Київський національний університет
імені Тараса Шевченка

На правах рукопису

ФІСУНЕНКО АНДРІЙ ЛЕОНІДОВИЧ

УДК 004.021:004.023:004.054:519.16

ПОБУДОВА ГЕНЕРАТОРА ГЕОМЕТРИЧНИХ ОБ’ЄКТІВ ІЗ ЗАДАНИМИ ВЛАСТИВОСТЯМИ НА ПЛОЩИНІ

01.05.01 – теоретичні основи інформатики та кібернетики

Дисертація на здобуття наукового ступеня
кандидата фізико-математичних наук

Науковий керівник:
Терещенко Василь Миколайович,
фізико-математичних наук,
професор

Київ – 2016

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ....... 4
ВСТУП 5
РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ ТА АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ДО РОЗВ’ЯЗКУ ЗАДАЧ ПОРОДЖЕННЯ ГЕОМЕТРИЧНИХ ОБ’ЄКТІВ..... 12
1.1. Попередні зауваження....... 12
1.2. Загальні означення та основні припущення. 13
1.3. Складність алгоритмів та модель обчислень....... 17
1.4. Узагальнена постановка задач 19
1.5. Аналіз існуючих підходів до розв’язку окремих випадків задач 23
Висновки до Розділу 1....... 30
РОЗДІЛ 2. АПАРАТ ДЛЯ АНАЛІЗУ ВХІДНИХ ДАНИХ ТА РОЗВ'ЯЗАННЯ ЗАДАЧ ПОРОДЖЕННЯ ГЕОМЕТРИЧНИХ ОБЄКТІВ....... 32
2.1 Вступні відомості 32
2.2 Діаграми еквівалентності зіркових розбиттів....... 36
2.3 Граф взаємної видимості вільних точок 40
Висновки до Розділу 2....... 41
РОЗДІЛ 3. МЕТОДИ ПОРОДЖЕННЯ ДОВІЛЬНИХ ПРОСТИХ МНОГОКУТНИКІВ....... 43
3.1 Доцільність оптимізації методів з повним перебором 43
3.2 Повний перебір з перевіркою самоперетинів....... 44
3.3 Повний перебір з відсіканням зворотних послідовностей....... 48
3.4 Повний перебір з перевіркою ланцюга на самоперетини 52
3.5 Повний перебір та граф взаємної видимості вільних точок 55
Висновки до Розділу 3....... 82
РОЗДІЛ 4. МЕТОДИ ПОРОДЖЕННЯ ОКРЕМИХ ТИПІВ МНОГОКУТНИКІВ....... 83
4.1 Побудова зіркових многокутників....... 83
4.2 Побудова простих многокутників нарощуванням опуклих оболонок 84
4.3 Квітко-подібні многокутники і метод їх породження 88

4.4 Побудова простих многокутників нарощуванням трикутниками....... 90
Висновки до Розділу 4 93
РОЗДІЛ 5. ПРАКТИЧНА РЕАЛІЗАЦІЯ 95
5.1 Загальна архітектура програмної реалізації генератора геометричних об’єктів 95
5.2 Формат вхідних даних 96
5.3 Формат вихідних даних.. 97
5.3 Приклади роботи генератора геометричних об’єктів та засобів візуалізації 99
5.4 Застосування точного розв’язку Задачі 1 для отримання точного значення оптимальної конфігурації при N=7 101
5.5 Застосування точного розв’язку Задачі 1 для оцінювання жадібного алгоритму знаходження простого многокутника мінімальної площі 102
Висновки до Розділу 5 103
ВИСНОВКИ.. 104
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 107
ДОДАТКИ... 115
Додаток А. Порівняння точного і наближеного розв’язків задачі про простий многокутник найменшої площі. 115
Додаток Б. Вихідний код окремих частин практичної реалізації генератора геометричних об’єктів.. 123

ПЕРЕЛІК ПОЗНАЧЕНЬ

$\boldsymbol{E}^d$ -	Евклідовий простір розмірності d.
S -	$S=\{p_1, p_2, ..., p_N\}$ – скінчена множина точок евклідового простору $\boldsymbol{E}^d$, де i-а точка p_i – це d-плекс $(x_1, x_2, ..., x_d)$, що складається з дійсних чисел x_i $(i=1, ..., d)$.
$CH(S)$	Опукла оболонка (convex hull) множини точок S.
$\forall$	для будь-якого
$\exists$	існує
$\|S\|$	міцність множини S (кількість елементів для скінченої дискретної множини)
$SP(S)$	$SP(S)=\{P_1(S), ..., P_M(S)\}$ – множина усіх можливих простих многокутників для заданої множини точок S.

ПЕРЕЛІК СКОРОЧЕНЬ

ГО	Геометричний об’єкт
ОО	Опукла оболонка
RAM	Random Access Memory (пам’ять довільного доступу)
CAD	Computer-Aided Design (автоматизоване проектування)
DFS	Depth First Search (пошук в глибину)
BCC	Biconnected Component (двозв’язна компонента)

ВСТУП

Актуальність теми. Сучасній людині щодня доводиться мати справу з результатами роботи алгоритмів обробки геометричних об’єктів обчислювальними системами: в графічних інтерфейсах цифрових пристроїв, в автоматизовано створених конструкціях, інструментах, машинах. Сучасні технології базуються на проектуванні виробів в CAD-системах, які, в свою чергу, зараз неможливо уявити без алгоритмів обчислювальної геометрії.

Як самостійна галузь науки, обчислювальна геометрія сформувалася в середині 70-х років минулого століття. Разом з розвитком комп’ютерної техніки і засобів графічного вводу-виводу потреби в результатах досліджень цієї тематики тільки зростали. Задачам і методам ефективної обробки геометричних об’єктів приділяли увагу такі вчені, як M. Shamos, F. Preparata, J. O’Rourke, R. Karp, M. Overmas, B. Chazelle, J. Goodman, N. M. Amato, K. Brown, C. Papadimitriou, S. Fekete, F. Hurtado, P. Erdos, M. Newborn, S. Akl, E. M. Arkin, Y.-J. Chiang, M. Held, J.S.B. Mitchell, V. Sacristan, S. Skiena, T.-C. Yang та багато інших.

Многокутники – це один з найпростіших геометричних об’єктів, але, незважаючи на простоту, дуже багато складних задач обчислювальної геометрії зводяться до побудови і знаходження їх певних характеристик, перевірки властивостей.

Окрім прямого традиційного застосування в CAD-системах, комп’ютерній графіці та системах візуалізації, многокутники інтенсивно використовуються в розпізнаванні образів, обробці супутникових та медичних зображень, мультимедійних та GIS-системах. Останнім часом, у зв’язку з бурхливим розвитком таких нових прикладних міждисциплінарних напрямків, як віртуальна та збагачена реальність, потреби в ефективних методах обробки геометричних об’єктів стають ще більш актуальними.

Тому алгоритми обробки многокутників – один із найбільш важливих розділів обчислювальної геометрії. Зокрема, цей напрямок досліджень

розвивався і вітчизняними вченими, такими як Анісімов А. В. , Шор Н.З., Ю.Г. Стоян, М.І. Шлезінгер, Є.М. Кисельова, Ю.І. Петунін, В.П. Клименко, О.М. Васюков, О.А. Ємець, Терещенко В. М., Клюшин Д.А., Рубльов Б. В., Семенова Н.В., Панкратов О. В та інші.

Водночас з практичними задачами існує низка відкритих фундаментальних проблем обчислювальної геометрії, зокрема питання щодо складності підрахунку усіх можливих простих многокутників для заданої множини точок, пошуку, так званих, оптимальних конфігурацій для заданої кількості точок, які максимізують кількість можливих простих многокутників, породження многокутників із заданими властивостями.

Ці задачі можуть використовуватись для *забезпечення якості програмно-апаратних комплексів, що використовують такі алгоритми* обчислювальної геометрії. Наприклад, для України є дуже нагальною проблема екологічного моніторингу лісових та інших ресурсів, оцінки та аналіз яких здійснюються з використанням супутникових зображень, які, в свою чергу, обробляються із застосуванням многокутників та алгоритмів над ними [64].

Таким чином, тематика даної дисертаційної роботи, що присвячена базисним задачам комбінаторної та обчислювальної геометрії є актуальною.

Зв’язок роботи з науковими програмами, планами, темами. Робота є складовою частиною наукових робіт, які ведуться на кафедрі математичної інформатики факультету кібернетики Київського національного університету імені Тараса Шевченка при виконанні теми “Створення теоретичних основ, методів та програмних засобів інтелектуалізації інформаційно-комунікаційних та трансформерних технологій” (№ держреєстрації – 0111U005416, 2011-2015 рр.).

Мета і задачі дослідження. Мета і задачі дослідження. Метою дисертаційного дослідження є створення ефективних методів і алгоритмів генерації геометричних об’єктів із заданими властивостями та їх практична реалізація.

Для досягнення поставленої мети необхідно було вирішити такі основні задачі:

1. розробка алгоритмів генерації простих многокутників із заданими властивостями;
2. дослідження особливостей генерації простих многокутників в залежності від конфігурації множини точок, які повинні бути вершинами отриманих многокутників;
3. пошук оптимальних об'єктів за визначеними критеріями;
4. узагальнення методів генерації простих многокутників для породження простих поліедрів у просторах більшої розмірності (*d=3, 4,…*);
5. практична реалізація алгоритмів та пов'язані задачі:
   – оптимізація алгоритмів генерації;
   – візуалізація алгоритмів та результатів їх роботи;
   – практичне застосування розроблених алгоритмів і методів.

Об'єкт дослідження – процес породження геометричних об'єктів на скінчених множинах точок з повним включенням усіх точок до породжених об'єктів.

Предмет дослідження – стратегії та підходи ефективного породження геометричних об'єктів.

Методи дослідження. Дисертаційна робота ґрунтується на:

- загальних методах прикладної теорії алгоритмів;
- методах обчислювальної геометрії;
- методах інтерактивної комп'ютерної графіки та візуалізації.

Наукова новизна одержаних результатів. У дисертаційній роботі розвинуто методи генерації простих многокутників.

Вперше:

- розроблено метод генерації простих многокутників нарощуванням опуклих оболонок, на основі якого доведено існування простого поліедра в евклідовому просторі довільного виміру *d* для скінченої множини точок, жодні

$d+1$ з яких не лежать в одній гіперплощині, та запропоновано новий алгоритм генерації простих поліедрів, що включають усі точки вхідної множини у якості вершин;

- показано, що раніше відомі спіральні многокутники, можуть бути отримані як окремий випадок методу вирізання та нарощування опуклих оболонок;

- розширено діапазон застосування алгоритму генерації випадкових елементів з повної множини усіх можливих простих многокутників з використанням поняття графа взаємної видимості вільних точок;

- розроблено метод підрахунку усіх можливих простих полігонізацій скінченої множини точок на площині;

- досліджено новий клас об'єктів - квітко-подібні многокутники і запропоновані методи їх породження і підрахунку;

- запропоновано новий клас розташувань – діаграму еквівалентності зіркових розбиттів та досліджено її застосування для алгоритму генерації та підрахунку квітко-подібних многокутників.

Практичне значення одержаних результатів. Робота має як теоретичну, так і прикладну спрямованість. Отримані результати розширюють коло вирішених задач, що пов'язані з відкритими проблемами обчислювальної геометрії; можуть бути застосовані при розробці та тестуванні алгоритмів обчислювальної геометрії, які оперують простими многокутниками та їх узагальненнями для просторів більших вимірів - поліедрів, а також для перевірки та порівняння різних наближених розв'язків оптимізаційних задач над множинами простих многокутників на заданій скінченій множині точок.

Результати, які отримані при проведенні роботи, можуть використовуватись для забезпечення перевірки доведення коректності та якості реалізації рішень оптимізаційних задач та задач генерації геометричних об'єктів із заданими властивостями.

Особистий внесок здобувача. Всі результати, які складають суть дисертаційної роботи, одержані здобувачем самостійно. З робіт, виконаних із

співавторами, на захист виносяться лише результати, одержані особисто здобувачем.

З робіт, виконаних із співавторами, на захист виносяться лише результати, одержані особисто здобувачем. Персональний внесок здобувача до всіх наукових праць, опублікованих із співавторами 2), 5)-11) полягає у застосуванні розроблених за темою дисертації методів генерації геометричних об'єктів із заданими властивостями, як вхідні дані для перевірки програмних реалізацій відповідних алгоритмів та вимірювання результатів чисельного експерименту, порівняння результатів з відомими результатами попередніх досліджень.

Апробація результатів дисертації. Основні ідеї і положення дисертаційного дослідження обговорювалися на наукових семінарах кафедри математичної інформатики Київського національного університету імені Тараса Шевченка. Основний зміст дисертаційної роботи викладено та оприлюднено у доповідях і тезах міжнародних і всеукраїнських наукових конференцій, зокрема:

- 16th International Conference on Information Visualisation IV2012, Montpellier, France, 11-13 July 2012 (міжнародна конференція, Франція);
- XIX International Conference "Problems of Decision Making under Uncertainties", Україна, 23-27 квітня, 2012 р. (міжнародна конференція, Україна);
- 1-а міжнародна науково-технічна конференція «Комп'ютерна графіка та розпізнавання зображень», 17-18 травня, Вінниця. Україна, 2012 р.;
- 12th IC "Parallel Computing Technologies", St.-Petersburg, Russia, September 30 - October 4, 2013, PaCT 2013 – (міжнародна конференція, РФ);
- International Conference "Parallel and Distributed Computing Systems" PDCS 2013, Ukraine, Kharkiv, March 13-14, 2013 (міжнародна конференція, Україна);

- 7th International Academic Conference “Computer Science & Engineering 2015” (CSE-2015) as a part of International Youth Science Forum “Litteris Et Artibus”, November 26–28, 2015, Lviv, Ukraine (міжнародна конференція, Україна).

Публікації. За результатами дослідження опубліковано 6 наукових праць – 4 наукових статей у фахових виданнях ВАК України, 2 – у міжнародних журналах, що індексуються у наукометричній базі SCOPUS, та 6 тез міжнародних конференцій.

Список опублікованих праць за темою дисертації.

