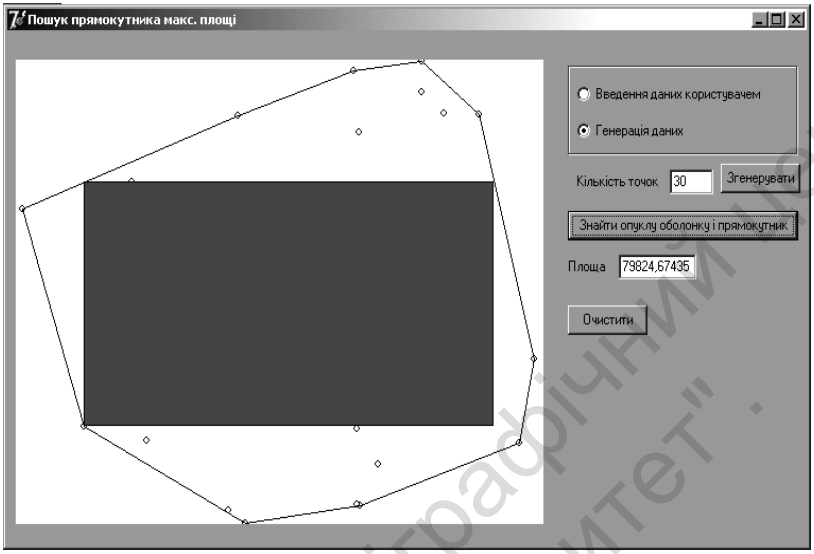


Рис. 3.12. Приклад реалізації алгоритму

Висновки. Запропоновано оптимальний алгоритм вписування прямокутника максимальної площі в опуклу оболонку на площині за оптимальний час $O(n\log n)$. Прискорити час можна лише на константу, сумістивши деякі кроки алгоритму. Інтерфейс виконано максимально просто для уникнення зайвих елементів і погіршення інтуїтивного розуміння програми. Більшість нюансів реалізації виникли на другому етапі, оскільки на ньому розглядаються чотири випадки розміщення прямокутника, для кожного з яких будується множина точок розбиття на інтервали. Оскільки ймовірно, що два різні випадки розглядають той самий прямокутник, то можна прискорити алгоритм, виключивши один із них, але, знову ж таки, прискорення буде лише на константу.

3.3.2. Алгоритм на основі методу січних прямих

У першій частині алгоритму знаходимо прямокутник найбільшої площі, сторони якого паралельні відповідним осям координат,

Видавничо-поліграфічний центр "Київський університет" Версія тестового друку

у другій – шуканий прямокутник. Тут використаємо одну із властивостей опуклого багатокутника, яку розглянемо нижче.

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 із центром у початку координат). Якщо зафіксувати напрямок $\alpha + \delta$, де $\delta \to 0$, то оптимальний для такого напрямку прямокутник отримаємо з початкового за допомогою зміщення вершин по ребрах багатокутника в напрямку зміни кута.

Алгоритм (*покроково*)

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. Максимальний вписаний у $CH(S)$ прямокутник знайдемо методом січних прямих. *Січними прямими* називатимемо прямі, паралельні осям OX та OY.

1. Оскільки у вписаного прямокутника принаймні три вершини належать сторонам багатокутника, то можливі два варіанти розташування прямокутника: точки верхньої сторони прямокутника належать сторонам багатокутника або точки нижньої сторони прямокутника належать сторонам багатокутника (варіанти не є взаємно виключними).

2. Опишемо пошук оптимального прямокутника першого типу (прямокутник другого типу шукатимемо аналогічно). Зауважимо, що хоча б один із двох прямокутників обов'язково існує.

3. Січну пряму відносно OX починаємо переміщувати з найвищої точки багатокутника. Січна пряма визначає чотири точки (рис. 3.13).

4. Розбиваємо рух січної прямої на етапи, на кожному з яких жодна із чотирьох точок не змінює ребра. Пошук максимального прямокутника LR на кожному етапі є мінімаксною задачею, розв'язок якої легко знаходимо за координатами відповідних ребер (можна отримати аналітичну формулу для знаходження оптимального положення січної прямої на етапі), а саме: за відомими ребрами a_1, a_2, a_3, a_4 знаходимо максимальний прямокутник (рис. 3.14).

Видавничий центр "Київський університет"

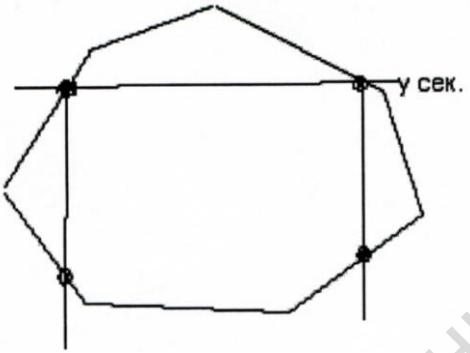


Рис. 3.13. Пошук максимального прямокутника за допомогою січних

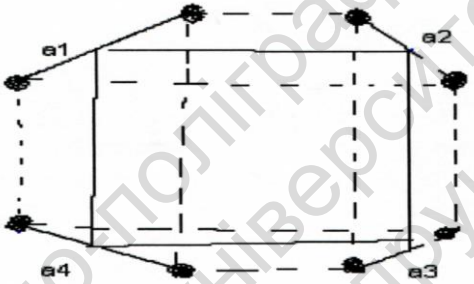


Рис. 3.14. Максимальний прямокутник за ребрами a_1, a_2, a_3, a_4

5. За результат беремо найбільший із побудованих на попередніх етапах прямокутників.

6. Рух січної прямої продовжуємо до тих пір, доки по її положенню можна визначити згадані вище чотири точки, тобто поки обидва промені, перпендикулярні до *OX* і проведені через точки, у яких січна пряма перетинає багатокутник, перетинають деякі ребра нашого багатокутника.

7. Для знаходження найбільшого прямокутника січну пряму необхідно переміщувати знизу вгору.

Видавничо-поліграфічний центр "Київський університет" Вологне для друку

8. Отже, після другого етапу ми маємо максимальний прямокутник P для напрямку $\alpha = 0$. Цей етап є ітераційною процедурою. Опишемо її.

8.1. На нульовій ітерації $\alpha_0 = \alpha_1 = 0$, оптимальним є прямокутник P.

8.2. Нехай на останній ітерації ми знайшли оптимальний прямокутник для діапазону напрямків $[\alpha_0, \alpha_1]$. Нехай точка O – центр цього прямокутника, а точки A, B, C – його вершини, що належать сторонам багатокутника (рис. 3.15).

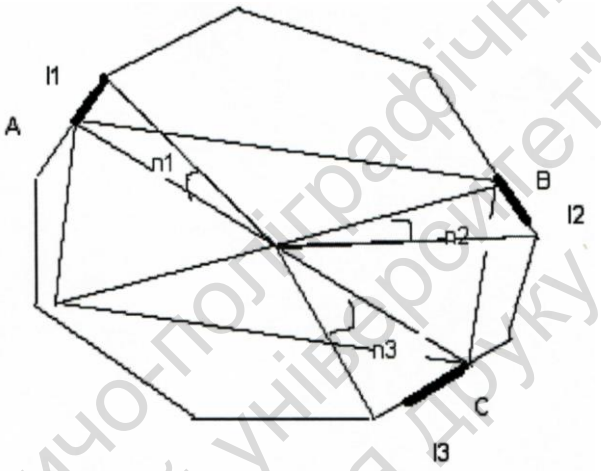


Рис. 3.15. Знаходження максимального прямокутника P для напрямку α за ітераційною процедурою

8.3. Знаходимо для точок A, B та C мінімальні кути при повороті проти годинникової стрілки відносно точки O, на які вони змістяться у вершину багатокутника.

8.4. Мінімальний із трьох кутів позначимо $\delta = \min(n_1, n_2, n_3)$.

8.5. Для наступної ітерації діапазон буде $[\alpha_l, \alpha_1 + \delta]$. Знаходимо максимальний прямокутник для діапазону $[\alpha_l, \alpha_1 + \delta]$.

8.6. Позначимо відрізки, по яких рухаються точки A, B, C, як 11, 12, 13, відповідно.

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

8.7. Із властивості опуклого багатокутника, яку було наведено у п. 8.6, випливає, що три вершини максимального прямокутника для зазначеного напрямку належатимуть ребрам 11, 12, 13, відповідно.

8.8. Можна вивести формули, за якими за координатами точок *A*, *B*, *C* і відрізків 11, 12, 13 знайдемо максимальний прямокутник для поточного напрямку. Вигляд формул залежатиме від кутів нахилу відрізків.

Зауваження. Може статися так, що четверта вершина знайденого прямокутника для діапазону $[\alpha_0, \alpha_{II})$ буде виходити за межі багатокутника. У цьому випадку максимальний прямокутник буде знайдено для напрямків від $(\alpha_0 + 180)$ до $(\alpha_1 + 180)$.

8.9. Для швидкої перевірки належності четвертої вершини багатокутнику після закінчення кожної ітерації зберігатимемо ребро багатокутника найближче до цієї вершини. На наступній ітерації найближче ребро або не зміниться, або перейде в наступне за годинниковою стрілкою.

8.10. Ітерації закінчуються, коли якась із вершин зробить повний обхід, тобто будуть переглянуті всі можливі напрямки від 0 до 360°.