1) Фісуненко А. Л., Модель мультиагентної системи обробки візуальної інформації для вирішення задач реконструкції / А. Л. Фісуненко // Вісник Київського університету. Серія: фізико-математичні науки, №3. – 2012. – С. 261-264.
2) Фісуненко А. Л., Особливості розв’язання задач регіонального пошуку для d-вимірного випадку / В. М. Терещенко, А. Л. Фісуненко // Вісник Київського університету. Серія: фізико-математичні науки, №4. – 2012. – С. 207-210.
3) Фісуненко А. Л., Генерація простих многокутників вирізанням та нарощуванням опуклих оболонок / А. Л. Фісуненко // Вісник Київського університету. Серія: фізико-математичні науки, спецвипуск – 2013. – С. 190-193.
4) Фісуненко А. Л., Діаграма еквівалентності зіркових розбиттів та генерація простих многокутників / А. Л. Фісуненко // Вісник Київського університету. Серія: фізико-математичні науки, №2 – 2014. – С. 210-214.
5) Fisunenko A. Domain Triangulation between Convex Polytopes. / V. Tereshchenko, S. Pilipenko, A. Fisunenko (SCOPUS Author ID: 55433978800) // Journal “Procedia Computer Science”, Volume 18, Elsevier, 2013. – P. 2500-2503.
6) Fisunenko A. Algorithm for Finding the Domain Intersection of a Set of Polytopes / V. Tereshchenko, S. Pilipenko, A. Fisunenko (SCOPUS Author

ID: 55433978800) // Journal "Procedia Computer Science", Elsevier, Volume 18, 2013. – P. 459-464.

7) Fisunenko A. Solving the Range Searching Problem for Region Bounded by a Convex Surface / V. Tereshchenko, O. Socolov, A. Fisunenko// Proceedings of the 16th International Conference on Information Visualisation IV2012, Montpellier, France, 11-13 July 2012. – P. 491- 494.
8) Фісуненко А. Л., Модель мультиагентної адаптивної обробки візуальної інформації. / Терещенко В. М., Фісуненко А. Л. // Abstracts of the XIX International Conference "Problems of Decision Making under Uncertainties", Україна, 23-27 квітня, 2012. – С. 214-215
9) Фісуненко А. Л. Триангуляція прямолінійного планарного графа гострими кутами / Терещенко В. М., Фісуненко А. Л. // Матеріали 1-ої міжнародної науково-технічної конференції «Комп'ютерна графіка та розпізнавання зображень», 17-18 травня, Вінниця. Україна, 2012. – С. 184-188.
10) Fisunenko A. The Unified Algorithmic Platform for Solving Complex Problems of Computational Geometry / V. Tereshchenko, I. Budjak, A. Fisunenko // In Proc. of the 12th IC "Parallel Computing Technologies" (PaCT 2013), Springer. 2013. – P. 424-429.
11) Fisunenko A. An approach to solving complex problems of computational geometry / V. Tereshchenko, I. Budjak, A. Fisunenko // In Proceedings of International Conference "Parallel and Distributed Computing Systems" PDCS 2013 (Ukraine, Kharkiv, March 13-14, 2013. - P.320.
12) Fisunenko A., One Approach for Computing Simple Polygons on a Given Point Set in the Plain / A. Fisunenko // In Proc. of 7th International Academic Conference "Computer Science & Engineering 2015" (CSE-2015) as a part of International Youth Science Forum "Litteris Et Artibus", Lviv, Ukraine, November 26–28, 2015. – P. 44-45.

РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ ТА АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ДО РОЗВ'ЯЗКУ ЗАДАЧ ПОРОДЖЕННЯ ГЕОМЕТРИЧНИХ ОБ'ЄКТІВ

1.1. Попередні зауваження

В роботі розглядається коло задач обчислювальної геометрії, пов'язаних з алгоритмами породження (генерації) на скінчених множинах точок на площині геометричних об'єктів, що «спираються» на всі точки заданої множини та розділяють своєю границею площину на дві області – відкриту зовнішню та замкнену внутрішню. Оскільки велика кількість практичних застосувань базується на дискретних скінчених множинах точок, а геометричні об'єкти в таких застосуваннях наближено можуть бути представленими простими многокутниками (або многогранниками в просторах більшої розмірності ніж 2), то надалі, в контексті цієї роботи, саме їх будемо узагальнено називати геометричними об'єктами. Хоча здебільшого в роботі досліджуються двовимірні задачі, у деяких випадках, коли це є доречним і можливим, зроблені узагальнення для евклідових просторів довільної розмірності.

Щоб створити контекст для узагальненої постановки основних задач дослідження (підрозділ 1.4) і подальшого аналізу існуючих підходів до їх розв'язку (підрозділ 1.5), в даному розділі вводяться необхідні поняття, означення і припущення, та визначається модель обчислень.

Неформально суть задач породження геометричних об'єктів полягає у наступному. *На вході* деякого алгоритму маємо скінчену множину точок на площині або, більш загально, у просторі довільного виміру d. Припускаємо, що будь-які точки не колінеарні. Для просторових задач задається узагальнена умова, що жодні $d+1$ точок не лежать в одній гіперплощині. *На виході* алгоритму потрібно отримати прості многокутники (поліедри у просторі), які задовольняють певним критеріям або виконати над ними задані обчислення, наприклад, підрахувати їх кількість (*див.* Рисунок 1.1).

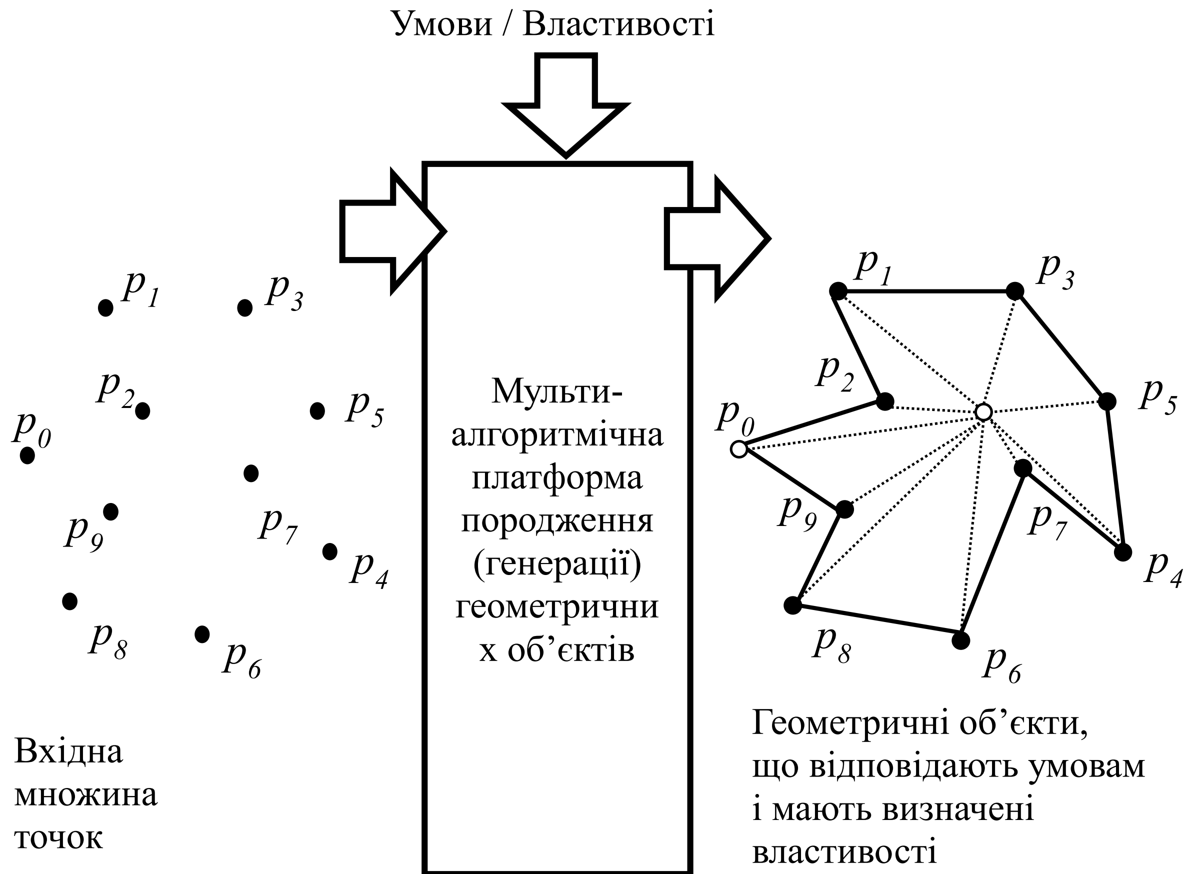


Рисунок 1.1.

1.2. Загальні означення та основні припущення.

Наведемо усі загальні означення, позначення та припущення, які надалі будуть використані в роботі. Основна нотація та поняття, що введені нижче, застосовується за роботою Препарати і Шеймоса [58], в якій систематизовано викладено основи обчислювальної геометрії, та яка цитується багатьма дослідниками алгоритмів в цієї галузі, тобто, фактично, є загальноприйнятою.

Позначимо через $\boldsymbol{E}^d$ d-вимірний евклідів простір, тобто простір d-плексів $\boldsymbol{x}$: $(x_1, x_2, \ldots, x_d)$, що складається з дійсних чисел $x_i \in \boldsymbol{R}$, $i=1, \ldots, d$, з наступною метрикою відстані:

$$\rho(\boldsymbol{x}, \boldsymbol{x}_0) = \sqrt{\sum_{i=1}^{d} (x_i - x_{0i})^2} \qquad (1.1)$$

Означення 1.1. *Точкою p* в евклідовому просторі $\boldsymbol{E}^d$ будемо називати *d*-компонентний вектор $\boldsymbol{x}$: $(x_1, x_2, \ldots, x_d)$.

Означення 1.2. Лінійна комбінація $aq_1+(1-a)q_2$, де $a \in \boldsymbol{R}$, q_1, $q_2 \in \boldsymbol{E}^d$ називається прямою.

Означення 1.3. Прямолінійним *відрізком*, що сполучає дві різних точки $q_1, q_2 \in \boldsymbol{E}^d$ будемо називати лінійну опуклу комбінацію $aq_1+(1-a)q_2$, де $a \in [0;1]$. Прийнято позначати відрізок $\overline{q_1 q_2}$.

Означення 1.4. Гіперплощиною в $\boldsymbol{E}^d$ виміру, що задається лінійно незалежними точками $q_1, \ldots, q_d \in \boldsymbol{E}^d$ називається лінійна комбінація:

$a_1q_1 + a_2q_2 + \ldots + a_{k-1}q_{k-1} + (1-a_1-\ldots-a_{d-1})q_d$, $(a_j \in \boldsymbol{R}, j=1,\ldots, d-1)$.

Означення 1.5. Область $D \in \boldsymbol{E}^d$ називатимемо опуклою, якщо для будь-якої пари точок $q_1, q_2 \in D$, відрізок $\overline{q_1 q_2}$ цілком належить D.

Примітка: Перетин опуклих областей є опуклою областю,

Означення 1.6. *Опуклою оболонкою CH(S)* множини точок $S \in \boldsymbol{E}^d$ називатимемо границю найменшої опуклою області $D(S)$ в $\boldsymbol{E}^d$, що охоплює S: (формально термін «охоплює» визначаємо як: $S \subset D(S)$).

Означення 1.7. *Многокутником* в просторі $\boldsymbol{E}^2$ (на площині) будемо називати скінчену множину відрізків, в якій кожен кінець відрізку належить рівно двом відрізкам, та ніяке з підмножин цих відрізків не має такої властивості. Відрізки, у такому випадку називаються *сторонами* (*ребрами*), а їх кінці – *вершинами* многокутника (*див.* Рисунок 1.2, А).

Означення 1.8. Многокутник називається *простим*, якщо жодна з пар його непослідовних сторін не мають спільних точок (*див.* Рисунок 1.2, Б).

Примітка: Простий многокутник розбиває площину на дві області, що не перетинаються: зовнішню – безкінечну та внутрішню – скінчену (теорема Жордана).

Означення 1.9. Простий многокутник *P* називається *опуклим*, якщо його внутрішня область є опуклою множиною.

Означення 1.10. Простий многокутник *P* називається *зірковим*, якщо існує така точка z, яка не є зовнішньою для *P*, і така, що $\forall p \in P$, відрізок $\overline{zp}$ цілком належить *P* (*див.* Рисунок 1.3, А)

Означення 1.11. Простий ланцюг називається *монотонним* (х-монотонним), якщо будь-яка вертикальна пряма перетинає його не більше ніж в одній точці. Простий многокутник, який утворений об'єднанням двох таких ланцюгів, кожен з яких зберігає вказану властивість, називається *монотонним* (*див.* Рисунок 1.3, А).

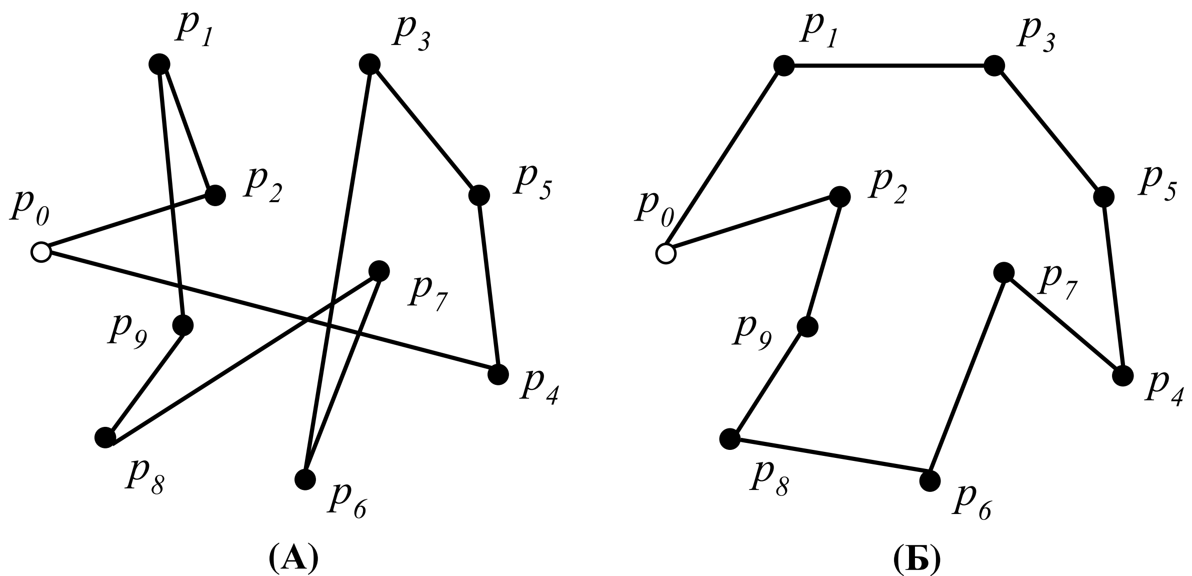


Рисунок 1.2. Приклади многокутників: А-загального вигляду, Б-простий.