9. Результатом роботи третього етапу буде найбільший з усіх прямокутників, які ми будували на ітераціях.

Оцінка складності алгоритму

Теорема 3.8. *Вписування прямокутника у n-вершинну опуклу оболонку методом січних прямих можна виконати за час* $O(n \log n)$.

Доведення. На першому етапі виконуємо два рази алгоритм руху січної прямої. Кожен із кроків, на які ми розбиваємо рух січної прямої, обмежується її положенням, при якому одна із чотирьох точок переходить у вершину багатокутника. 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 постійний час, тому не важко підрахувати, що складність алгоритму руху січної прямої становить $O(n)$. Отже, і весь перший етап має складність $O(n)$. На другому етапі для вершини прямокутника, яка не належить стороні багатокутника, знаходимо найближче ребро за час $O(n)$.

Видавничо-поліграфічний центр "Київський університет" Роздруковано для друку

Складність кожної ітерації становить $O(1)$, а складність переходу від ітерації до ітерації – $O(1)$.

Нам залишилось оцінити найбільшу можливу кількість ітерацій на цьому етапі. Оскільки границя кожного етапу визначається ситуацією, коли одна із 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 становить $O(n)$. Підсумовуючи складності всіх трьох етапів, бачимо, що складність усього алгоритму становить $O(n \log n)$.

Алгоритм (*реалізація*)

Наведений алгоритм реалізовано у 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 у неї. Точки можна вводити мишкою довільно (максимальна кількість точок 30000). На рис. 3.16 показано приклад роботи програми.

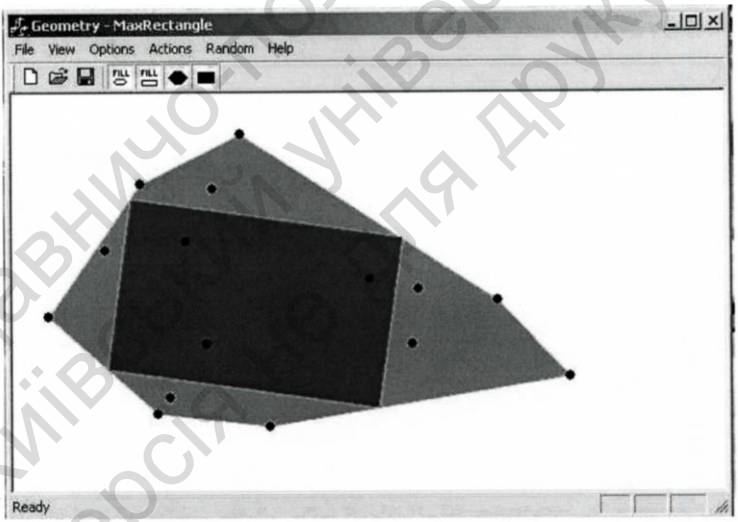


Рис. 3.16. Приклад реалізації алгоритму

Видавничо-поліграфічний центр "Київський університет" Версія для друку

3.3.3. Алгоритм на основі методу prune-and-search для фіксованих точок

Розглянемо випадок, коли найбільший прямокутник має дві або три вершини на границі опуклої оболонки. Для розв'язання задачі можна скористатися алгоритмами Alt, Hsu та Snoeyink [38]. У цьому випадку задача зводиться до пошуку фіксованої точки композиції із двох або трьох функцій. В основі алгоритму лежить ідея, що максимальна площа прямокутника визначається у фіксованій точці та можна використати метод prune-and-search для нерухомої точки двох чи трьох функцій. Обидва методи мають дихотомію пошуку.

Алгоритм (основна ідея)

Розглянемо границю опуклої оболонки $CH(S)$ як деякий багатокутник P на площині. Розіб'ємо P на чотири ланцюги, розрізавши його у крайніх напрямках, як у напрямку x, так і y, як це показано на рис. 3.17.

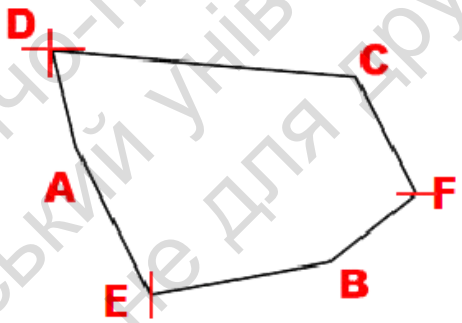


Рис. 3.17. Розбиття багатокутника P на ланцюги

Визначимо обхід ланцюгів 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 цій вершині мала унікальну точку

Видавничо-поліграфічний центр "Київський університет" Россія не для друку

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 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 двох вершин на границі *P*. Якщо найбільший прямокутник *LR* має рівно два кути на *P* (рис. 3.18), то ці кути будуть діагонально протилежними. Якби кути були суміжними, то протилежний кінець *LR* узагалі не торкався би границі *P* і його можна було б відсунути, утворивши більший прямокутник.

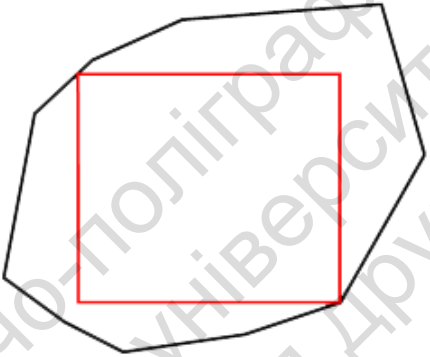


Рис. 3.18. Дві вершини прямокутника на границі багатокутника

Аналогічно, якщо один кут *LR* міститься на границі *P*, то два кінці *LR* можуть вільно рухатися і утворювати більший прямокутник. Діагонально протилежні кути *LR* мають контактувати із двома вершинами *P* або однією вершиною та одним ребром. Згідно з лемою 3.1, якщо жоден із двох кутів *LR* не контактує з вершиною, то можна знайти більший прямокутник.

Лема 3.1 [44]. *Нехай e_1 та e_2 – відрізки ліній, що не перетинаються. Розглянемо набір прямокутників, сторони яких паралельні осям декартової системи координат, які мають*

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

діагонально протилежні кути на e_1 та e_2. У цьому наборі є прямокутник із найбільшою площею із принаймні одним кутом у кінцевій точці e_1 або e_2.

Доведення. У роботі [44] стверджується, що функція площі із двома параметрами може бути показана як поверхня сідла, тому максимум лежить на границі. Більш інтуїтивно, якщо задовольняється припущення загального положення, то два рядки, що містять відрізки, перетинатимуться з одного боку відрізків. Для будь-якого прямокутника з кутами, які ще не містяться на вершинах, можна буде зсунути його у зворотному напрямку, а потім розгорнути, оскільки один із кутів більше не буде контактувати з одним краєм багатокутника. Це видно на рис. 3.19.

Рис. 3.19. Двохкутовий прямокутник, який не міститься на жодній вершині *P*, можна зробити більшим

Випадок трьох вершин на границі *P*. Найбільший за площею прямокутник *LR* може мати три вершини на границі *P* (рис. 3.20).

Якщо багатокутник перебуває в загальному положенні, то максимум два кути *LR* будуть у вершинах *P*, оскільки сусідні кути не можуть розміщатися на вершинах. Можливо, жоден із кутів не розміщений 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 всі вершини *LR* лежать на границі *P*. Тоді процедура пошуку буде аналогічною розглядуваному випадку.

Видавничо-поліграфічний центр "Київський університет" Видання не для друку

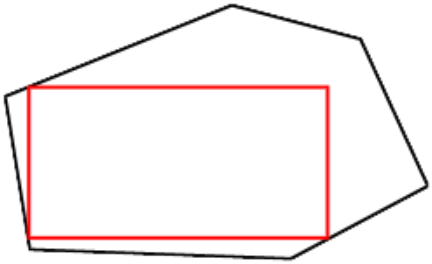


Рис. 3.20. Випадок трьох вершин прямокутника *LR* на границі *P*

Алгоритм (покроково)

Алгоритм пошуку прямокутника найбільшої площі (*LR*) детально описаний у роботі [38]. Суть його зводиться до розгляду випадків розташування вершин *LR* на границі *P*. Якщо дві вершини *LR* лежать на *P*, то вони будуть належати ланцюгам *A* і *C* або ланцюгам *B* і *D*. Ми отримаємо найбільший прямокутник, знайшовши два максимуми. Робимо те саме для випадку із 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, як ми знайшли всі можливі варіанти для двох і трьох кутів, їхній максимум буде найбільшим ортогональним прямокутником. Розглянемо більш детально зазначені випадки.

Випадок двох вершин, що лежать на P

Лема 3.2. *Припустимо, що прямокутник максимальної площі, вписаний у P, має точно дві вершини на границі P, а саме* $a \in A$, $c \in C$ (рис. 3.21). *Звідси випливає, що P має паралельні дотичні до a і c із нахилом m, де* $m = -m_{ac}$ (рис. 3.22).