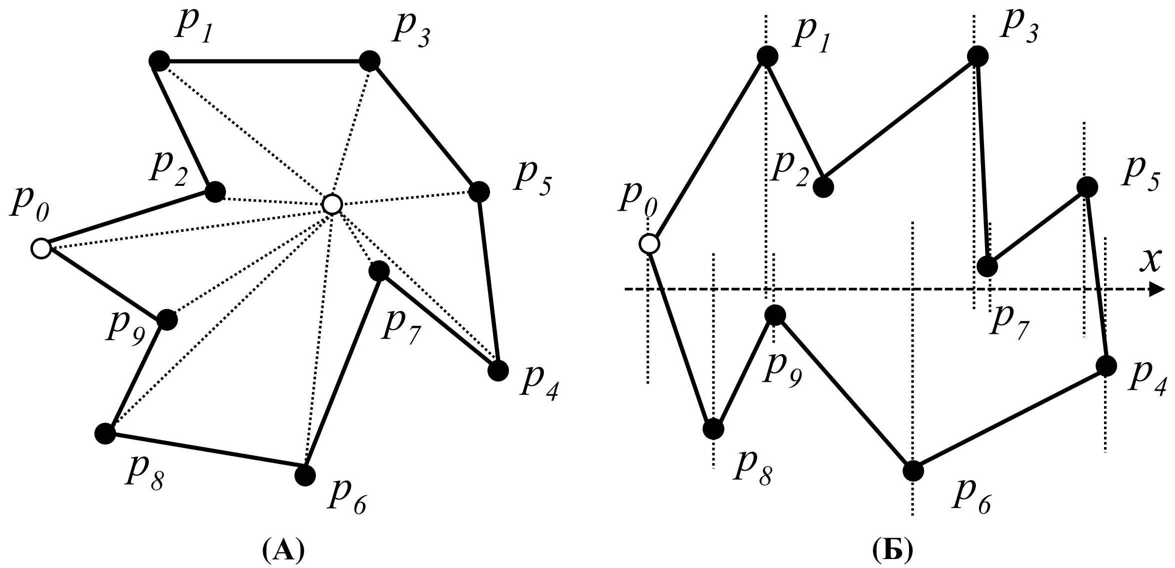


Рисунок 1.3. Приклади простих многокутників:

А- зірковий, Б - монотонний

Означення 1.12. Простий многокутник є *псевдо-зірковим* [13], якщо існує точка *p*, така, що для будь-якої точки *p'* всередині многокутника відрізок $\overline{pp'}$ перетинає границю многокутника не більше одного разу.

Означення 1.13. Простий многокутник є *спіральним* [28], якщо він має не більше одного ланцюга, що складається з увігнутих вершин, та рівно один ланцюг з опуклими вершинами.

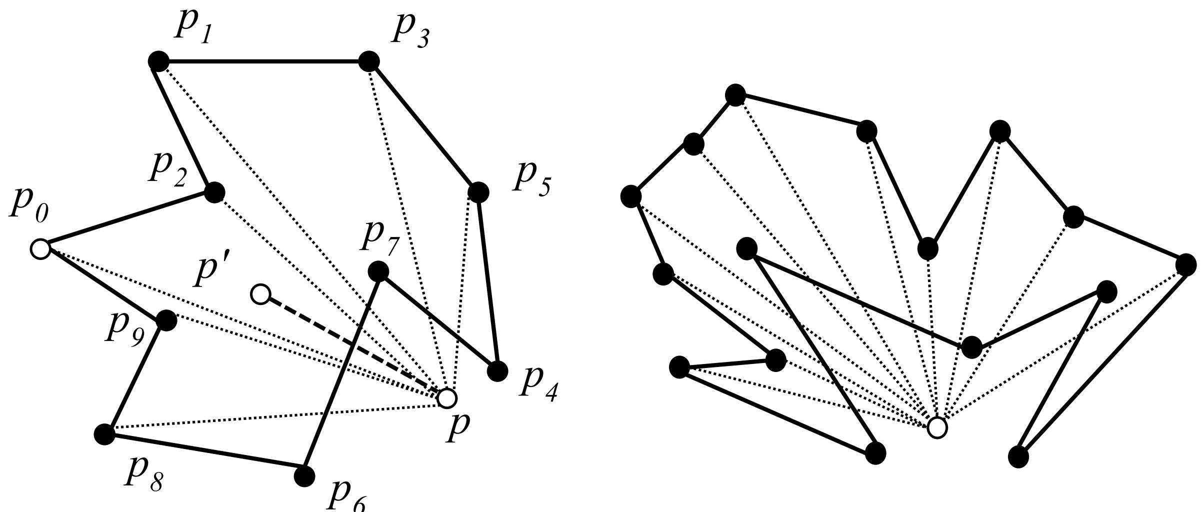


Рисунок 1.4. Приклади псевдо-зіркових многокутників

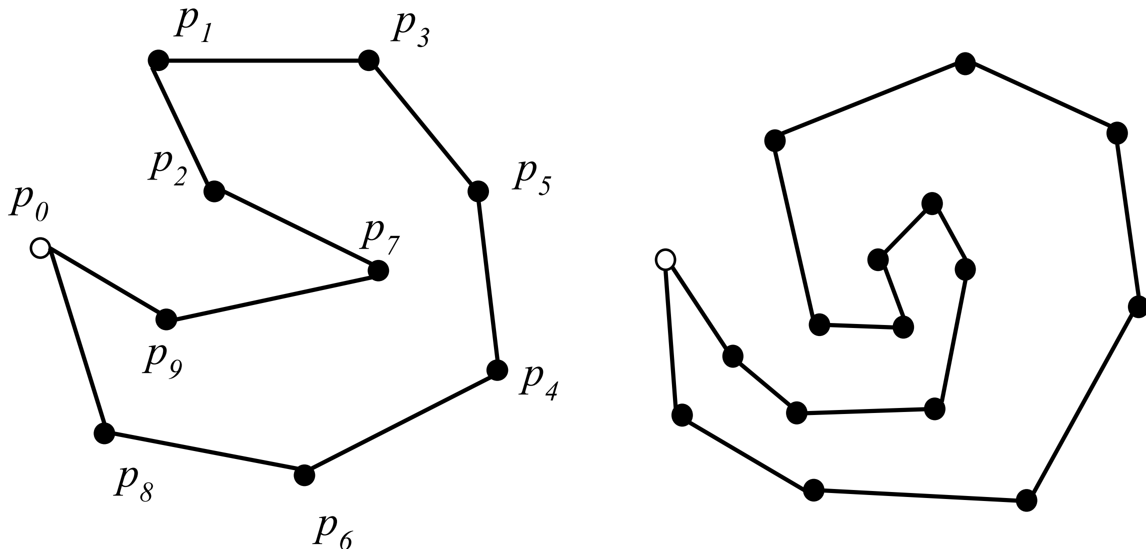


Рисунок 1.5. Приклади спіральних многокутників

1.3. Складність алгоритмів та модель обчислень

Для подальших задач нам знадобляться поняття складності алгоритмів і певна модель обчислень (*див., наприклад*, [58], []).

При аналізі та побудові алгоритмів виражають час виконання алгоритмів, як і інші міри ефективності, з точністю до мультиплікативної константи. При цьому в аналізі алгоритму враховуються лише його певні ключові операції: будь-які пропущені (невраховані) операції можуть тільки збільшувати оцінку. Однак, коли робиться таке спрощення потрібно враховувати, щоб внесок таких операцій впливав на верхні оцінки складності таким чином, щоб вони відрізнялися лише не більше ніж в константу разів від внеску усіх операцій, що виконуються алгоритмом.

Означення 1.14. Якщо N – розмір вхідних даних алгоритму розв'язку задачі T, функція $f(N)$ – час, який витрачається алгоритмом на розв'язку задачі T, гранична поведінка $f(N)$ при $N \to \infty$ називається *асимптотичною часовою складністю алгоритму*.

Будемо використовувати наступні означення для асимптотичних оцінок складності.

Так звана *О-нотація,* $O(f(N))$ (*О*-велике від $f(N)$) застосовується для позначення множини функцій, які не більше $f(N)$ з точністю до постійного числа разів – цей спосіб потрібний для опису верхніх оцінок, або більш формально:

$$O(f(N)) = \{g(N) | \exists C > 0 \wedge \exists N_0 > 0: |g(N)| \leq Cf(N), \forall N>N_0 \}$$

Відповідно, Ω-нотація використовується для позначення множини функцій, які не менше ніж у деяке постійне число разів більше ніж $f(N)$. Така асимптотична оцінка складності використовується для нижніх оцінок складності алгоритмів:

$$\Omega(f(N)) = \{g(N) | \exists C > 0 \wedge \exists N_0 > 0: g(N) \geq Cf(N), \forall N>N_0 \}.$$

Крім цього, для описування оптимальних алгоритмів, потрібні функції того ж самого порядку, що і $f(N)$, у такому випадку використовується θ-нотація:

$$\theta(f(N)) = \{g(N) | \exists C_1, C_2 > 0 \wedge \exists N_0 > 0: C_1 f(N) \leq g(N) \leq C_2 f(N), \forall N>N_0 \}.$$

Означення 1.15. Модель обчислення визначає набір елементарних операцій над даними і вартість цих операцій. Відповідно, елементарні операції – це операції з фіксованою (константною) вартістю.

Примітка: вартість різних елементарних операцій може відрізнятися одна від одної.

В роботі використовується модель RAM. Ця модель передбачає, що у кожній комірці пам'яті може зберігатися єдине дійсне число, при цьому над числами може виконуватися наступний набір операцій:

- арифметичні операції: +, -, *, /;
- операції порівняння двох чисел: <, <=, =, >=, >;
- непряме адресування пам'яті цілим числом;
- логічні, алгебраїчні та тригонометричні операції (за необхідністю).

Часова складність у гіршому випадку RAM-програми – це функція $f(N)$, що дорівнює найбільшій із сум часів (за всіма вхідними даними розміру N), які витрачені на кожну команду.

1.4. Узагальнена постановка задач

Розглянемо скінчену множину S з N точок $p_i=(x_i, y_i)$, $i=0, ..., N-1$ на площині ($S\in \mathbf{E}^2$): $S = \{p_0, p_2, ..., p_{N-1}\}$, де $N>2$. Зробимо спрощуюче припущення, що усі точки множини знаходяться у, так званому, загальному розташуванні: будь-які три точки множини S не належать спільній прямій.

Примітка: оскільки в більшості практичних застосувань використовується наближене представлення координат точок за допомогою раціональних чисел обмеженої точності, то має місце таке припущення: якщо деякі три точки належать одній прямій, можна змінити координати будь-якої з них на деяке невелике число δ у межах заданої точності ε для задачі: $p_i=(x_i, y_i) \rightarrow p_i` = (x_i+\delta, y_i)$: $\delta<\varepsilon$ та задовільнити вимогі загального розташування.

Без зменшення загальності вважатимемо, що нумерація точок в S починається з точки, яка має найменшу координату x, а у випадку, коли є дві точки зі співпадаючою найменшою координатою x, то і найменшу координату y: три чи більше точок з однаковою координатою х не може існувати за зробленим вище припущенням загального розташування точок.

Позначимо через $P(S)$ такий простий многокутник, у якого усі точки заданої вхідної множини S являються його вершинами. Будемо також казати, що у цьому випадку многокутник породжений множиною S або, що він спирається на множину точок S.

Відповідно, через $SP(S) = \{P_1(S), ..., P_M(S)\}$ позначимо множину усіх можливих простих многокутників для даної множини точок.

Теорема 1.1. $\forall S\ \exists P(S)$: для будь-якої множини точок S на площині, що задовольняє зробленим вище припущенням, завжди існує принаймні один простий многокутник, що спирається на всі точки множини S.

Хоча доведення Теореми 1.1 відоме принаймні з 1964 [46], нижче, у Розділі 4, ми наведемо варіант доведення та відповідний алгоритм побудови простого многокутника, що має узагальнення для евклідового простору $\boldsymbol{E}^d$ довільного виміру $d \geq 2$.

Зауваження: якщо $\sigma=\{\sigma(0), \sigma(1), ..., \sigma(N-1)\}$ перестановка індексів точок S, то усі можливі перестановки індексів однозначно визначають $N!$ різних варіантів з'єднання точок відрізками у замкнені цикли, тобто у многокутники загального вигляду (див. Означення 1.7). Передбачається, що цикл замикається з останньої на першу точку перестановки, тобто многокутник утворений множиною відрізків $\{\overline{p_{\pi(0)}p_{\pi(1)}}, \overline{p_{\pi(1)}p_{\pi(2)}}, \dots, \overline{p_{\pi(N-2)}p_{\pi(N-1)}}, \overline{p_{\pi(N-1)}p_{\pi(0)}}\}$. Але, по-перше, для кожного многокутника існує N-1 інших варіантів послідовностей, які геометрично еквівалентні з точністю до циклічного зсуву номерів послідовності точок по позиціях «еталонній перестановці» $\sigma=\{0, \sigma(1), ..., \sigma(N-1)\}$, *в якій* на першій позиції знаходиться точка p_0. Тому кількість перестановок, що дають різні многокутники, з урахуванням двох напрямків обходу дорівнює $(N-1)!$. По-друге, для кожної перестановки, що задає обхід точок за напрямком руху годинникової стрілки, існує відповідна «дзеркальна» перестановка, що задає обхід точок у протилежному напрямку $\sigma'=\{0, \sigma(N-1), ..., \sigma(2)\}$. Тому, зробивши припущення, що відрізки, які утворюються парами точок, неорієнтовані, та позначивши многокутник, які породжується перестановкою σ як $P(S, \sigma)$ можемо сформулювати наступне твердження.