Доведення. Якщо зафіксувати початок на *a*, то можна отримати функцію сталої площі *F* через *c* і записати її як $F = xy$, де *x* та *y* – відповідні координати *c*. Записавши $F = xy$, отримуємо *m* (з нахилом дотичних до *c*), узявши похідну $dy/dx = -F/x^2$. Нахил діагоналі від *a* до *c* m_{ac} становить y/x або F/x^2, коли ми замінюємо *y* на F/x. Багатокутник повинен мати дотичну до *c* із нахилом *m*, інакше ми можемо перемістити *c* на невелику відстань, щоб отримати прямокутник площею більше *F*. Якщо багатокутник має

Видавничо-поліграфічний центр "Київський університет" Власність для друку

дотичну до c із нахилом m, то перенесення c тільки зменшить площу прямокутника. Такі самі результати ми отримаємо, спочатку зафіксувавши c і застосувавши функцію сталої площі через a. Поєднавши ці результати, одержимо, що дотичні до a і c паралельні (рис. 3.22). Дотична з нахилом 2/3 належить як функції площі, так і c, нахил діагоналі становить 2/3.

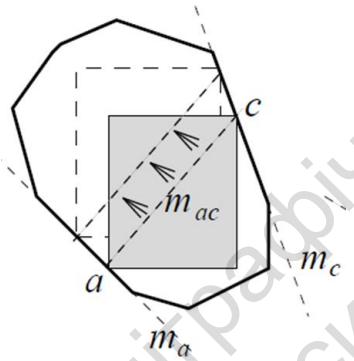


Рис. 3.21. Випадок двох вершин прямокутника LR на границі P_c

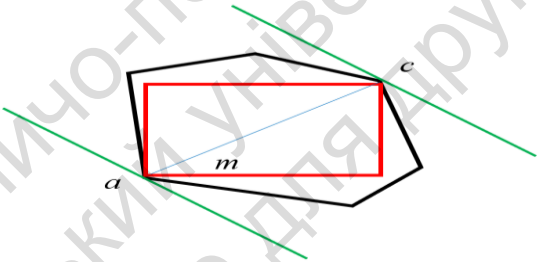


Рис. 3.22. Паралельні дотичні до багатокутника і функція площі з нахилом $-m_{ac}$

З попередньої леми ми можемо визначити дві функції f та g, одна з яких буде зростаючою, а інша спадною, таким чином, щоб фіксована точка їхньої композиції забезпечила найбільший прямокутник. Для точки $a \in A$ нехай $f(a)$ буде точкою на C такою, щоб дотичні на a та $f(a)$ були паралельними. Для точки a

Видавничо-поліграфічний центр "Київський університет". Версія не для друку

точка $c \in C$ є дотичною, яка має нахил m. Нехай $g(c)$ буде точкою на A такою, щоб лінія від $g(c)$ до c мала нахил $-m$. При обході A і C проти годинникової стрілки f зростає, а g спадає так, що їхня композиція має фіксовану точку. У цій точці $a = g(f(a))$ так, що $c = f(a)$ і $g(c) = a$. Це означає, що дотичні a та c паралельні, отже, нахил діагоналі від'ємний відносно нахилу дотичної до c. Ця умова задовольняє попередню лему, тобто ми отримуємо прямокутник збільшеної площі.

Метод 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 A і C та оберемо одне специфічне значення для дотичної з усіх можливих. Порівнюючи $f(a)$ із c та a із $g(c)$, ми можемо виключити половину A або C. Наприклад, якщо $f(a)$ міститься після c, а $g(c)$ – перед a, то ми можемо виключити все після a, оскільки знаємо, що фіксована точка не може бути розміщена на тій частині A. Функція f, оцінена для будь-якої точки на виключеному ланцюзі, завжди буде лежати вище c на C. Оцінка g у цій новій точці буде завжди лежати нижче $g(c)$ на A, яка сама розташована під a, тому ми ніколи не повернемось до нашої початкової точки. Усі можливі випадки, які дозволяють виключити верхню або нижню частину A або C, зображені на рис. 3.23.

Цей процес може бути виконаний без оцінки $f(a)$ або g(c). Замість того, щоб порівняти $f(a)$ і c, нам потрібно порівняти дотичні до a і c, а щоб порівняти $g(c)$ і a, ми порівняємо нахил до c із нахилом діагоналі з a до c. Коли частина ланцюга виключена, 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 з лемою 3.1 можна вважати, що ми маємо чотири варіанти випадку з ребром і вершиною. Тоді нахил дотичної – це нахил ребра, а нам лише

Видавничо-поліграфічний центр "Київський університет" Вебсайт не для друку

треба знайти лінію, яка проходить через вершину, що перетинає ребро. Це і буде діагональ найбільшого прямокутника.

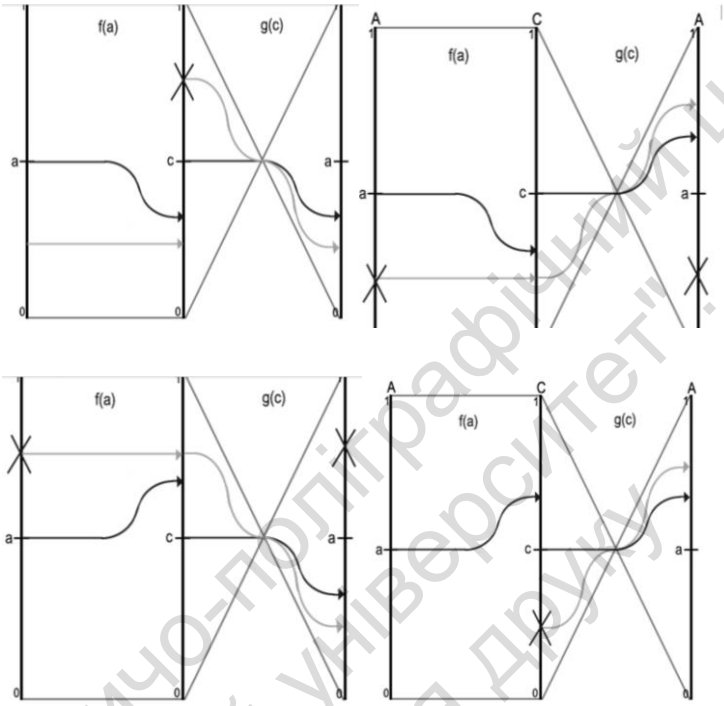


Рис. 3.23. Випадки, які дозволяють виключити верхню або нижню частину *A* або *C*

3. Потім необхідно перевірити випадки, щоб пересвідчитись, що прямокутники вписані. Для цього необхідно пересвідчитися, що дві вершини, які лишилися, лежать усередині багатокутника *P*. Це можна зробити за час $O(\log n)$, виконавши двійковий пошук по ланцюгах *B* і *D* (для нижньої та верхньої вершин, відповідно), щоб знайти ребро, яке має ту саму координату *x* або *y*.

4. Потім за час $O(1)$ перевіримо, чи містяться ці точки всередині.

5. Найбільший прямокутник з вершинами на *A* і *C* є максимальним з усіх можливих прямокутників.

Видавничо-поліграфічний центр "Київський університет" Вибрана не для друку

6. Тепер, коли в нас є найбільший прямокутник з вершинами на A і C, ми можемо повторити ті самі кроки, щоб знайти найбільший прямокутник з вершинами на B і D.

7. Щоб знайти найбільший прямокутник у загальному випадку, нам необхідно знайти найбільший прямокутник із трьома вершинами на P і взяти максимум. На рис. 3.24 зображено весь процес пошуку прямокутника із двома вершинами на A і C.

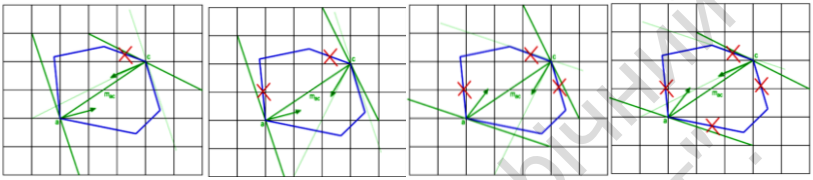


Рис. 3.24. Кроки пошуку прямокутника із двома вершинами на A і C

Три вершини на границі P

Лема 3.3. *Припустимо, що максимальний прямокутник вписано в P і три його вершини лежать на границі P. Тоді існують дотичні a в A, b у B та c у C із нахилом* $m_a < 0$, $m_b > 0$, $m_c < 0$, *які задовольняють умови* $-m_a >= m >= -m_c$ *і* $m = -m_a * (m_c - m_b)/(m_a - m_b)$, *відповідно, де* $m = m_{ac}$ *у випадку, коли найбільший прямокутник* (*LR*) *має максимальну площу.*

Доведення [38]. Для того щоб отримати рівняння для m, площу обчислюємо як функцію від a, b і c. Узявши похідну і прирівнявши її до нуля, отримаємо максимум і мінімум. Те саме можна сказати і тоді, коли точки B і C збігаються. Якщо ми виключимо ці два випадки, то залишиться лише єдиний нуль, який відповідає максимуму. Зрештою це приводить до рівняння для m. З рис. 3.25 видно, що треба зробити для того, щоб $-m_a$ був більшим за $-m_c$. Якщо це не так, то прямокутник може ковзати вздовж b, після чого буде розширений.