Твердження 1.1. Кількість геометрично різних многокутників $P(S, \sigma) = \{p_0, p_{\sigma(1)}, ..., p_{\sigma(N-1)}\}$ загального вигляду, що можуть спиратися на множину S з N точок ($N>2$) дорівнює $(N-1)!/2$.

Очевидно, що не всі породжені многокутники будуть простими, оскільки у загальному випадку серед можливих перестановок будуть зустрічатися такі, в яких ребра перетинаються на своїх внутрішніх точках (*див.* Рисунок 1.2, А), тому справедливе наступне.

Твердження 1.2. $M(S) = |SP(S)| \leq (N-1)!/2\ (\forall N>2)$.

Зауважимо, що оскільки еталонна перестановка $\sigma=\{0, \sigma(1), \ldots, \sigma(N-1)\}$ однозначно задає многокутник у геометричному сенсі, то ми можемо використовувати такі перестановки (послідовності з $N-1$ індексів j точок p_j: $1<j\leq N$) для однозначної ідентифікації многокутників серед інших для заданої множини точок S.

Сформулюємо також декілька простих, але корисних у подальшому, тверджень.

Твердження 1.3. Опуклою оболонкою $CH(S)$ скінченої множини точок $S = \{p_0, p_2, \ldots, p_{N-1}\}$, де $N>2$ є опуклий многокутник.

Твердження 1.4. Якщо $S \in CH(S)$, то $M(S) = |SP(S)| = 1\ (\forall N>2)$.

Твердження 1.5. Якщо $\exists p \in S: p \notin CH(S)$, то $M(S) = |SP(S)| > 1\ (\forall N>3)$.

Тобто, у загальному випадку, коли усі точки вхідної множини S лежать на опуклій оболонці, то можливим єдиним простим многокутником є сама опукла оболонка (опуклий многокутник - згідно Означенню 1.9); якщо ж частина точок множини S не лежить на опуклій оболонці $CH(S)$, та $N>3$, то кількість можливих простих многокутників $M(S)$ для S більша ніж 1. Очевидно, що при цьому точки S, що не належать опуклій оболонці, є «внутрішніми» точками S. Визначимо такі точки формально:

Означення 1.16. Підмножиною внутрішніх точок $Interior(S)$ множини $S=\{p_1, p_2, \ldots, p_N\}$, де $N>3$, називається множина точок p: $p \in S \wedge p \notin CH(S)$. Відповідно, кожна точка $Interior(S)$ називається внутрішньою точкою S.

Примітка: підмножину внутрішніх точок можна альтернативно визначити, як $Interior(S) = S \setminus (S \cap CH(S))$.

Означення 1.17 Відбірковим предикатом для множини простих многокутників $SP(S) = \{P_1(S), \ldots, P_M(S)\}$ будемо називати функцію $F(S, P(S))$, яка приймає значення 1 («істина»), коли простий многокутник $P(S)$ задовольняє деякій властивості, та значення 0 - у протилежному випадку.

З урахуванням усього викладеного вище, тепер ми можемо сформулювати у найбільш загальній постановці основні задачі дослідження.

<u>Задача 1</u>. (*«Перелічення» або «Enumeration»*) Дано: S – вхідна множина точок на площині, *F*(*S*, *P(S)*) – відбірковий предикат для множини простих многокутників. Знайти підмножину *SP'(S)* серед усіх можливих простих многокутників $SP(S) = \{P_1(S), \ldots, P_M(S)\}$ ($SP'(S) \subset SP(S)$), для кожного з елементів такої підмножини відбірковий предикат приймає значення 1 («істина»).

<u>Задача 2</u>. (*«Підрахунок» або «Counting»*) Дано: S – вхідна множина точок на площині, *F*(*S*, *P(S)*) – відбірковий предикат для множини простих многокутників. Знайти кількість елементів підмножини $SP'(S) \subset SP(S)$, для яких відбірковий предикат приймає значення 1 («істина»).

Зауваження: Хоча на перший погляд задачі дуже схожі, але складність перелічення, як правило, виявляється більшою ніж складність задач підрахунку, які можуть використовувати деякі геометричні властивості об'єктів або деякі виродження чи результати попередніх кроків алгоритмів, що часто зменшують часову складність алгоритмів.

<u>Задача 3</u>. (*«Випадкове породження» або «Random Generation»*) Дано: *S* – вхідна множина точок на площині, *F*(*S*, *P(S)*) – відбірковий предикат для множини простих многокутників. Знайти випадкий елемент підмножини *SP'(S)* серед усіх можливих простих многокутників $SP(S) = \{P_1(S), \ldots, P_M(S)\}$ ($SP'(S) \subset SP(S)$), причому для кожного з елементів підмножини відбірковий предикат приймає значення 1 («істина»), а ймовірність дорівнює $1/|SP'(S)|$.

1.5. Аналіз існуючих підходів до розв’язку окремих випадків задач

Перш за все, наведемо історичну довідку про роботи щодо визначення нижніх та верхніх оцінок максимальної кількості простих многокутників $|SP(S)|$ для заданої кількості точок на площині N, використовуючи узагальнюючий матеріал [15].

В роботах [4]-[23] показано, що для нижньої границі справедлива наступна оцінка:

$$|SP(S)| = \Omega(b^N) \tag{1.2}$$

де b – деяка константа. Послідовне уточнення оцінки в роботах [4]-[23] відображене в Таблиці 1.1 (у хронологічному порядку).

Таблиця 1.1

Рік	**Базис *b* нижньої границі $\Omega(b^N)$**	**Посилання**
1	**2**	**3**
1979	2.27	[4]
1980	2.15	[32]
1987	3.26846179	[25]
1995	4.642	[22]
1998	3.60501960	[23]

Відповідно, для верхньої границі відомо, що:

$$|SP(S)| = O(b^N) \tag{1.3}$$

Таблиця 1.2 показує історію послідовного уточнення оцінки верхньої границі у наведених в стовпці 3 роботах.

Таблиця 1.2

Рік	**Базис *b* верхньої границі $O(b^N)$**	**Посилання**
1	**2**	**3**
1982	1013	[3]
1989	1,384,000	[22], [44]
1997	53,000	[3], [34]
1998	38,837	[22], [40]
1997	2,226	[18]
1999	1,888	[19]
1999	936	[6]
2003	199	[39]
2005	87	[43]
2009	70	[9], [41]
2011	56	[42]

Іншими словами, хоча спосіб точного підрахунку $|SP(S)|$ наразі невідомий, але відомо, що кількість простих многокутників для заданої множини з N точок на площині у загальному випадку зростає *експоненційно*.

З іншого боку, в переліку відкритих проблем обчислювальної геометрії [14], [32] питання про існування алгоритму з поліноміальною складністю для розв’язку задачі підрахунку усіх простих многокутників загального вигляду (див. Задачу 2 вище) залишається відкритим. З 2000 року ця задача фігурує як Задача 16 (Problem 16) у зазначеному списку.

В рамках проектів «Rectilinear Crossing Number Project» і «ComPoSe: Combinatorics on Point Sets and Arrangements of Objects» [2] отримані наступні результати для максимально можливої кількості простих многокутників при певній заданій кількості точок N (див. Таблиця 1.3)

Таблиця 1.3

Кількість точок, *N*	**Максимально можлива кількість простих многокутників, P_N**	**P_N / P_{N-1}, *(наближене)***	**Примітка**
1	**2**	**3**	**4**
3	1		Точне значення
4	3	3	Точне значення
5	8	2,667	Точне значення
6	29	3,625	Точне значення
7	92	3,172	Нижня границя
8	339	3,685	Нижня границя
9	1282	3,782	Нижня границя
10	4994	3,895	Нижня границя

До сьогоднішнього дня автори намагаються вирішувати різні варіанти Задачі 2 для окремих типів простих многокутників, тобто різних відбіркових предикатів в термінах, що введені вище.

Є два основних напрямки досліджень: перший – розглядати деякі евристики, які дозволяють породжувати прості многокутники загального типу другий – це вивчення певних типів простих многокутників: зіркових, псевдо-зіркових, монотонних, монотонних за кутом, «соняшникових», спіральних та інших.

Наприклад, в публікації [7] розглядається 5 різних евристичних алгоритмів для породження простих многокутників: *Steady Growth, Space Partitioning, Permute & Reject, 2-opt Moves, Incremental Construction & Backtracking*.

Евристика *Steady Growth* передбачає почергове додавання точок до замкненого простого многокутника, спочатку утвореного трьома довільно

обраними точками. Точки додаються наступним чином. На кожній ітерації обирається довільне ребро, яке видно з однієї із ще вільних точок. Причому в трикутнику, який утворюється цим ребром та обраною точкою, не повинно бути жодної з інших вільних точок. Автор показує, що таке ребро і відповідна вільна точка завжди існують. Часова складність алгоритму породження випадкового многокутника $O(N^2)$.

Недоліком цієї евристики є те, що множина простих многокутників, які породжуються таким чином є лише підмножиною усіх можливих многокутників.

Проста за суттю та реалізацією евристика *Permute&Reject* може застосовуватися лише для невеликих множин точок (N<15 для обчислювальних систем на момент написання цієї роботи), якщо йдеться про задачу перелічення чи підрахунку. Основна ідея цього методу – це обчислити перестановку індексів точок і перевірити, чи отриманий таким чином ланцюг не має самоперетинів.

Очевидно, що у випадку задачі породження випадкових многокутників, евристика дає рівномірну функцію розподілу ймовірності породження будь-якого елементу з *SP(S)*. Але її основним недоліком є недетермінована кількість невдалих спроб, потрібних для отримання кожного наступного многокутника.

Дещо кращою, але не на багато, є ситуація з евристикою *Incremental Construction & Backtracking*, основний підхід в реалізації якої полягає в обході дерева можливих варіантів вибору та додавання наступної точки до ланцюга з відсіканням заздалегідь непридатних гілок (тобто точок). Відсікання робиться на основі аналізу графа взаємної видимості точок та зводиться до перевірки необхідних умов існування простого многокутника. Хоча покращена версія цього алгоритму (Підрозділ 3.5) розширює можливості аналізу множин точок розміром до $N=20$, але має ті самі недоліки, що і *Permute&Reject*: час виконання алгоритму до знаходження наступного простого многокутника недетермінований.

Не дивлячись на те, що обидва підходи *Permute&Reject* та *Construction & Backtracking* мають суттєві обмеження та недоліки, вони мають велике значення для первинної перевірки інших алгоритмів, принаймні, на множинах з малою кількістю точок $N<20$.

Space Partitioning базується на підході «Розділяй і володарюй»: рекурсивно вхідна множина S розділяється на підмножини, які мають роз'єднанні опуклі оболонки, які потім, при виході з рекурсії об'єднуються в єдиний простий многокутник. Хоча цей алгоритм має детермінований час виконання і виконується у гіршому випадку за $O(N^2)$, він не забезпечує повного покриття *SP(S)*.

Єдиним більш-менш практичним алгоритмом, відомим на сьогодні, є *2-opt Moves*. Він забезпечує повне покриття *SP(S),* але, на жаль, не забезпечує рівномірної ймовірності породження різних многокутників: деякі многокутники можуть бути отримані декількома ланцюгами перетворень типу *2-opt*, а деякі – унікальною єдиною послідовністю [55]. Основна ідея підходу полягає в наступному. Спочатку генерується довільна перестановка індексів точок вхідної множини. Відрізки отриманого ланцюга перевіряється на взаємні перетини та створюється список усіх таких перетинів. До тих пір, поки цей список не порожній, серед четвірок точок, що належать двом відрізкам, які перетинаються, повторюється процедура заміни вершин відрізків: в чотирикутнику, що утворюється відрізками «діагоналі» замінюються на «протилежні сторони».

У другому напрямку досліджень, через накладені умови на вигляд многокутників, задача має певну виродженість і це може різко зменшити можливу кількість многокутників, і, відповідно, уможливити алгоритм з поліноміальним часом для її розв'язку.

Проаналізуємо декілька відомих на сьогодні робіт у цьому напрямку.

Зіркові многокутники. В [16] розглядаються алгоритми перелічення, а в [45] і [7], як перелічення, так породження випадкових *зіркових* многокутників. Зокрема в [16] пропонується алгоритм, що будує усі можливі

зіркові многокутники з часовою складністю $O(N^5)$ та який потребує $O(N^4)$ пам'яті. В публікації [7] розглядається алгоритм, який має часову складність $O(N^5 logN)$ та потребує $O(N^2+Nk)$ пам'яті, де k – кількість зіркових многокутників, які потрібно зберігати в. пам'яті.

В роботі [45] немає посилання на [16] (тільки на [7]): наведений алгоритм підрахунку зіркових многокутників має часову складність $O(N^5 logN)$ та декілька кращі вимоги до пам'яті: $O(N)$. Також, в [45] додатково розглянуто задачу породження випадкового зіркового многокутника з рівномірним розподіленням ймовірності: очікуваний час виконання алгоритму $O(N^2 logN)$, при цьому він вимагає $O(N)$ пам'яті, що робить його ближче до практичних застосувань, ніж [16] та [7].

Зауважимо, що загальним для вказаних алгоритмів породження зіркових многокутників є попереднє обчислення відповідних областей еквівалентності, які розглядаються далі.

Монотонні многокутники. В роботі [55] наводиться як алгоритм точного підрахунку, так і алгоритм породження випадкових монотонних многокутників.

Примітка: зауважимо, що в [55] доведено, що якщо можливо порахувати прості многокутники, то їх можна породжувати з рівномірною ймовірністю.

Алгоритм [55] підрахунку усі монотонних многокутників, вершинами яких є усі точки вхідної множини S потребує $O(N)$ пам'яті і має часову складність $O(K)$, де $N<K<N^2$ – це кількість ребер графа видимості монотонного ланцюга на S. Після підрахунку, породження випадкового многокутника потребує $O(N)$ часу.