Метод prune-and-search для знаходження фіксованої точки трьох функцій. Використовуючи лему 3.3, необхідно визначити три функції такі, щоб їхня фіксована точка виявила найбільший прямокутник, три вершини якого лежать на P. Для

Видавничий поліграфічний центр "Київський університет" Версія для друку

цього необхідно, щоб усі три функції були спадними при обході P проти годинникової стрілки. Очевидно, що ми можемо визначити $f(a)$ як точку на B із тією самою координатою y, що і на A. Аналогічно можемо визначити $g(b)$ – точку на C із такою самою координатою x, як b на B. Нарешті, нехай $h(c)$ – точка на A така, що $m_{ac} = m$ (див. лему 3.3). Як і у випадку із двома вершинами, нам не потрібно безпосередньо визначати f, g або h. Ми отримаємо $f(a)$ і $g(b)$ виходячи з того, що маємо координату y на a і координату x на b. Щоб оцінити $h(c)$, порівняємо нахил лінії від a до c із m, який знаходимо з відповідного рівняння (лема 3.3). На рис. 3.26 показано, що в більшості випадків (шість із восьми) ми можемо вилучити половину одного із трьох ланцюгів так, як робили у випадку із двома функціями.

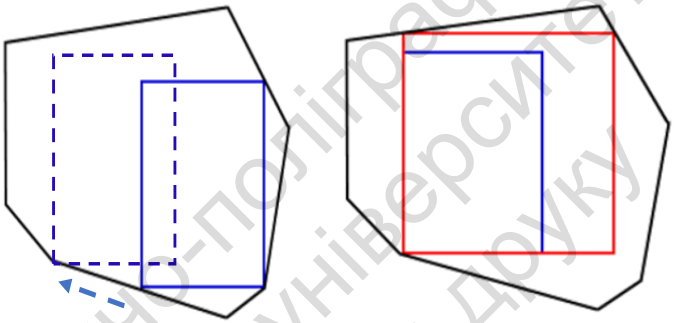


Рис. 3.25. Прямокутник може бути збільшений, якщо $-m_a < -m_c$

Проте в інших випадках, зображених на рис. 3.26, ми можемо лише попередньо вилучити половину всіх ланцюгів. Без попередніх вилучень одержимо випадок двох функцій. Коли ж є вилучення, повертаємось попередній режим. У цьому режимі ми циклічно переміщуємо a, b і c. У результаті нічого не змінюється і ми продовжуємо попереднє вилучення або можемо назавжди вилучити частину одного ланцюга. Коли це відбувається, ми можемо скасувати інші попередні вилучення та повернутися до звичайного режиму. Після того, як кожен ланцюг був зведений до одного ребра чи вершини, ми можемо знайти постійних кандидатів для найбільшого прямокутника за постійний час.

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

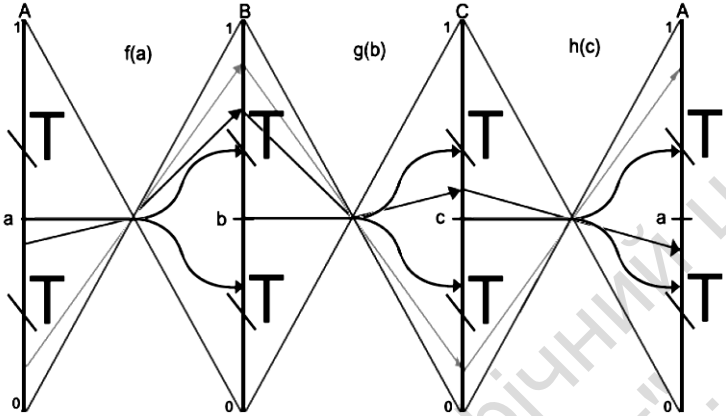


Рис. 3.26. Частину кожного з ланцюгів *A*, *B* і *C* умовно вилучаємо

Може існувати розв'язок, у якому немає вершин на *P*, тоді нам необхідно продовжити ребра, що містять *a, b,* і *c*, до ліній, які будуть перетинатися і утворювати трикутник. Якщо параметризувати область як функцію положення *b*, то отримаємо максимальну площу, коли *b* буде точно посередині відповідного ребра трикутника. Якщо *b, a* і *c* розташовані одночасно на границі *P*, то це й буде шуканим розв'язком. Інші кандидати будуть містити принаймні одну вершину, тому ми можемо вивчити всі шість із них (по дві для кожного ребра в *A, B* і *C*) і зберегти лише правильні. Для всіх правильних кандидатів ми маємо переконатися, що вони вписані точно так, як це було зроблено для двох вершин на *P*. Це дасть нам максимальний прямокутник із кутами на *A, B* і *C*. Повторюючи кроки для трьох інших конфігурацій, отримаємо найбільший прямокутник із трьома вершинами на *P*. Порівнюючи результат з випадком для двох вершин на *P*, одержимо остаточно найбільший прямокутник. На рис. 3.27 показано приклад, коли найбільший прямокутник має вершини *A, B* і *C* на *P*.

Оцінка складності алгоритму

Теорема 3.9. *Вписування прямокутника у n-вершинну опуклу оболонку за допомогою методу prune-and-search для фіксованих точок можна виконати за час O(nlogn).*

Видавничо-поліграфічний центр "Київський університет"
Вільна електронна версія

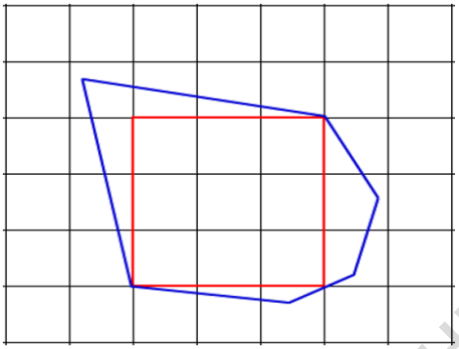


Рис. 3.27. Приклад із трьома вершинами на границі *P*

Доведення. При пошуку прямокутника із двома вершинами *A* і *C*, які лежать на границі *P* і збігаються 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 перевіряємо кандидатів на вписування за час $O(\log B) + O(\log D)$. Складність попереднього prune-and-search-методу оцінена у [38] та [40] за допомогою потенційної функції, щоб показати, що ненульова стала кількість вершин відкидається на кожному попередньому кроці. У результаті досягається складність $O(\log n)$, щоб знайти нерухому точку трьох функцій. Перевірка приналежності виконується за час $O(\log D)$. Це повторюється чотири рази, щоб отримати найбільший прямокутник із трьома кутами. Як результат, загальна складність становить $O(\log n)$, а наведений вище алгоритм є найбільш ефективним із відомих алгоритмів пошуку найбільшого ортогонального прямокутника в опуклому багатокутнику.

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

Видавничо-поліграфічний центр "Київський університет"
Верстка: Київський університет

3.4. ЕЛІПС НАЙБІЛЬШОЇ ПЛОЩІ, ВПИСАНИЙ В ОПУКЛУ ОБОЛОНКУ (ВМЕО (ВО4))

Постановка задачі. *У задану n-вершинну опуклу оболонку на площині вписати еліпс найбільшої площі.*

Узагалі алгоритмів для розв'язання цієї задачі дуже мало, тому розглянемо коротко ідеї можливих алгоритмів, хоча й не зовсім оптимальних.

3.4.1. Алгоритм бінарного пошуку вершин максимального трикутника (варіант 1)

Існує гіпотеза, що еліпс однозначно задається п'ятьма точками на границі. Для нашої задачі двоїстим поняттям буде те, що еліпс однозначно задається п'ятьма прямими, які є дотичними до його границі. 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 довільно вибираємо п'ять 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, яке не дотикається до еліпса.

3. Якщо на основі цього ребра і чотирьох основних ребер ми можемо побудувати більший еліпс, то збільшимо площу початкового еліпса.

4. Кількість кроків для збільшення площі буде не більше n (за кількістю ребер в опуклій оболонці).

Оцінка складності алгоритму

Теорема 3.10. *Вписування еліпса максимальної площі у n-вершинну опуклу оболонку за допомогою запропонованого алгоритму можна виконати за час* $O(n^2)$.

Випускний університет "Київський університет"

Доведення. Доводимо сортуванням $O(n\log n)$. Вибираємо початковий еліпс за час $O(1)$. Ітерації збільшення площі вписаного еліпса потребують часу $O(n)$ кожна. Отже, загальна складність алгоритму становить $O(n^2)$.

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

3.4.2. Алгоритм бінарного пошуку вершин максимального трикутника (варіант 2)

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 багатокутник P. Тоді має місце така ідея алгоритму.

***Алгоритм* (*основна ідея*)**

1. Побудуємо бісектриси всіх кутів багатокутника P.

2. Із множини точок перетину бісектрис оберемо 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 багатокутника за орієнтацією вписування (діаметр вписаного еліпса не має їх перевищувати). Для цього проведемо через знайдені точки A та B пряму. Той відрізок прямої, що лежатиме всередині багатокутника, буде більшим діаметром шуканого еліпса.

4. Перевіримо еліпс на входження в багатокутник. Для цього підійде така властивість еліпса: якщо точка P належить еліпсу, то

$$\|F_1P\| + \|F_2P\| = 2a,$$

де F_1, F_2 – фокуси еліпса, a – його більший радіус.

Оцінка складності становить $O(n\log n)$.