Спіральні многокутники. Вперше спіральні многокутники згадуються в [28], де наведено алгоритм, що має часову складність $O(NlogN)$ *і* потребує $O(N)$ пам'яті. Майже одночасно в [47] представлений подібний алгоритм з такими ж оцінками складності та пам'яті.

Автором дисертації в публікації [63] породження спіральних многокутників розглянуте як окремий випадок більш загального алгоритму нарощування опуклих оболонок.

Подібний до запропонованого в [63] алгоритм з’явився декілька пізніше в [1], але саме, як вузький (не узагальнений) варіант, спеціалізований тільки для породження спіральних многокутників.

«Соняшникові» многокутники. Даний тип простих многокутників був вперше описаний в роботі [31]. Формальне означення в [31] відсутнє, автори описали лише процедуру породження соняшникового (sunflower) многокутника. Вона полягає у наступному. Для заданої множині точок будуються шари вкладених опуклих оболонок (convex hull layers); обирається найменший внутрішній шар; з його вершин уздовж сторін випускаються промені, які поділяють решту точок на окремі області. Точки в кожній області впорядковуються за полярним кутом між променем області та променем на дану точку і з’єднуються відрізками у встановленому порядку (див. Рисунок 1.6).

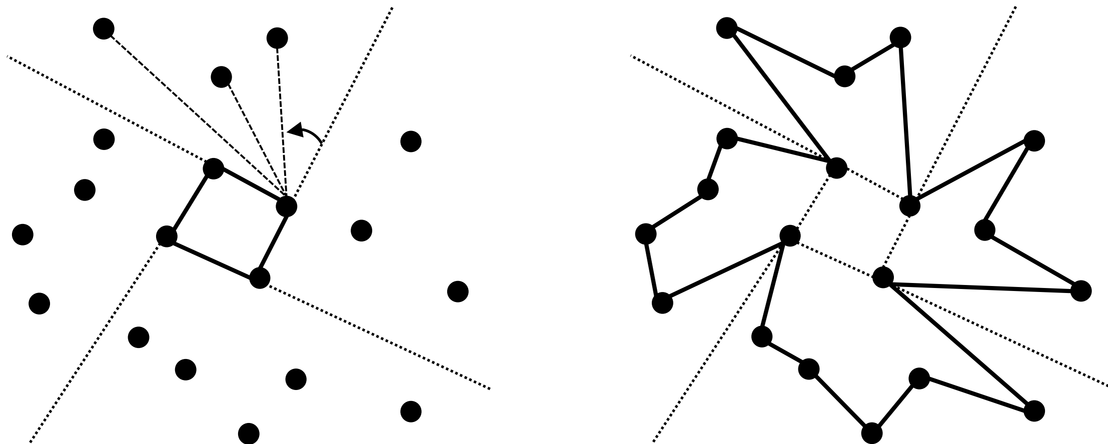


Рисунок 1.6. Процедура породження соняшникового многокутника

Висновки до Розділу 1

Підсумуємо відомості щодо простих многокутників у вигляді наступної таблиці (див. Таблиця 1.4), яка надає найкращі оцінки, що відомі на момент написання цієї дисертації.

Таблиця 1.4

Тип простого многокутника, основне посилання	**Складність перевірки на належність типу**	**Міцність множини**	**Часова складність породження випадкового екземпляру**	**Часова складність підрахунку кількості /пам'ять**
1	**2**	**3**	**4**	**5**
Загальний, [7]	$O(N)$	$O(b^N)$, $\Omega(b^N)$	$O(N^4 logN)$	Відкрита проблема
Зірковий, [45]	$O(N)$	$O(N^3)$	$O(N^2 logN)$ $O(N)$	$O(N^5 logN)$ $O(N)$
Монотонний, [55]	$O(N)$	$O(5^{(N-2)/2})$	$O(N)$	$O(N)$ $O(K)$, $N<K<N^2$
Псевдо-зірковий, [13]	$O(N)$	Невідомо	Невідомо	$O(N^2)$
Спіральний, [28]	$O(N)$	$O(1)$	-	Тільки два екземпляри
Соняшниковий, [31]		$O(1)$	-	Тільки два екземпляри

З Таблиці 1.4 можна зробити наступні висновки:

1. На сьогодні, практично, не існує ефективних алгоритмів для породження простих многокутників з повним покриттям множини можливих простих многокутників на заданій множині точок.
2. Більшість дослідників пропонують деякі евристики, але вони виявляються або з недетермінованим часом виконання, або мають низьку ефективність по часу виконання з ростом розміру вхідних даних, або не покривають всієї множини *SP(S)*.
3. Для породження випадкових екземплярів з повним покриттям множини *SP(S)* наразі існує тільки один метод, що має складність $O(N^4 logN)$.
4. З окремих типів многокутників більш-менш опрацьованими можна вважати тільки зіркові та монотонні.
5. Через те, що спіральні та соняшникові прості многокутники мають тільки по два екземпляри, автор вважає більш доцільним класифікувати їх скоріше як геометричні інваріанти заданої множини точок, ніж, як окремі класи простих многокутників.
6. Відомий лише один наближений алгоритм знаходження простого многокутника з мінімальною площиною – методи пошуку оптимальних многокутників дослідниками системно, практично, не розглядалися.

Враховуючи викладене вище в Розділі 1, можна зробити висновок, що задачі 1-3, пов’язані з породженням таких геометричних об’єктів, як прості многокутники, залишаються актуальними.

РОЗДІЛ 2. АПАРАТ ДЛЯ АНАЛІЗУ ВХІДНИХ ДАНИХ ТА РОЗВ'ЯЗАННЯ ЗАДАЧ ПОРОДЖЕННЯ ГЕОМЕТРИЧНИХ ОБЄКТІВ

2.1 Вступні відомості

У цьому розділі розглянемо декілька типів об'єктів, які важливі для подальшого викладення методів породження простих многокутників: діаграми еквівалентності (*equivalence diagrams*), що базуються на розташуваннях (*arrangements*), типи розташування (*order types*), та граф взаємної видимості точок на площині (*mutual point visibility graph*). Почнемо з неформального представлення перелічених сутностей.

Припустимо, що ми розглядаємо побудову зіркових многокутників для деякої скінченої множини точок $S = \{p_0, p_2, \ldots, p_{N-1}\}$ на площині, наприклад, таку, як на наступному рисунку (рис.2.1).

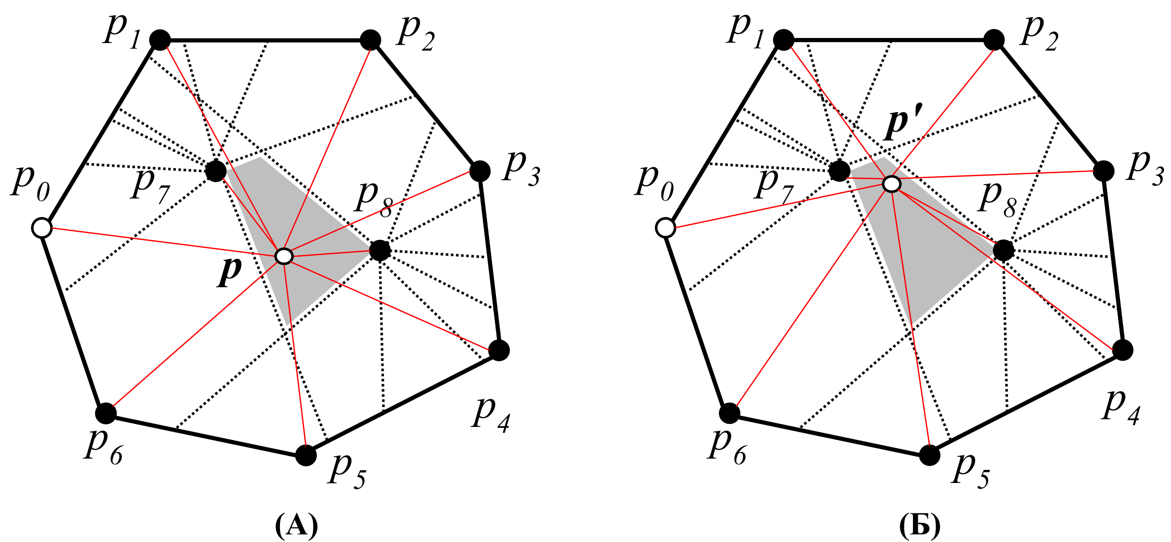


Рисунок 2.1. Приклад діаграми еквівалентності зіркових многокутників

Якщо ми побудуємо зірковий многокутник, узявши точки у порядку зростання полярного кута променів, які випущені з точки p на лівому рисунку (А), то отримаємо наступну послідовність вершин: $\{p_0, p_7, p_1, p_2, p_3, p_8, p_4,$

p_5 , p_6}. Але, у той самий час, узявши будь-яку іншу точку p’, що належить області, яка показана на обох рисунках (А) і (Б) сірим кольором, та повторимо побудову зіркового многокутника, то зірковий многокутник буде такий самий: {p_0, p_7, p_1 , p_2 , p_3 , p_8 , p_4 , p_5 , p_6}. Тобто, сіра область визначає деяку множину точок на площині, відносно яких процедура побудови зіркового многокутника дає однакові результати: такі області будемо називати *областями еквівалентності*.

Зауважимо, що усі області, що показані на рисунку, отримані таким чином: береться кожна можлива пара різних точок вхідної множини p_i, p_j; з кожної точки пари випускається промінь: $R(p_i, p_j)$ і $R(p_j, p_i)$ у напрямках уздовж прямої $L(p_i, p_j)$ на протилежну від відрізку $S(p_i, p_j)$ сторону і знаходиться перетин з опуклою оболонкою множини *CH(S)*. У результаті ми отримуємо множину відрізків, що належать многокутнику, обмеженому опуклою оболонкою *CH(S)* та вершин самої опуклої оболонки. Подібні множини відрізків, прямих ліній, променів прийнято називати *розташуваннями*. Якщо ми знайдемо усі точки перетинів таких відрізків одне з одним, то можемо побудувати усі області, які є нічим іншим, як областями еквівалентності для процедури породження зіркового многокутника. Множина таких областей називається *діаграмою еквівалентності* зіркових многокутників. Очевидно, що між діаграмою еквівалентності і множиною усіх можливих зіркових многокутників є однозначна відповідність – кількість областей і є кількістю можливих зіркових многокутників.

Щоб представити ще одну сутність – *тип розташування*, поглянемо спочатку на приклад на наступному рисунку 2.2.

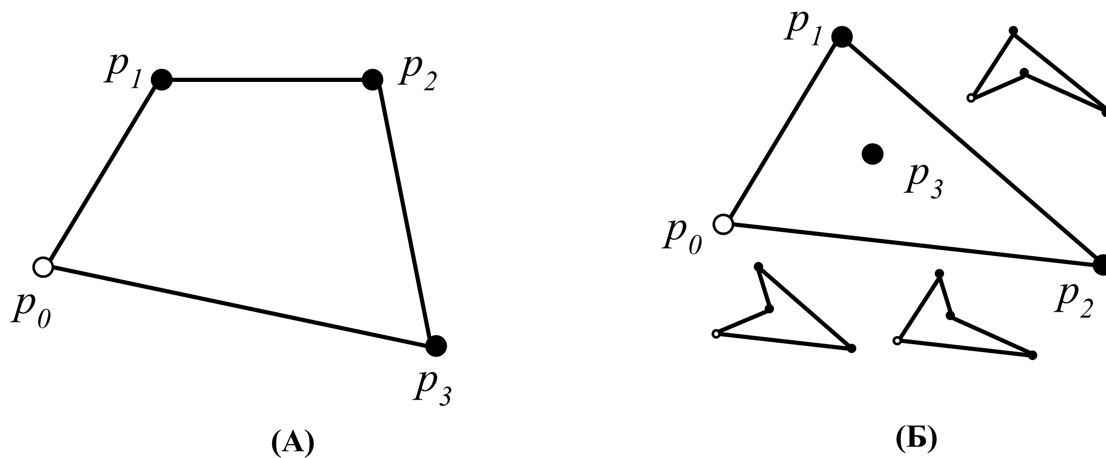


Рисунок 2.2. Два можливих типи розташування для N=4

Для будь-якої множини з чотирьох точок (*N=4*) є усього два можливих варіанти взаємного розташування точок, які з точністю до довільних афінних перетворень площини описують суттєво різні комбінаторні класи: чотирьохкутник і трикутник з точкою всередині. Ці варіанти відрізняються тим, що породжують різну кількість можливих простих многокутників: конфігурація точок (А) – тільки один, а (Б) – 3 можливих простих многокутника (відображені поряд у зменшеному масштабі).

Узагальнимо на довільну кількість точок: *тип розташування* однозначно визначає комбінаторні властивості заданої множини точок щодо породження простих многокутників. Хоча такі прості многокутники і відрізняються розмірами та розташуванням, але вони можуть бути суміщені один з іншим деякою послідовністю перетворень: масштабуванням, переносом, поворотами; але при цьому такі перетворення повинні зберігати тип кута (гострий чи тупий) між будь-якою трійкою точок множини. Тут доречно нагадати, що за нашим припущенням, жодна трійка точок вхідної множини не належить одній прямій.

Також очевидно, що тип розташування визначає, які відрізки, що утворені усіма можливими парами точок і в якому порядку перетинають інші відрізки.

Розглянемо ще один корисний для подальшого інструмент – *граф взаємної видимості точок*. Припустимо, що ми плануємо вивчати прості ланцюги, включаючи замкнені, тобто прості многокутники. З'єднуємо відрізками усі можливі пари точок вхідної множини та отримуємо повний граф (див. Рисунок 2.2. А).