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

4. ЗАДАЧІ НА РОЗТАШУВАННЯ

4.1. НАЙБІЛЬШЕ ПОРОЖНЄ КОЛО РМК (АО5)

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

Задачі такого роду можуть виникати у промисловості, коли об'єкт – шкідливе виробництво, наприклад хімічний завод, атомна станція, – має бути розміщений якнайдалі від заданих об'єктів (населених пунктів, будинків тощо), щоб мінімізувати ефект від такого сусідства, або при створенні нового виробництва, якщо бажано уникнути конкуренції за вільний простір. Прикладом може бути і задача вибору місця розташування магазину продажу тютюнових і алкогольних виробів, який не можна розміщувати поблизу від соціальних або побутових закладів – шкіл, дитсадків, лікарень тощо. Інший приклад – задача розкроювання, коли прямокутник (шматок тканини, листовий метал) має дефектні точки і необхідно оптимізувати витрати матеріалу. У військовій справі є задачі, які зводяться до задачі РМК, зокрема для знаходження максимального радіуса дії радіолокаційних станцій на місцевості з урахуванням існуючих обмежень і особливостей рельєфу, які впливають на якість передавання та приймання сигналу.

Перші алгоритми [45] розв'язували цю задачу за $O(n^3)$ часу в найгіршому випадку. Шеймос і Препарата [5] показали, що розв'язання може бути проведене за час $O(n\log n)$ з використанням діаграми Вороного [46], що й зумовило алгоритм розв'язання задачі, який є найбільш оптимальним і простим у реалізації. Доведення описане в роботі [47].

Алгоритм пошуку найбільшого порожнього кола на основі діаграми Вороного. Переформулюємо нашу задачу: треба знайти точку p_0 (як центр кола) на площині, яка б задовольняла умову

$$P_0 = \min \max \left\{ (x_0 - x_i)^2 + (y_0 - y_i)^2 \right\}.$$

Виробничо-торгівельний центр "Київський університет" Вовсій не для науки

Алгоритм (*покроково*)

Можливі два випадки розташування центра шуканого кола.

1. Усередині опуклої оболонки. Нехай коло з таким центром "роздувається", поки не торкнеться якоїсь точки. Очевидно, що радіус не буде максимальним, якщо коло пройде тільки через одну (p_0 усередині комірки) або дві (p_0 на ребрі діаграми Вороного) точки. Максимум можливий, якщо коло перетне три точки, тоді шукана p_0 – вершина діаграми Вороного (при цьому точки лежать більше ніж на одному півколі).

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

Отже, 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 усередині опуклої оболонки. Якщо так, то підрахувати радіус кола із центром у цій вершині, за потреби зберегти поточний результат як максимум.
4. Для кожного ребра діаграми Вороного підрахувати перетин його з опуклою оболонкою і знайти радіус кола з відповідним центром, за потреби зберегти поточний результат як максимум.
5. Повернути максимум у вигляді шуканого кола.

Оцінка складності алгоритму

Теорема 4.1. *Пошук найбільшого порожнього кола на множині S із n точок на площині можна виконати методом на основі діаграми Вороного за оптимальний час* $O(n\log n)$.

Доведення. Як відомо, усі вершини діаграми Вороного можна отримати за час $O(n\log n)$, таку саму складність може мати побудова опуклої оболонки (доведення див. у [5]). Нижче буде показано, що перетин ребер опуклої оболонки і ребер діаграми Вороного можна знайти за час $O(n)$.

Розглянемо спочатку два факти.

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

Видавничо-поліграфічний центр "Київський університет"
Безоплатно для друку

2. Кожне ребро опуклої оболонки перетинає хоча б одне ребро діаграми Вороного, оскільки кожне ребро опуклої оболонки з'єднує дві різні точки із заданої множини, які належать двом різним багатокутникам Вороного.

Припустимо, що кожен необмежений багатокутник Вороного закритий відрізком на нескінченно віддаленій прямій, що лежить у площині. Нехай ребра e_i опуклої оболонки перераховані в порядку обходу проти годинникової стрілки (рис. 4.1). Згідно із п. 2 завжди існує точка u перетину довільного ребра e_j з ребром діаграми Вороного r.

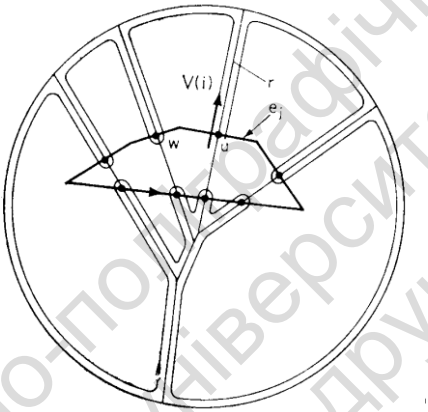


Рис. 4.1. Взаємне розташування опуклої оболонки та діаграми Вороного

Визначивши орієнтацію r у зовнішню відносно опуклої оболонки область, позначимо через $V(i)$ багатокутник діаграми Вороного, що міститься ліворуч від r. Рухаючись із точки u границею цього багатокутника 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}. Кожне ребро $V(i)$ перевіряємо на перетин із двома ребрами оболонки, що відбувається за постійний час. Потім обхід повторюємо для точки w тощо, поки не повернемось у вихідну точку u.

Видавничо-поліграфічний центр "Волинський університет". Версія 1.0 для друку

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 максимум чотири рази, тобто маємо оцінку $O(n)$. Звідси отримуємо загальну оцінку часу виконання всього алгоритму $O(n\log n)$.

На рис. 4.2 зображено результат програмної реалізації розв'язання задачі пошуку найбільшого порожнього кола на множині із 200 точок мовою Java.

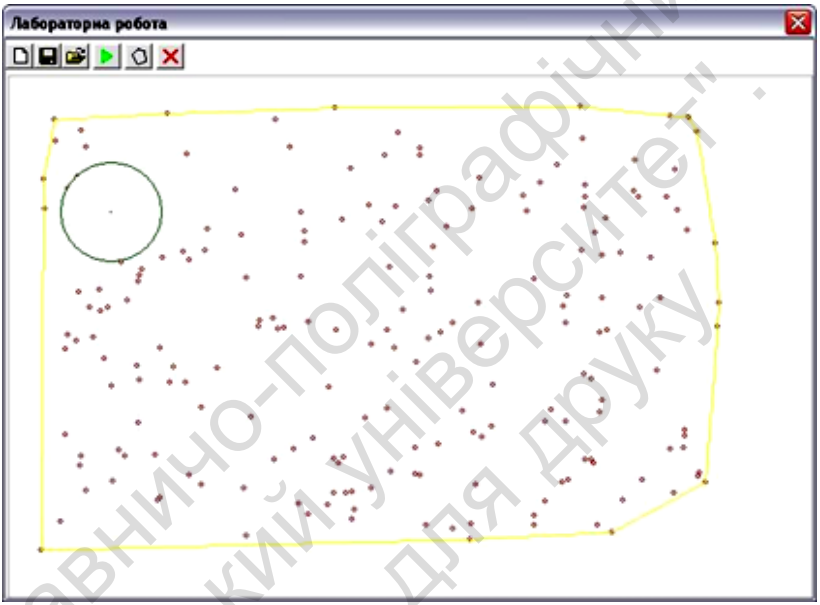


Рис. 4.2. Приклад програмної реалізації пошуку найбільшого порожнього кола на множині із 200 точок

Висновки. Описано оптимальний алгоритм пошуку кола найбільшого радіуса на множині точок на площині на основі діаграми Вороного з оцінкою складності $O(n\log n)$. Його вдосконалення можуть бути пов'язані лише з певними особливостями програмної реалізації, але це стосуватиметься конкретних випадків із конкретними завданнями, вимогами й

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

обмеженнями. Зауважимо, що діаграма Вороного робить зручнішою реалізацію, до того ж чудово вписується в сучасну концепцію повторного використання програмного забезпечення.

4.2. 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 усередині жодної точки множини S.*

Розглянемо задачу пошуку найбільшого порожнього прямокутника, яка виникає, наприклад, коли потрібно знайти найкраще розташування для магазину чи стадіону, а також при роботі з базами даних. Для визначеності вхідної множини задачі будемо вважати, що задана множина обмежена деяким прямокутником (рис. 4.3).

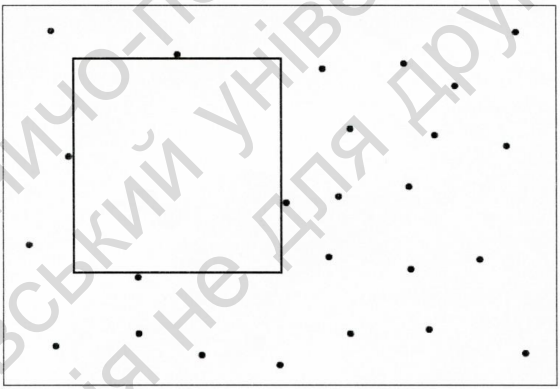
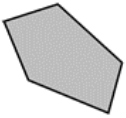
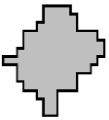


Рис. 4.3. Задання вхідних даних задачі

Незважаючи на практичне значення, лише для простих областей (Convex, Constrained Staircase) були знайдені алгоритми з оптимальною оцінкою [48–51] (табл. 4.1).

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

Таблиця **4.1**

№	Форма області		Оцінка складності	Автори
1	Найбільший опуклий багатокутник		$\theta(n)$	Н. Амента [49] (опукле програмування)
2	Найбільший ортогональний східчастий багатокутник 1		$\theta(n)$	А. Аггарвал, Дж. Вайн [48]
3	Найбільший ортогональний східчастий багатокутник 2		$O(n\log^2 n)$ $O(na(n))$	М. МакКенна, Дж. О'Рурк, С. Шурі [50] (використання LAECR)
4	Найбільший ортогональний ізотетичний багатокутник		$O(n\log^2 n)$	М. МакКенна, Дж. О'Рурк, С. Шурі [50] (використання LAECR)
5	Найбільший порожній прямокутник		$O(n\log^2 n)$	Б. Чазелле, Р. Л. Дрісдейл III, Д. Т. Лі [52] А. Аггарвал, С. Шурі [53]

Для більш складних задач, зокрема нашої задачі РМП, це виявилося непростим завданням. Нині можна виділити два найбільш перспективні шляхи її розв'язання. Це методи "розділяй та володарюй" [52] і перебирання всіх дотичних прямокутників [50]. Чазелле, Дрісдейл, Лі (Chazelle, Drysdale, Lee) [52] у 1986 р. запропонували алгоритм, який використовує метод "розділяй та володарюй" та діаграму Вороного для РМП із часовою складністю $O(n\log^3 n)$. Аггарвал та Шурі (Aggarwal, Suri) [53] поліпшили теоретичну оцінку до $O(n\log^2 n)$, але зауважили, що константа виходить дуже великою, тому для реалізації краще використовувати попередній алгоритм.

Видавничо-поліграфічний центр "Київський університет" Безоплатно для друку

4.2.1. Алгоритм пошуку найбільшого порожнього прямокутника Орловського

Для розв'язання задачі про РМП скористаємось методом перебирання всіх дотичних прямокутників, який був уперше запропонований Орловським [54]. Уведемо поняття дотичного прямокутника.

Означення 4.1. ***Дотичним прямокутником*** *називається прямокутник, який кожною стороною дотикається до точки або сторони оточуючого прямокутника, проте не повністю міститься в оточуючому прямокутнику і не містить жодної точки всередині.*

Лема 4.1. *Розв'язком задачі про РМП буде дотичний прямокутник.*

Доведення. Припустимо від противного, що розв'язком задачі є РМП, який не є дотичним прямокутником. Тоді існує хоча б одна сторона, до якої він може бути розширений, що неможливо. Отже, дійшли суперечності.

Лема 4.2. *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. Тоді для того, щоб розв'язати задачу РМП, необхідно перебрати всі можливі дотичні прямокутники. Це можна зробити, як показав Орловський [54], за допомогою двох відсортованих списків: за *x* та *y,* відповідно. Можна запропонувати інший підхід пошуку РМП, який використовує структуру даних *висотне дерево.*

Означення 4.2. ***Висотним деревом*** *називається геометрична структура даних, яка зберігає точки на площині та має такі властивості:*

1) *є бінарним деревом;*

2) *лівий син міститься нижче* (*менша y-координата*) *і лівіше* (*менша x-координата*);

3) *правий син міститься не вище* (*y-координата не більша*) *і не лівіше* (*x-координата не менша*).

Видавничо-поліграфічний центр "Київський університет" Бібліотека для пошуку

Лема 4.3. *Для будь-якого набору точок висотне дерево може бути побудоване за час* $O(n\log n)$.

Доведення. Для доведення задіємо алгоритм побудови висотного дерева.

1. Відсортовуємо точки за парою (x, y).

2. Будуємо висотне дерево, на кожному кроці додаючи до нього по одній точці з відсортованого списку (час $O(\log n)$). Для цього використаємо *дерево відрізків*.

Означення 4.3. ***Лівою праворуч гілкою*** *називається список вершин, який складається з лівого сина та правого сина лівого сина, правого сина правого сина лівого сина тощо аж до кінця.*

Означення 4.4. ***Правою ліворуч гілкою*** *називається список вершин, який складається із правого сина та лівого сина правого сина, лівого сина лівого сина правого сина тощо аж до кінця.*

Лема 4.4. *Усі дотичні прямокутники, які верхньою стороною дотикаються до кореня висотного дерева, іншими сторонами дотикаються або до сторін оточуючого прямокутника, або до точок гілок "ліва праворуч" та "права ліворуч".*

Доведення. Звернувши увагу на означення гілок висотного дерева, можна побачити, що між лівою та правою гілками відсутні точки. Оскільки дотичні прямокутники не містять жодної точки всередині, то всі вони будуть формуватись лише точками гілок і сторін оточуючого прямокутника.

Корінь висотного дерева може бути видалений за лінійний час відносно суми точок гілок "ліва праворуч" та "права ліворуч" за допомогою *зшивання гілок*. Отже, ми можемо розглянути всі дотичні прямокутники, що зверху дотикаються до кореня висотного дерева, потім видалити цей корінь і повторити процес. У результаті ми розглянемо всі дотичні прямокутники, окрім тих, що верхньою стороною спираються на верхню сторону оточуючого прямокутника. Їх ми розглянемо окремо.

Алгоритм (*покроково*)

1. Розглянути дотичні прямокутники, що дотикаються верхньою стороною до верхньої сторони оточуючого прямокутника.

2. Побудувати висотне дерево.

Видавничо-поліграфічний центр "Київський університет" Версія для друку

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, то перейти до п. 3.

Оцінка складності алгоритму

Теорема 4.2. *Пошук найбільшого порожнього прямокутника на множині S із n точок на площині можна виконати алгоритмом Орловського за час* $O(n^2)$ *у найгіршому та* $O(n\log n)$ *у середньому випадку.*

Доведення. П. 1 алгоритму можна виконати за час $O(n)$ за умови, що на вхід надходять відсортовані по x точки. Це не впливає на час виконання всього алгоритму. П. 2 можна виконати (як було показано) за час $O(n\log n)$. Інші пункти алгоритму перебирають дотичні прямокутники, витрачаючи на кожний постійну кількість часу. Отже, запропонований алгоритм буде працювати за час, що дорівнює $O(n\log n+s)$, де s – кількість дотичних прямокутників, і, як було показано в лемі 4.2, дотичних прямокутників буде щонайбільше n^2 у найгіршому випадку, тобто оцінка складності алгоритму в найгіршому випадку буде $O(n^2)$ та $O(n\log n)$ – у середньому. Алгоритм також потребує $O(n)$ пам'яті.

Висновки. Описано алгоритм, який дозволяє за час $O(n^2)$ у найгіршому 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 найгіршому випадку з використанням методу перебирання всіх дотичних прямокутників не може бути поліпшена, оскільки на розглядання одного дотичного прямокутника витрачається постійна кількість часу.

4.2.2. Алгоритм на основі діаграми Вороного

У розділі 4.1 описано ідею оптимального алгоритму пошуку найбільшого порожнього кола на множині S із n точок на площині за допомогою діаграми Вороного. Звідси випливає

Видавничий центр "Київський університет" Вороня не для друку

логічне питання щодо застосування цього підходу до пошуку найбільшого порожнього прямокутника, щоб отримати оптимальний алгоритм. Розглянемо ідею такого алгоритму.

Алгоритм (*основна ідея*)

Розглянемо спочатку задачу для випадку, коли наша множина S складається 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, які утворюють опуклий чотирикутник). Можна довести, що для прямокутника максимальної площі, сторони якого проходять через вершини опуклого чотирикутника, виконується таке твердження [55].

Твердження 4.1. *Одна з пар протилежних сторін прямокутника перпендикулярна до діагоналі опуклого чотирикутника, що визначається точками, через які проходять сторони прямокутника.*

Спираючись на цю властивість, побудуємо наш алгоритм.

Як було показано у [55], якщо вписати прямокутник максимальної площі у чотири точки, то одна пара сторін цього прямокутника буде паралельна одній із діагоналей опуклого багатокутника, утвореного із цих точок.

Перевіримо дію 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. Побудуємо прямі l_1 та l_2, що проходять відповідно через точки p_1 та p_2 і перпендикулярні 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 вважатимемо, що p_3 є найближчою точкою ліворуч від p_1p_2, а p_4 – праворуч). 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 отримаємо в результаті, що чотири прямі l_1, l_2, l_3, l_4 визначають прямокутник, який є можливим роз-

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

в'язком задачі. Зауважимо, що точки p_3 та p_4 не завжди існують, але в тому випадку, коли одна (чи обидві) з них не існують, пара точок p_1 та p_2 не може утворювати діагональ, що перпендикулярна до сторін найбільшого порожнього прямокутника.

Згідно із цим підходом необхідно перебрати всі пари точок і для кожної пари знайти кандидата на максимальний порожній прямокутник. На жаль, такий підхід дає складність $O(n^3)$. Тому потрібно певним чином оптимізувати перебирання пар точок. Для цього звернемося до діаграми Вороного. Припустимо, що ми побудували її для нашої множини. Розглянемо точки p_1 та p_2 і відповідні до них багатокутники Вороного V_1 та V_2. Можна показати [56], що точки p_3 та p_4, які ми визначили вище, будуть точками, що відповідають багатокутникам Вороного V_3 та V_4, які є суміжними одночасно із V_1 та V_2. Більше того, Чазелле [57] показав, що якщо ми вже знаємо одну з точок (напр. p_1), через яку проходить ребро максимального порожнього багатокутника, то три інші точки мають бути серед точок, що відповідають тим багатокутникам Вороного, які є суміжними із V_1, а також тим, які суміжні із першими суміжними багатокутниками.