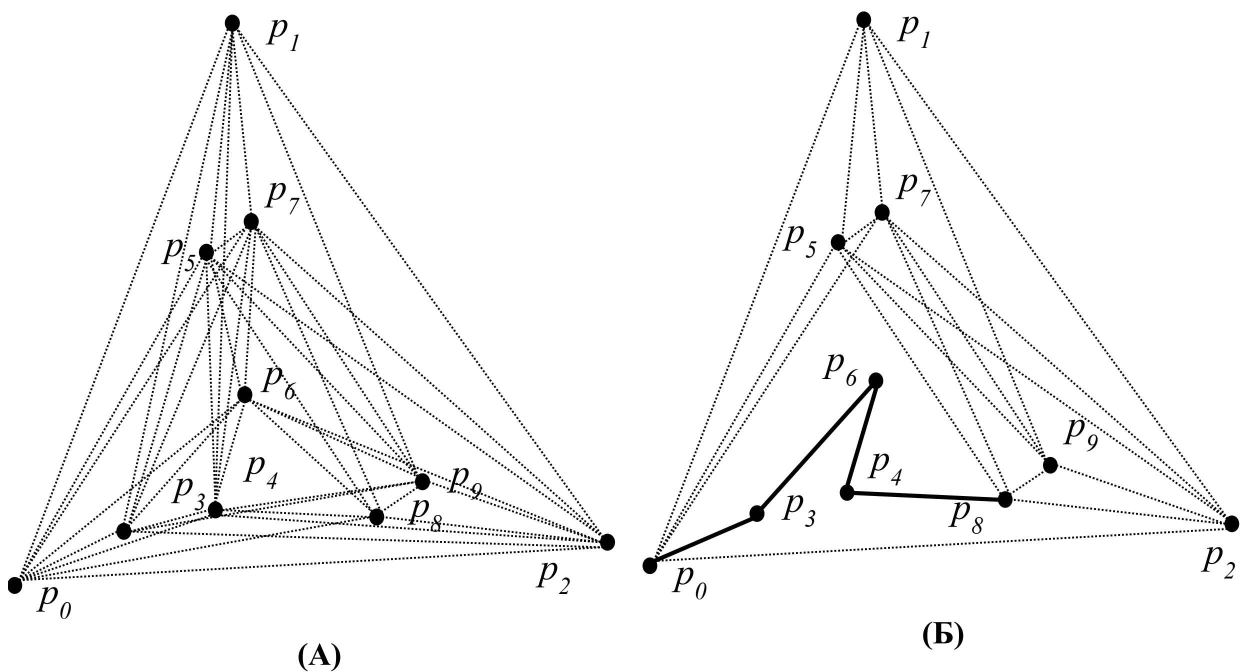


Рисунок 2.3. Приклад графа взаємної видимості (вільних) точок

Далі будуємо простий ланцюг, додаючи точку за точкою. Нехай точка p_i – остання додана до ланцюга точка. При цьому, ті відрізки, які перетинають наш простий ланцюг, а також починаються на його вершинах, окрім початкової p_0 та останньої доданої видаляються, і граф стає більш розрідженим (Рисунок 2.2, Б). Видаляється також і відрізок, який з'єднує останню додану точку p_i і початкову точку p_0 (на рисунку 2.2 Б це відрізок p_0p_8) окрім випадка коли це останній відрізок перед замиканням простого ланцюга і отримання простого многокутника.

Очевидно, що така конструкція описує взаємну видимість наступних точок: початкової вершини простого ланцюга p_0, останньої доданої до ланцюга точки p_i і точок, які ще не увійшли до його складу (тобто, вільних точок).

Попередньо зауважимо, що аналіз структури графу взаємної видимості точок дозволяє суттєво підсилити критерії відсікання дерева варіантів при вичерпному пошуку усіх можливих простих многокутників (розглядається далі).

Перейдемо до більш формального та детального опису представлених вище сутностей.

2.2 Діаграми еквівалентності зіркових розбиттів

Розглянемо деяку скінчену множина точок на площині $S = \{p_0, p_2, \ldots, p_{N-1}\}$, яка задовольняє усім умовам і припущенням з Розділу 1.

Означення 2.1. Нехай p - внутрішня точка многокутника, обмеженого $CH(S)$, причому $p \notin CH(S) \wedge p \notin S$, та $T_i(p)$ – трикутник, що визначається внутрішньою точкою p та вершинами i-го ребра $CH(S)$; *зірковим розбиттям* $\prod(S, p)$ множини точок на площині S відносно точки p є множина підмножин внутрішніх точок $Interior(S)$, які опиняються всередині того чи іншого трикутника $T_i(p)$: $\prod(S, p)=\{S_i(p): \forall i=1..h\ S_i(p) \in Ti(p) \wedge S_i(p) \subset Interior(S)\}$, де h – кількість вершин опуклої оболонки $CH(S)$. Точка p називається центром зіркового розбиття.

Тобто зірковим розбиттям множини S ми називаємо множину підмножин S, які утворюється таким чином: з довільної точки p, що лежить всередині опуклого многокутника $CH(S)$, проводимо відрізки до кожної з h вершин $CH(S)$. Оскільки точка p не належить вхідній множині і усі трійки точок з S не коленіарні, то точки S будуть належати певним трикутникам (див. Рисунок 2.1). Номер підмножини відповідає номеру трикутника, який, в свою чергу, відповідає номеру ребра опуклої оболонки. Підмножини можуть бути

порожніми. Для випадку, коли усі точки S є вершинами опуклої оболонки, усі підмножини зіркового розбиття порожні.

Наприклад, розглянемо Рисунок 2.1, А: $\prod(S, p)=\{S_0, S_1, S_2, S_3, S_4, S_5, S_6\}=\{\{p_7\}, \varnothing, \varnothing, \{p_8\}, \varnothing, \varnothing, \varnothing\}$. Відповідно, Рисунок 2.1, Б: $\prod(S, p')=\{S_0, S_1, S_2, S_3, S_4, S_5, S_6\}=\{\{p_7\}, \varnothing, \varnothing, \{p_8\}, \varnothing, \varnothing, \varnothing\}$.

Означення 2.2. Припустимо $p \neq p'$; будемо казати, що два зіркові розбиття $\prod_1(S, p)$ та $\prod_2(S, p')$ еквівалентні тоді, та тільки тоді, коли $S_i(p)=S_i(p')$, $\forall i=1..h$.

Рисунок 2.1, А і Б показує два еквівалентних розбиття. Безпосередньо з Означення 2.1 видно, що кількість елементів (підмножин) завжди однакова для будь-якої внутрішньої точки многокутника обмеженого $CH(S)$ і дорівнює кількості сторін опуклої оболонки.

Означення 2.3. Геометричне місце точок на площині $R_k(S)$ називається областю еквівалентності зіркового розбиття, якщо: $\forall p, p'$: $p\in R_k(S)$ та $p'\in R_k(S)$ відповідні зіркові розбиття $\prod_1(S, p)$ та $\prod_2(S, p')$ є еквівалентними.

Зробимо наступні позначення та припущення.

1. $\{p_{CH(1)}, \ldots, p_{CH(h)}\}\subset S$ - множина усіх вершин опуклої оболонки $CH(S)$;
2. $S\backslash CH(S)=\{p_{I(1)}, \ldots p_{I(N-h)}\}\subset S$ – множина усіх внутрішніх точок; розглядаємо тільки не вироджені випадки: $N\text{-}h>0$;
3. $r(S,i, j)$ - промінь на прямій $p_{I(i)}p_{H(j)}$, що випущений з внутрішньої точки $p_{I(i)}$ у напрямку, який протилежний вершині $p_{H(j)}$;
4. $A(S,i, j)$ – кут, що утворюється променями $r(S,i, j)$ та $r(S,i, j+1)$ або $r(S,i, 0)$, якщо $j=h$.

Лема 2.1. Області, що утворені перетином опуклої оболонки множини точок *S та множиною усіх променів* $r(S,i, j)$ $(i=1,\ldots, N\text{-}h, j=1, \ldots, h; h>2; N\text{-}h>1)$ *є опуклими многокутниками*.

Доведення: Усі кути $A(S,i, j)$ є опуклими множинами. Перетин опуклих множин – опукла множина.

Застосовуємо почергово для всіх внутрішніх точок та усіх можливих кутів наступне. Обираємо $A(S,i, j)$ та робимо перетин по парах: $A(S,i, j)\cap A(S,m,n)$, де $m\neq i$, $n=1, \ldots, h$. Кінцевий результат – непорожня опукла область, оскільки за

умовами Леми 2.1 $h>2$; $N\text{-}h>1$ та існує принаймні один промінь кута $A(S,m,n)$, що перетинає один з променів $A(S,i, j)$. Останнє витікає з того, що кожен з h променів, що виходить з кожної внутрішньої точки лежить на прямій, яка визначена цією точкою та вершиною опуклої оболонки $CH(S)$.

Перетинаючи отримані області по парах з опуклою оболонкою $CH(S)$, отримаємо обмежені опуклі області R_k(S). Тобто R_k(S) - опуклі многокутники ■.

Означення 2.4. Припустимо наступне: $R(S)=\{R_1(S),\ldots , R_K(S)\}$ – множина опуклих многокутників, яка отримана процедурою, що використана в доведенні Леми 2.1. Множину $R(S)$ будемо називати *діаграмою еквівалентності зіркових розбиттів* множини точок S.

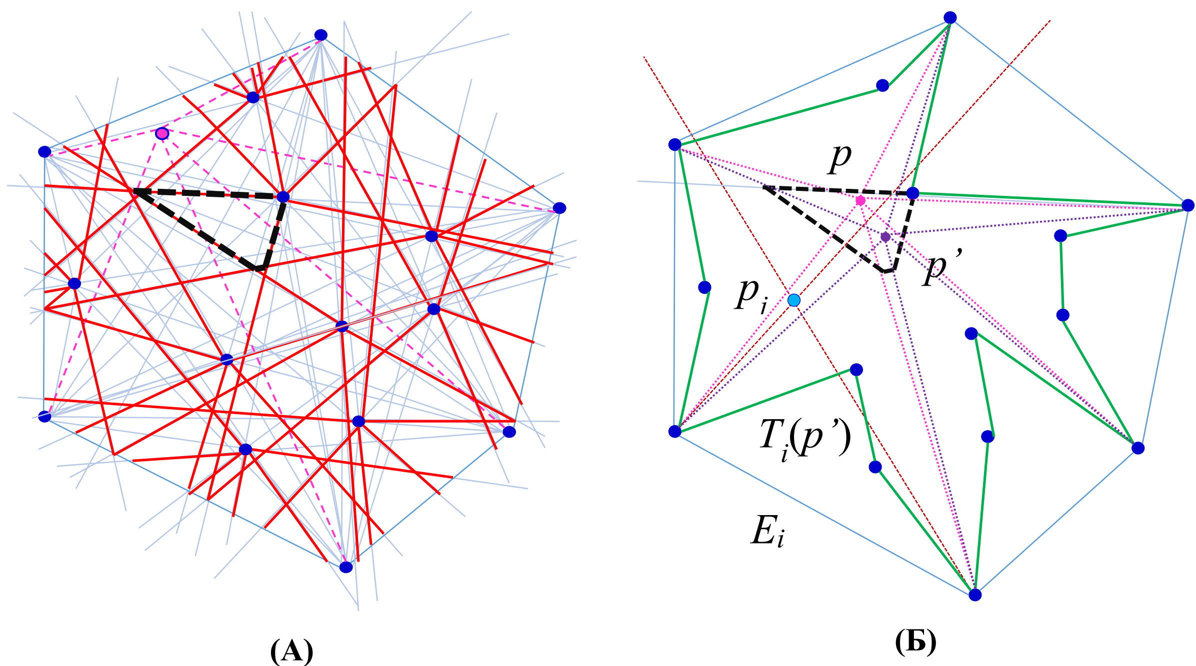


Рисунок 2.4. Допоміжна ілюстрація для доведення Теореми 2.1

Теорема 2.1. *Кожен опуклий многокутник $R_k(S)$ є областю еквівалентності зіркового розбиття $\prod_k(S, p)$ для будь-якої точки $p \in R_k(S)$.*

Доведення: Беремо опуклу область $R_k(S)$ та довільні дві точки $p\in R_k(S)$, $p'\in R_k(S)$, $p\neq p'$ та припустимо протилежне. Тоді $\prod_1(p)$ та $\prod_2(p')$ не є еквівалентними: $\exists i$: $S_i(p)\neq S_i(p')$.

Трикутники $T_i(p)\neq T_i(p')$ мають спільне *i-е* ребро опуклої оболонки $CH(S)$. Оскільки p та p' – внутрішні точки, перетин $T_i(p)\cap T_i(p')$ є не порожньою множиною (трикутником). Множини $T_i(p)\backslash T_i(p')$ та $T_i(p')\backslash T_i(p)$ є, також, трикутниками.

Єдиний спосіб задовольнити наше припущення $S_i(p)\neq S_i(p')$ – це прийняти, що $\exists p_j\in S$: $p_j\in T_i(p)\backslash T_i(p')$ або $p_j\in T_i(p')\backslash T_i(p)$. Випускаємо промені із загальних вершин трикутників $T_i(p)$ і $T_i(p')$ через точку p_j.

Точки p та p' лежать на протилежних сторонах одного з цих променів. Цей поділяючий промінь виходить з вершини опуклої оболонки $CH(S)$ через внутрішню точку p_j (див. Рисунок 2.4, Б). З Леми 2.1 витікає, що цей промінь визначає границю однієї з опуклих областей.

Це означає, що відрізок між p та p' перетинає границю опуклої області. Але це суперечить опуклості $R_k(S)$, що доводить теорему ■.

Припустимо, що обрана будь-яка внутрішня точка p області еквівалентності $R_k(S)$. Обираємо будь-який трикутник $T_i(p)$ зіркового розбиття $\prod(S, p)$. Легко бачити, якщо підмножина S_i не є порожньою (тобто, існує принаймні одна точка) тоді усі можливі прості ланцюги (тобто ті, що не мають само-перетинів), які починаються та закінчуються на кінцях ребра i, що є основою трикутника, не залежить від точки p доки вона залишається всередині області еквівалентності $R_k(S)$ (Рисунок 2.4). Ця властивість напряму витікає з означення областей еквівалентності та Теореми 2.1. Таким чином, ми можемо сформулювати наступний наслідок.

Наслідок 2.1. Припустимо, що E_i – ребро опуклої оболонки $CH(S)$; $\prod(S, p)$ – зіркове розбиття множини точок S; $R_k(S)$ – області еквівалентності: $R_k(S)\subset R(S)$; тоді множина усіх простих ланцюгів, що побудована на кінцях E_i та внутрішніх точках, та, відповідно множина усіх простих многокутників, які є об'єднанням усіх цих ланцюгів $i=1..h$, є однаковими $\forall p\in R_k(S)$.