Алгоритм (*покроково*)

1. Будуємо діаграму Вороного. Її можна зобразити реберним списком з подвійними зв'язками (РСПЗ).

2. Перебираємо всі багатокутники Вороного, і для точки, що відповідає кожному такому багатокутнику, знаходимо інші три, які можуть разом визначати кандидата для максимального порожнього багатокутника. Для цього нам треба лише перебрати всі суміжні (до другого порядку) багатокутники, користуючись процедурою, описаною на початку. Як ми вже зауважували, такі три точки не завжди існують. Якщо вони не існують, то ми переходимо до наступного багатокутника Вороного, інакше порівнюємо площу новознайденого прямокутника з попередньою максимальною площею. Якщо вона більша, то ми зберігаємо

Видавничо-поліграфічний центр "Київський університет" Електронна версія для друку

новознайдений прямокутник як максимальний порожній. Далі переходимо до наступного багатокутника Вороного.

3. Перебираємо всі пари суміжних точок і відповідно пари точок, які суміжні одночасно із двома точками. Якщо таких (одночасно суміжних) точок немає, то переходимо до наступної пари.

4. Знаходимо прямокутник, вписаний у чотири точки, сторона якого паралельна одній із діагоналей багатокутника, описаного навколо цих чотирьох точок.

5. Порівнюємо площу S отриманого прямокутника із максимальною площею S_{max} прямокутників, отриманих на попередніх кроках. Якщо $S > S_{max}$, то покладаємо $S_{max} = S$ і зберігаємо цей прямокутник.

6. Беремо наступну пару суміжних точок і повторюємо із кроку 4.

Нижче наведено фрагмент алгоритму 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
```

Оцінка складності алгоритму

Теорема 4.3. *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 випадку.*

Доведення. Діаграму Вороного можна побудувати за час $O(n \log n)$. Оскільки кожен багатокутник Вороного може містити не більше ніж $O(1)$ ребер, то для перегляду кожної групи точок необхідно мати лінійний час $O(n)$. Звідси отримуємо загальну

Видавничо-поліграфічний центр "Київський університет" Зверстано для друку

складність алгоритму $O(n^2 \log n)$. На рис. 4.4 наведено приклад роботи програмної реалізації задачі.

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 у дещо іншій постановці: на площині задана множина S із n точок, що міститься всередині деякого заданого прямокутника R. Знайти прямокутник максимальної площі, що міститься всередині R, сторони якого паралельні сторонам R, при цьому він не має містити всередині точок із множини S. Для цієї задачі теж існує алгоритм зі складністю $O(n \log n)$ [53]. Наш варіант задачі про найбільший порожній прямокутник розглядався також у [55], проте основний результат був отриманий у [57].

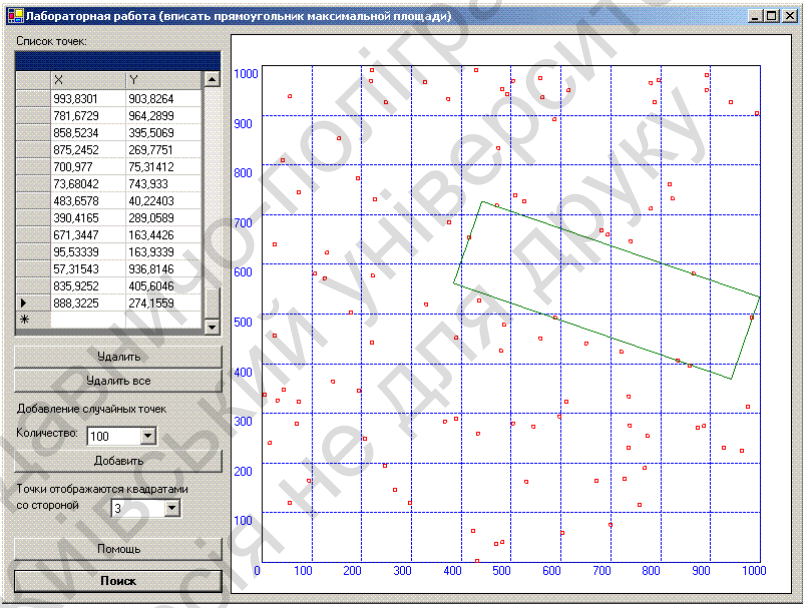


Рис. 4.4. Приклад роботи програмної реалізації задачі

Зауважимо, що наш алгоритм ілюструє одне з багатьох застосувань діаграми Вороного.

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

4.2.3. Алгоритм з використанням тріангуляції

***Алгоритм* (*покроково*)**

1. Побудувати опуклу оболонку множини точок за допомогою алгоритму обходу Грехема (рис. 4.5 а).

2. Тріангулювати множину точок усередині опуклої оболонки (рис. 4.5 б).

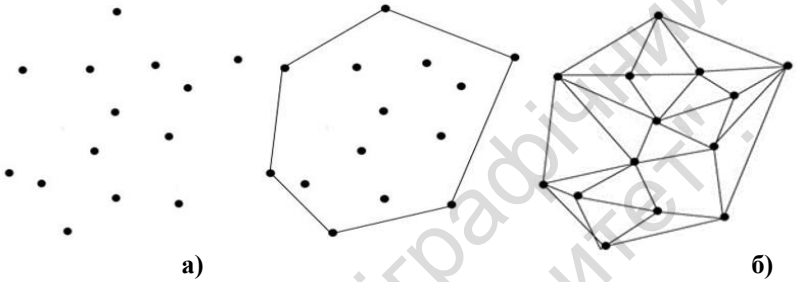


Рис. 4.5. Кроки 1, 2: а) побудова опуклої оболонки; б) тріангуляція

3. За один прохід по тріангуляції знайти трикутник найбільшої площі (рис. 4.6 а).

4. Розширити трикутник таким чином, щоб утворилася максимально можлива множина, в основі якої лежить трикутник без точок усередині (рис. 4.6 б).

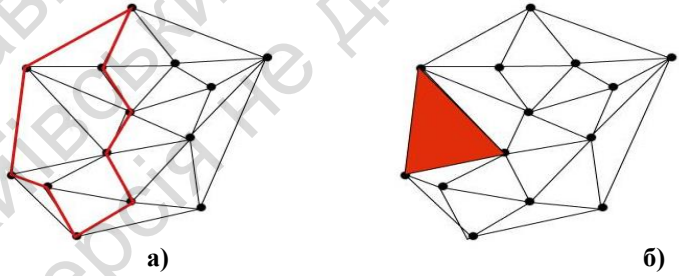


Рис. 4.6. Кроки 3, 4 алгоритму: а) пошук максимального трикутника; б) розширення області навколо трикутника

Видавничо-поліграфічний центр "Київський університет" Верстання для друку

5. За допомогою оптимізаційної функції з можливих прямокутників вибрати прямокутник найбільшої площі (рис. 4.7).

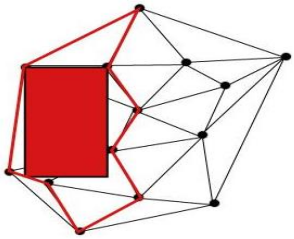


Рис. 4.7. Знайдений РМП

Оцінка складності алгоритму

Теорема 4.4. *Пошук найбільшого порожнього прямокутника на множині S із n точок на площині можна виконати алгоритмом з використанням тріангуляції за час O*(*n*log*n*) *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 вписати прямокутник.

Видавничо-поліграфічний центр "Київський університет". Версія не для друку

ЛІТЕРАТУРА

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

2. Терещенко В. М. Принцип зводимості в задачах обчислювальної геометрії / В. М. Терещенко // Вісн. Київ. ун-ту. Серія: Фіз.-мат. науки. – 2009. – № 2. – С. 149–154.

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

4. Карп Р. М. Сводимость комбинаторных проблем / Р. М. Карп // Кибернетич. сб. – М. : Мир, 1975. – Вып. 12. – С. 16–38.

5. Preparata F. Computational Geometry: An introduction / F. Preparata, M. I. Shamos. – Berlin : Springer-Verlag, 1985.

6. Гэри М. Вычислительные машины и труднорешаемые задачи / М. Гэри, Д. Джонсон. – М. : Мир, 1982.

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

8. Griffor E. R. Handbook of Computability Theory / E. R. Griffor. – Amsterdam : North Holland Publishing Co., 1999.

9. Mount Dave. Data Structures : Lecture Notes [Electronic resource] / Dave Mount. – 2001. – P. 1–23. – Access mode : http://www.cs.umd.edu/users/mount/420/Lects/420lects.pdf

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

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

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

13. Sylvester J. J. A question in the geometry of situation / J. J. Sylvester // Quarterly Journal of Mathematics. – 1857. – P. 1–79.

14. Shamos M. I. Closest point problems / M. I. Shamos, D. Hoey // Proceedings of 16th Annual IEEE Symposium on Foundations of Computer Science. Massachusetts Ave.; NW Washington, DC; United States. – 1975. – P. 151–162.

15. Welzl E. Smallest enclosing disks (balls and ellipsoids) / E. Welzl // New Results and New Trends in Computer Science : Lecture Notes in Computer Science. – Springer-Verlag, 1991. – № 555. – P. 359–370.

Видавничо-поліграфічний центр "Київський університет" Видано за авторською редакцією

16. Megiddo N. Linear-time algorithms for linear programming in R3 and related problems / N. Megiddo // SIAM Journal on Computing. – 1983. – 12 (4). – P. 759–776.

17. Elzinga J. The minimum covering sphere problem / J. Elzinga, D. W. Hearn // Management Science. – 1972. – 19. – P. 96–104.

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

19. Yaglom I. M. Boltyanski. Convex Figures / I. M. Yaglom, V. G. Boltyanski. – N.Y. : Holt, Rinehart and Winston, 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. Moshtagh N. Minimum volume enclosing ellipsoid [Electronic resource] / N. Moshtagh. – GRASP Lab, University of Pennsylvania, 2005. – Access mode : http://www.seas.upenn.edu/nima/papers/Mim˜volellipse.pdf.

22. Рубльов Б. В. Різні модифікації оптимального алгоритму побудови еліпса мінімальної площі / Б. В. Рубльов // Вісн. Київ. ун-ту. Серія: Фіз.-мат. науки. – 1999. – № 1. – С. 223–228

23. Peng S. Computation of minimum volume covering ellipsoids / S. Peng, R. M. Freund // Operations Research. – 2004. – № 52. – P. 690–706.

24. Kumar P. Minimum volume enclosing ellipsoids and core sets / P. Kumar, E. A. Yıldırım // J. Optim. Theory Appl. – 2005 – № 126. – P. 1–21.

25. The Computational Geometry Algorithms Library. CGAL Open Source Project [Electronic resource]. – Access mode : [http://www.cgal.org/]

26. Акопян А. В. Геометрические свойства кривых второго порядка / А. В. Акопян, А. А. Заславский. – 2-е изд., доп. – М. : МЦНМО, 2011.

27. Препарата Ф. Обчислювальна геометрія: вступ / Ф. Препарата, М. Шеймос. – М. : Мир, 1989.

28. Hoffmann Gernot. Ellipse Through Four Points [Electronic resource] / Gernot Hoffmann. – Access mode : //[http://docs-hoffmann.de/ ellipse08032004.pdf]

29. Анісімов А. В. Основні алгоритми обчислювальної геометрії : навч.-метод. посіб. / А. В. Анісімов, В. М. Терещенко, І. В. Кравченко. – К. : ВПЦ "Київ. ун-т", 2002. – С. 82.

30. Тот Л. Ф. Расположение на плоскости, на сфере и в пространстве / Л. Ф. Тот. – М. : Гос. изд-во физ.-мат. лит., 1958.

31. Красильников И. Нахождение вписанной окружности в выпуклом многоугольнике с помощью тернарного поиска [Электронный ресурс] / И. Красильников. – 2008. – Режим доступа : [http://e-maxx.ru/algo/ inscribed_circle_ternary]

Видавничо-поліграфічний центр "Київський університет" Електронна версія для друку

32. Красильников И. Нахождение вписанной окружности в выпуклом многоугольнике методом "сжатия сторон" [Электронный ресурс] / И. Красильников. – 2012. – Режим доступа : [http://e-maxx.ru/algo/inscribed_circle]

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

34. Math Central. Department of Mathematics and Statistics University of Regina Regina, Saskatchewan, S4S 0A2, Canada [Electronic resource]. – Access mode : [http://mathcentral.uregina.ca/QQ/database/QQ.09.06/h/matthew2.html]

35. Алгоритмы. Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – М. : Вильямс, 2013.

36. Ахо Х. Построение и анализ вычислительных алгоритмов / Х. Ахо, Дж. Хопкрофт, Дж. Ульман. – М. : Мир, 1979.

37. Роджерс Д. Алгоритмические основы машинной графики / Д. Роджерс. – М. : Мир, 1989.

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

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

40 Kirkpatrick D. Tentative prune-and-search for computing fixed-points with applications to geometric computation / D. Kirkpatrick, J. Snoeyink // Fundamental Informatic. – 1995. – 22. – P. 353–370.

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

42. Sud D. Computing the largest orthogonal rectangle in a convex polygon [Electronic resource] / D. Sud // CS507A-Computational Geometry. – 2003. – Access mode : [http://cgm.cs.mcgill.ca/~athens/cs507/Projects/2003/DanielSud.]

43. Херн Д. Компьютерная графика и стандарт OpenGL / Д. Херн, М. П. Бейкер. – 3-е изд. ; пер. с англ. – М. : Изд. дом "Вильямс", 2005.

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

45. Dasarathy B. On some maximum location and classifier problems : in Computer Science Conference, Washington, D.C. (unpublished lecture) / B. Dasarathy, L. J. White. – 1975.

Вирішення інтерактивних задач "Київський університет" Видавничо-поліграфічний центр

46. Fortune S. J. A sweep line algorithm for Voronoi diagrams / S. J. Fortune // Algorithmica. – 1987. – 2. – P. 153–174.

47. Manber U. Propabilistic, nondeterministic and alternating decision trees / U. Manber, M. Tompa // Tech. Rep. № 82-03-01, Univ. – Washington, 1982.

48. Aggarwal A. 18.409 Computational Geometry : Lecture Notes / A. Aggarwal, J. Wein. – Cambridge, Mass. : Laboratory for Computer Science, MIT, 1988.

49. Amenta N. Finding a line transversal of axial objects in three dimensions : In proceedings of the third annual ACM-SIAM symposium on Discrete algorithms / N. Amenta. – 1992. – P. 66–71.

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

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

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

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

54. Orlowski M. A new algorithm for the largest empty rectangle problem / M. Orlowski // Algorithmica. – 1990. – 5. – P. 67–73.

55. Chaudhuri J. Largest empty rectangle among a point set / J. Chaudhuri, S. C. Nandy, S. Das // Journal of Algorithms. – 2003. – 46(1). – P. 54–78.

56. Lee D. T. Voronoi diagram: revisiting / D. T. Lee, D. Dobkin // Discrete and Computational Geometry. – 1993. – Vol. 4, № 3. – P. 123–136.

57. Chaselle B. Computing the Largest Empty Rectangle / B. Chaselle, R. L. Scot Drysdale, D. T. Lee // SIAM Journal on Computing. – 1970. – 15(1). – P. 43–54.

58. Radamacher H. The Enjoyment of Mathamathics / H. Radamacher, O. Toeplitz. – Princeton University Press, 1967.

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

60. Chang J. S. A polynomial solution for potato-peeling and other polygon inclusion and enclosure problems : in Proceedings of Foundation of Computer Science, IEEE Computer Society, West Palm Beach, Fla / J. S. Chang, C. K. Yap. – 1984. – P. 408–416.

Видавничо-поліграфічний центр "Київський університет"

61. Depano N. A. A. Finding restricted k-envelopes for convex polygons : In Proc. 22nd Allerton Conf. on Comm., Conk, and Comp., Univ. of Illinois, Urbana / N. A. A. Depano, A. Aggarwal. – 1984. – P. 81–90.

62. Reynolds Craig W. Flocks, herds and schools: A distributed behavioral model / W. Craig Reynolds // ACM SIGGRAPH Computer Graphics. – 1987. – 21. – P. 25–34.

63. Emergence of Collective Behavior in Evolving Populations of Flying Agents / L. Spector, J. Klein, C. Perry, M. Feinstein // Genetic Programming and Evolvable Machines. – 2005. – Vol. 6. – P. 111–125.

64. Search and tracking algorithms for swarms of robots: A survey / M. Senanayake, I. Senthooran, J. C. Barca et al. // Robotics and Autonomous Systems. – 2016. – 75(B). – P. 422–434.

65. An optimal algorithm for findng minimal enclosing triangles / J. O'Rourke, A. Aggarwal, S. Maddila, M. Baldwin // Journal of Algorithms. – 1986. – 7(2). – P. 258–269.

66. Klee V. Finding the smallest triangles containing a given convex polygon / V. Klee, M. C. Laskowski // J. Algorithms. – 1985. – 6. – P. 359–375.

67. Parvu Ovidiu. Implementation of linear minimum area enclosing triangle algorithm / Ovidiu Parvu, David Gilbert // Computational and Applied Mathematics. – 2016. – 35. – P. 423–438.

Видавничо-поліграфічний центр "Київський університет" Версія не для друку

Навчальне видання

ТЕРЕЩЕНКО Василь Миколайович

Аналіз методів
розв'язання оптимізаційних задач обчислювальної геометрії

Навчальний посібник

Редактор Н. Земляна

Оригінал-макет виготовлено ВПЦ "Київський університет"
Виконавець В. Гаркуша

Формат 60x84$^{1/16}$. Ум. друк. арк. 6,51. Наклад 100. Зам. № 222-10394.
Гарнітура Times New Roman. Папір офсетний. Друк офсетний. Вид. К8.
Підписано до друку 01.03.2022

Видавець і виготовлювач
ВПЦ "Київський університет",
б-р Тараса Шевченка, 14, м. Київ, 01601, Україна
☎ (38044) 239 32 22; (38044) 239 31 72; тел./факс (38044) 239 31 28
e-mail: vpc_div.chief@univ.net.ua; redaktor@univ.net.ua
http: vpc.knu.ua
Свідоцтво суб'єкта видавничої справи ДК № 1103 від 31.10.02

Видавничо-поліграфічний центр "Київський університет"
Версія ноутбука