Означення 2.5. *Розташуванням*, що породжується множиною точок S є декомпозиція (розбиття) площини на грані, ребра та вершини, які утворюються попарними прямими лініями, відрізками і променями, що визначаються усіма можливими парами точок S.

Наслідок 2.2. Діаграма еквівалентності зіркових розбиттів множини S є розташуванням (витікає безпосередньо з означень вище та Теореми 2.1).

Примітки.

Зазначимо, що $\boldsymbol{k}$ попарних перетинів $\boldsymbol{n}$ відрізків на площині можуть бути обчислені за час $O(\boldsymbol{n} log \boldsymbol{n}+\boldsymbol{k})$; час виконання є оптимальним, вимоги по пам'яті $O(\boldsymbol{n}+\boldsymbol{k})$. Ця теорема була доведена Chazelle and Edelsbrunner ([11], Theorem 5). У випадку розбиття, що розглядається, $\boldsymbol{n}=h(N-h)$.

2.3 Граф взаємної видимості вільних точок

Припустимо, що на вхідній множині точок S ми будуємо *k-й* простий многокутник, починаючи з точки p_0 і послідовно додаючи точки із тих, які можуть бути з'єднані з останньою доданою точкою без перетинів ланцюга. Нехай $C_m(S, k)$ – простий ланцюг на кроці m побудови k-го многокутника, що містить m точок з вхідної множини у якості своїх вершин, які у свою чергу є кінцями відрізків – ребер цього ланцюга: $C_m(S, k) =\{s(p_0, p_{\sigma(1)}), s(p_{\sigma(1)}, p_{\sigma(2)}), \ldots, s(p_{\sigma(m-1)}, p_{\sigma(m)})\}$, $m \leq N-1$. Нагадаємо, що простий ланцюг не має само-перетинів: жодна пара відрізків не перетинається своїми внутрішніми частинами (можливо, тільки на кінцях).

Означення 2.6. Геометричним графом називається об'єднання точок-вершин та ребер-відрізків, які з'єднують деякі з точок на площині.

Означення 2.7. *Графом взаємної видимості вільних точок* $VG_m(S, k)$ будемо називати геометричний граф, утворений на вільних точках-вершинах, які ще не входять до складу простого ланцюга $C_m(S, k)$ і кінцевих точок ланцюга p_0 та $p_{\sigma(m)}$, при цьому жоден з цих відрізків-ребер геометричного графу $VG_m(S, k)$ не перетинається з ланцюгом $C_m(S, k)$ ніде, окрім, можливо,

його кінцевих точок (див. Рисунок 2.3). Також, альтернативно будемо називати точки $p_{\sigma(m)}$ *вхідною* точкою графу $VG_m(S, k)$, а точку p_0 – *вихідною* точкою графу $VG_m(S, k)$.

На початку побудови простого многокутника граф $VG_0(S, k)$ з'єднує усі точки вхідної множини. Відрізки, що складають граф $VG_m(S, k)$ є потенційними ребрами простого многокутника. Однак, слід зауважити, що не всі відрізки, які додаються до простого ланцюга, приводять в результаті до успішного закінчення процедури побудови простого многокутника. Безпосередньо з означення витікає, що ті відрізки, які опиралися на $p_{\sigma(m-1)}$, після додавання точки $p_{\sigma(m)}$, повинні бути видалені з графу $VG_m(S, k)$.

В наступному розділі розглянемо, як за допомогою властивостей графа взаємної видимості вільних точок з'ясовується можливість чи неможливість існування Гамільтонового шляху на ньому, що включає усі вільні точки та завершує побудову замкненого простого ланцюга, тобто простого многокутника.

Висновки до Розділу 2

В даному розділі ми розглянули допоміжні об'єктів, що базуються на вхідних множинах точок і дозволяють робити аналіз конфігурацій: діаграма еквівалентності зіркових розбиттів та граф взаємної видимості точок.

Узагальнюючи, можна сказати, що діаграми еквівалентності гарантують, що для заданої множини точок і певного типу многокутників, які інваріантні до зіркових розбиттів, ми забезпечуємо вичерпний (у строгому розумінні) перелік усіх можливих варіантів.

Зауважимо, що попередній аналіз точок часто дозволяє використовувати той чи інший алгоритм в задачах, пов'язаних з простими многокутниками. Наприклад, якщо більшість точок вхідної множини опиняються на опуклій оболонці, і внутрішніх точок небагато, то таку конфігурацію інколи можна обробляти навіть переборними алгоритмами для отримання точного розв'язку.

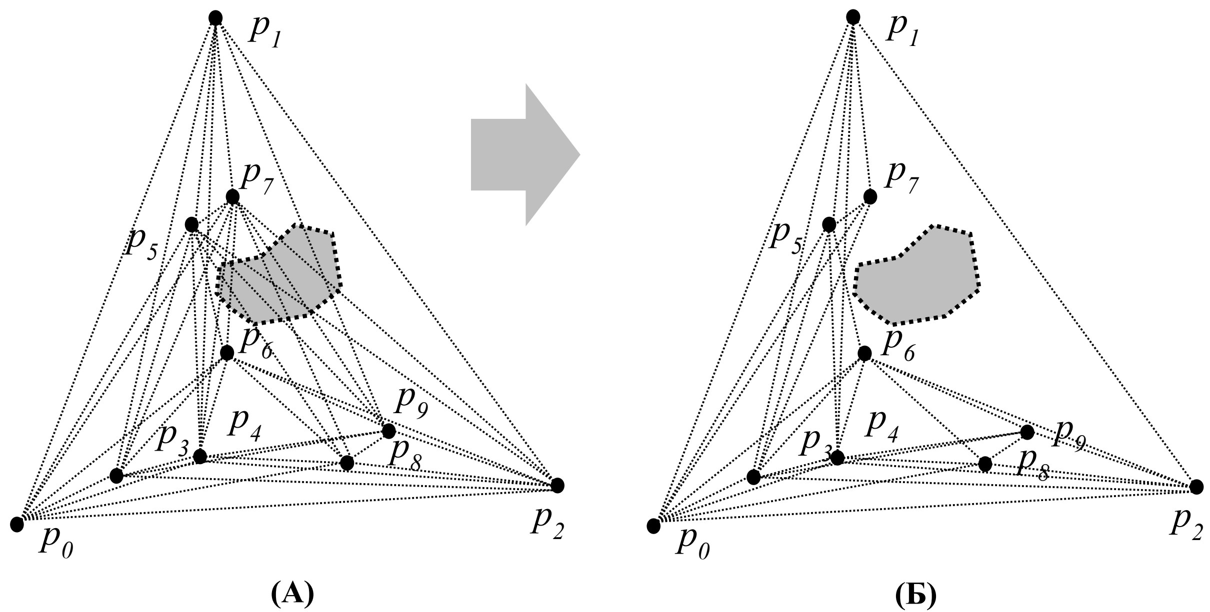


Рисунок 2.5. Вплив заборонених областей на граф взаємної видимості точок (на етапі аналізу вхідної множини точок).

Подібна ситуація із суттєвим виродженням задачі також може виникнути, якщо ми розглядаємо прості многокутники при умові існування заборонених областей, що розташовані серед вхідної множини точок. Незалежно від того, як задаються заборонені області: многокутником, аналітичною кривою, наприклад другого порядку, тощо, результат попереднього аналізу лише буде у зміні конфігурації графу взаємної видимості точок. Ті відрізки, які перетинають заборонені області, видаляються з графу видимості (див. приклад на Рисунку 2.5).

Таким чином, в окремих випадках, використовуючи відомості з теорії графів та додаткові геометричні властивості множини точок, можна суттєво знизити обчислювальну складність основних задач. Причому це можна зробити для широкого кола варіацій Задач 1-3.

Таким чином, введені в Розділі 2 допоміжні об’єкти, діаграми еквівалентності зіркових розбиттів та граф взаємної видимості точок, дозволяють зробити важливі узагальнення, які будуть розглянуті далі.

РОЗДІЛ 3. МЕТОДИ ПОРОДЖЕННЯ ДОВІЛЬНИХ ПРОСТИХ МНОГОКУТНИКІВ

3.1 Доцільність оптимізації методів з повним перебором

Перед тим, як детально розглянути декілька методів повного (вичерпного) пошуку перебором усіх можливих варіантів простих многокутників на заданій множині точок S, зробимо декілька попередніх зауважень.

Як зазначалося вище, у Розділі 1 (див. Таблиця 1.1, Таблиця 1.2) відомо, що *кількість* можливих простих многокутників у загальному випадку довільного (невиродженого – див. далі) розташування точок *експоненційно зростає* з розміром вхідної множини *S*.

Виключенням є лише вироджені варіанти: а) коли усі точки *S* розташовані на опуклій оболонці - $|SP(S)|=1$ (лише один многокутник, який співпадає з опуклою оболонкою); б) всі точки на оболонці, одна точка знаходиться у внутрішній частині многокутника, який обмежений опуклою оболонкою вхідної множини точок *CH(S)* - $|SP(S)|=N$ (по одному многокутнику на кожне ребро опуклої оболонки – див. Рисунок 3.1 нижче); в) і, можливо, варіант декількох точок у внутрішній частині многокутника *CH(S)*.

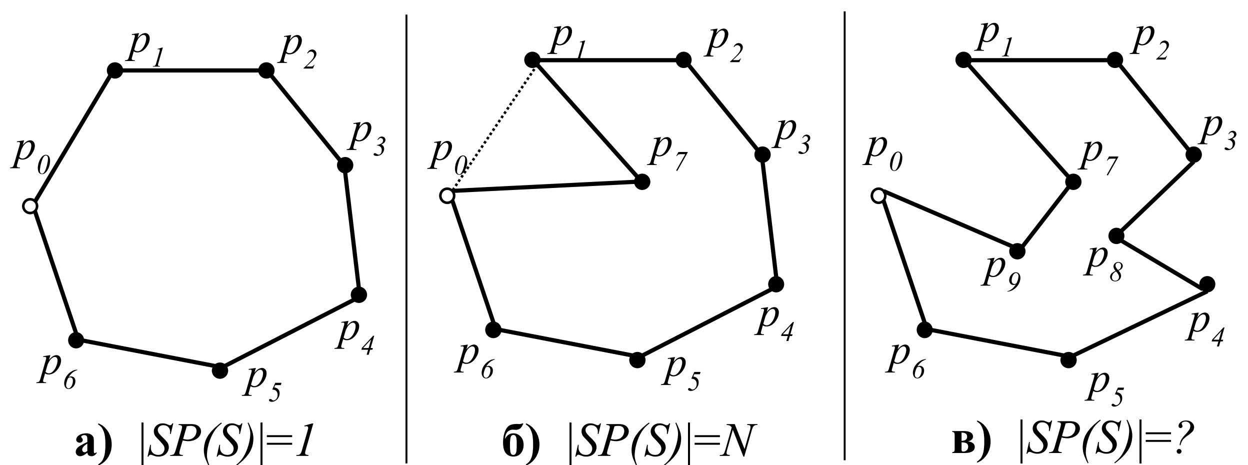


Рисунок 3.1. Приклади вироджених варіантів вхідної множини S.

Зауваження. Варіант в), практично, ніким не досліджений і *представляє окремий теоретичний інтерес*. Питання полягає у тому, при якій абсолютній кількості точок та при якому співвідношенні кількості точок на опуклій оболонці і внутрішніх кількість можливих многокутників залежність *|SP(S)|* від N перестає бути поліноміальною та стає експоненційною.

Основною мотивацією для уваги до оптимізації алгоритмів з повним перебором є можливість перевірки нових алгоритмів на коректність, принаймні, на відносно невеликих множинах точок ($N<20$).

Зрозуміло, що чим більший розмір вхідної множини точок ми можемо використовувати для перевірок, там швидше працює алгоритм перевірки, та тим більш надійною ми можемо вважати реалізацію деякого нового алгоритму генерації многокутників.

Тому надалі, окрім теоретичних викладок для доведення коректності алгоритмів, ми застосовуємо верифікацію практичної реалізації наступних алгоритмів за допомогою попередніх, вже перевірених. Такий підхід гарантує коректність як теоретичної частини даної роботи, практичної реалізації та висновків, які базуються на експериментальних перевірках.

3.2 Повний перебір з перевіркою самоперетинів

Найпростішим базовим підходом до розв'язку Задач 1-3 є породження перестановок індексів точок вхідної множини виду $\sigma_k=\{0, \sigma(1), \ldots, \sigma(N-1)\}$, яка однозначно визначає ланцюг C_N з відрізків, утворених попарним з'єднанням точок з відповідними індексами $C_N=\{\overline{p_{\pi(0)}p_{\pi(1)}}, \overline{p_{\pi(1)}p_{\pi(2)}}, \ldots, \overline{p_{\pi(N-2)}p_{\pi(N-1)}}, \overline{p_{\pi(N-1)}p_{\pi(0)}}\}$, та попарна перевірка усіх N відрізків на перетини з іншими. Якщо перетинів немає, то ми маємо простий многокутник. Хоча метод досить очевидний, вперше в публікаціях згадується відносно недавно [7].

Відповідний рекурсивний алгоритм представлено нижче (Рисунок 3.2).

```
typedef std::vector<Point> Points;
Points points; //вхідна множина точок
size_t pidx[MAX_POINTS]; //індекси точок вхідної множини
std::vector<Points> polygons; //вихідна множина простих многокутників

//процедура перевірки ланцюга на самоперетини
bool chainHasSelfIntersection(const size_t chainSize)
{
    if (chainSize <= 2) return false;
    size_t index;
    for (size_t i = 0; i < chainSize - 1; i++){
        for (size_t j = i + 1; j < chainSize; j++){
            index = (j == chainSize - 1) ? 0 : j + 1;
            if(segmentsCrossInsideBounds(
                 pidx[i], pidx[i + 1], //перший відрізок
                 pidx[j], pidx[index])) //другий відрізок
                return true;
        }
    }
    return false;
}
//рекурсивна процедура породження усіх можливих перестановок
//індексів точок з відбором тих, які утворюють прості многокутники
void makeAllPermutations (size_t m){
        if (m == points.size()){
            if (!chainHasSelfIntersection(m)){
                    Points polygon;
                    for (size_t i = 0; i<points.size(); i++)
                        polygon.push_back(points[pidx[i]]);
                    polygons.push_back(polygon);
            }
        }
        else{
            for (size_t i = m; i < points.size(); i++){
                if (i == m){
                    makeAllPermutations (m + 1);
                }
```

```
                else{
                    size_t v = pidx[m];
                    pidx[m] = pidx[i];
                    pidx[i] = v;
                    makeAllPermutations (m + 1);
                    v = pidx[m];
                    pidx[m] = pidx[i];
                    pidx[i] = v;
                }
            }
        }
    }
}
```

Рисунок 3.2. Рекурсивний алгоритм повного перебору усіх перестановок з перевіркою відрізків ланцюга на самоперетини. Примітка: процедура segmentsCrossInsideBounds приймає на вхід індекси першого (1-ша пара індексів) та другого (2-га пара індексів) відрізків та повертає значення true, якщо відрізки перетинаються в точці, яка лежить всередині кожного з них.

Теорема 3.1. Алгоритм рішення Задач 1 і 2 методом повного перебору усіх перестановок з перевіркою самоперетинів після отриманої перестановки має часову складність $O(N!)$ і потребує $O(N)$ пам’яті.

Доведення. З описаної процедури безпосередньо витікає оцінка складності алгоритму повного перебору усіх перестановок з перевіркою самоперетинів: кількість перестановок становить $(N-1)!$, кількість попарних перевірок відрізків отриманого ланцюга на наявність самоперетинів у найгіршому випадку $N(N-1)/2$, відповідно: $O((N-1)!+N(N-1)/2)= O((N-1)!)+O(N^2)=O(N!)$. Кількість кроків рекурсії ніколи не перевищує N, і ми використовуємо $O(1)$ пам’яті для зберігання тимчасових змінних, тому загальне використання пам’яті не перевищує $O(N)$ ■.

Наведений алгоритм, окрім того, що він може застосовуватися лише для маленьких множин точок з *N~13*, має ще один суттєвий недолік: він породжує такі перестановки, які, хоча і мають різні послідовності індексів, але геометрично однакові (див. також про зворотні («дзеркальні») перестановки в підрозділі 1.4). Тому, у наступному підрозділі розглянемо *спосіб усунення цього недоліку*.

Зробимо зауваження щодо рішення Задачі 3 (породження випадкових простих многокутників) за допомогою наведеного алгоритму. Якщо замість проходу по циклу в рекурсивній процедурі використати випадковий вибір наступного індексу для перестановки, то ми отримаємо розв'язок Задачі 3. Однак, з одного боку, кількість можливих перестановок становить *(N-1)!*, а, з іншого, кількість простих многокутників $|SP(S)|=\Omega(a^N)$ та $|SP(S)|=O(b^N)$ (див. Розділ 1), тому, користуючись формулою Стірлінга:

$$N! \approx \sqrt{2\pi N}\left(\frac{N}{e}\right)^N \tag{3.1}$$

для відношення кількості перестановок до кількості простих многокутників отримуємо:

$$\begin{aligned}\frac{(N-1)!}{|SP(S)|} &= \Omega\left(\sqrt{2\pi(N-1)}\left(\frac{N-1}{e}\right)^{N-1}\left(\frac{1}{b}\right)^{N-1}\frac{1}{b}\right) \\ &= \Omega\left(\frac{1}{b}\sqrt{2\pi(N-1)}\left(\frac{N-1}{eb}\right)^{N-1}\right)\end{aligned} \tag{3.2}$$

Інакше кажучи, ймовірність отримати простий многокутник таким алгоритмом з ростом N за визначений час стає дуже малою: алгоритм недетермінований і тому на практиці безплідний навіть для невеликої кількості точок.

Єдиним умовним плюсом рекурсивної процедури makeAllPermutations, який ми зможемо використати у подальшому, можна вважати можливість її певної геометричної інтерпретації: кожний наступний крок рекурсії у прямому

напрямку (у «глибину»), коли ми підставляємо на m-ту позицію наступний індекс точки, відповідає послідовному додаванню цієї точки до ланцюга і утворенню його наступного відрізка.

3.3 Повний перебір з відсіканням зворотних послідовностей

Перед тим, як розглянути узагальнений алгоритм з відсіканням зворотних послідовностей (усуненням дублювання многокутників), що відповідають «дзеркальним» перестановкам і відрізняються лише напрямом обходу, звернемося до прикладу на наступному рисунку нижче, що показує усі варіанти перестановок, які можуть бути утворені на 4-х точках, та які породжують прості многокутники.

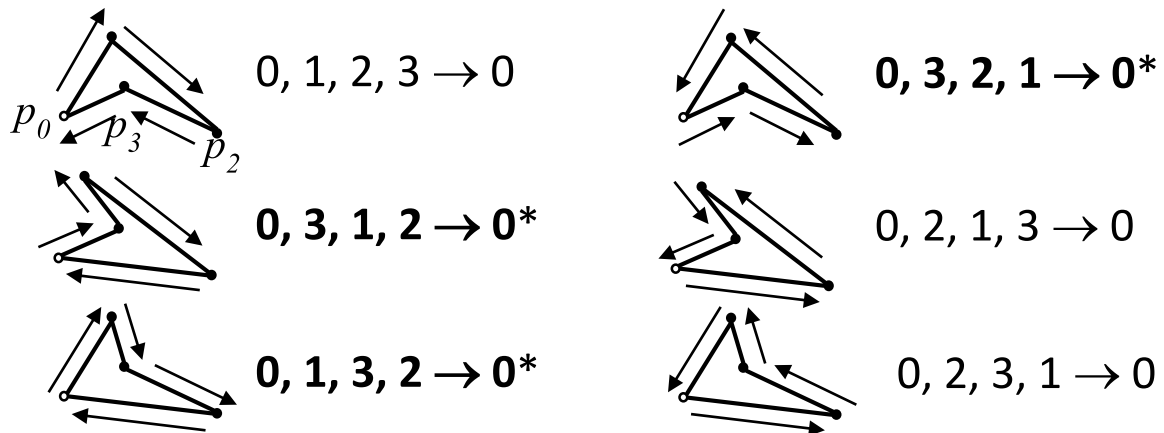


Рисунок 3.3. Усі можливі перестановки з 4-х точок, які породжують прості многокутники: тільки 3 *з них є різними у геометричному сенсі (якщо відрізки вважаються неорієнтованими).*

З наведеного прикладу видно, що у випадку чотирьох точок, одна з яких знаходиться всередині опуклої оболонки, існує шість різних перестановок індексів точок, що породжують прості многокутники (послідовності індексів

показані праворуч, поряд з відповідними многокутниками). Але тільки три з породжених простих многокутників є різними у геометричному сенсі, якщо вважати відрізки неорієнтованими.

Задача полягає у тому, щоб усунути дублювання простих многокутників, які утворені зворотними («дзеркальними») перестановками.

Не дивлячись на те, що задача відноситься до основ комбінаторики, автору не вдалося знайти повноцінного опису рекурсивного алгоритму, який, з одного боку, має корисну властивість будувати перестановку послідовно, що відповідає побудові ланцюга з точок, а, з іншого боку, усуває дублювання. Тому наведемо опис такого алгоритму.

Розглянемо множину з 5 точок: $S=\{p_0, p_1, p_2, p_3, p_4\}$. Як вже було зазначено раніше, без втрачання загальності, ми можемо обрати точку p_0, як початкову для усіх можливих перестановок–кандидатів на прості многокутники і, відповідно, нас будуть цікавити перестановки множини індексів $\{1, 2, 3, 4\}$, $N=4$.

Усю множину перестановок для кожного наступного N ми можемо отримувати, додаючи до перестановок N-1 індекс N на усі можливі позиції праворуч і ліворуч від індексу N-1.

Наприклад, наведемо усі шість можливі перестановок для $N=3$: {1, 2, 3}, {2, 1, 3}, {1, 3, 2}, {1, 3, 2}, {2, 3, 1}, {3, 2, 1}.

Додаємо індекс 4 на усі можливі позиції праворуч від 3 і отримуємо «правий» набір перестановок:

{1, 2, 3} → {1, 2, 3, **4**}

{2, 1, 3} → {2, 1, 3, **4**}

{1, 3, 2} → {1, 3, 2, **4**}, {1, 3, **4**, 2}

{2, 3, 1} → {2, 3, 1, **4**}, {2, 3, **4**, 1}

{3, 1, 2} → {3, 1, 2, **4**}, {3, 1, **4**, 2}, {3, **4**, 1, 2}

{3, 2, 1} → {3, 2, 1, **4**}, {3, 2, **4**, 1}, {3, **4**, 2, 1}

Додаємо індекс 4 на усі можливі позиції ліворуч від 3, відповідно, маємо «лівий» набір перестановок:

{1, 2, 3} → {1, 2, **4**, 3}, {1, **4**, 2, 3}, {**4**, 1, 2, 3}

{2, 1, 3} → {2, 1, **4**, 3}, {2, **4**, 1, 3}, {**4**, 2, 1, 3}

{1, 3, 2} → {1, **4**, 3, 2}, {**4**, 1, 3, 2}

{2, 3, 1} → {2, **4**, 3, 1}, {**4**, 2, 3, 1}

{3, 1, 2} → {**4**, 3, 1, 2}

{3, 2, 1} → {**4**, 3, 2, 1}

З прикладу видно, що жодна перестановка з «правого» набору не зустрічається у «лівому» і навпаки. Крім того, для кожної перестановки з «правого» набору є відповідна симетрична («дзеркальна») перестановка з індексами у зворотному напрямку обходу.

Узагальнюючі викладене вище, ми можемо модифікувати рекурсивну процедуру makeAllPermutations з попереднього розділу таким чином, щоб вона породжувала лише перестановки, наприклад, «правого» набору.

```
//рекурсивна процедура породження усіх можливих перестановок
//індексів точок з відбором тих, які утворюють прості многокутники
//Перебирає лише «праві» перестановки.
void makeAllPermutationsHalved(size_t m){
    if (m == points.size()){
            if (!chainHasSelfIntersection(m)) {
            Points polygon;
            for (size_t i = 0; i<points.size(); i++)
                polygon.push_back(points[pidx[i]]);
            polygons.push_back(polygon);
        }
    }
    else{
        for (size_t i = m; i < points.size(); i++)
        {
```

```
            if (i == m){
                makeAllPermutationsHalved(m + 1);
            }
            else{
                size_t v = pidx[m];
                if (i != MaxIdx){
                    if (pidx[i] == MaxIdxPrior) plPermuted = true;
                    pidx[m] = pidx[i];
                    pidx[i] = v;
                    makeAllPermutationsHalved(m + 1);
                    v = pidx[m];
                    pidx[m] = pidx[i];
                    pidx[i] = v;
                    if (pidx[i] == MaxIdxPrior) plPermuted = false;
                }
                else{
                    if (plPermuted) {
                        pidx[m] = pidx[i];
                        pidx[i] = v;
                        makeAllPermutationsHalved(m + 1);
                        v = pidx[m];
                        pidx[m] = pidx[i];
                        pidx[i] = v;
                    }
                }
            }
        }
    }
}
```

Рисунок 3.4. Рекурсивний алгоритм повного перебору усіх перестановок з перевіркою відрізків ланцюга на самоперетини та відсіканням зворотних («дзеркальних») перестановок. Примітки: MaxIdx – максимальний індекс (N-1), MaxIdxPrior – індекс який передує MaxIdx, тобто N-2.

З наведеного опису алгоритму видно, що ми використовуємо змінну-флаг *plPermuted*, за допомогою якої ми відслідковуємо, чи був переставлений індекс, що передує максимальному індексові (*MaxIdxPrior*). Якщо так, то ми можемо в циклі переставляти на всі позиції праворуч від нього максимальний індекс *MaxIdxPrior*. У протилежному випадку перестановка пропускається. Це і забезпечує перебір лише «правого» набору при будь-якому *N*.

Означення 3.1. *Правою перестановкою* множини послідовних натуральних чисел 1, 2, …, K будемо називати будь-яку перестановку $\sigma_K=\{\sigma(1), \sigma(2),\ldots, \sigma(K)\}$, в якій виконується наступна умова: для будь-яких порядкових номерів *i* та *j*, $\sigma(i)=K-1$ та $\sigma(j)=K$, тоді та тільки тоді, коли, $i<j$.

Лема 3.1. Рекурсивний алгоритм *makeAllPermutationsHalved* забезпечує перебір усіх $(N-1)!/2$ можливих правих перестановок індексів точок, начальна послідовність яких $0, 1, 2, 3, \ldots, N-1$ (масив pidx) і індекс 0 завжди на першій позиції.

Доведення. Безпосередньо витікає з рекурсивності описаної процедури, Означення 3.1 та зауваження про змінну-флаг *plPermuted* вище ■.

Тепер, коли ми гарантуємо обхід тільки усіх різних у геометричному сенсі перестановок, перейдемо до оптимізації шляхом відсікання зайвих гілок дерева можливих варіантів значущих правих перестановок.

3.4 Повний перебір з перевіркою ланцюга на самоперетини

По-перше, відзначимо, що наведена в попередньому підрозділі рекурсивна процедура *makeAllPermutationsHalved,* як і її прототип *makeAllPermutations,* має просту геометричну інтерпретацію: вона забезпечує послідовне додавання точок до ланцюга, що будується.

Найпростішою очевидною оптимізацією є перевірка наступного відрізку, який додається до ланцюга, на те, що він не перетинає жоден з попередньо доданих відрізків із складу ланцюга. Таки чином, процедура побудови автоматично забезпечує те, що ланцюг буде простим на кожному кроці. Хоча,

як буде показано далі на прикладах, такий спосіб відсікання (бектрекінгу) не виключає багатьох тупикових послідовностей (блокуючих ланцюгів), що не можуть бути завершені замиканням простого ланцюга в простий многокутник, але він, все рівно, дозволяє суттєво зменшити дерево обходу варіантів.

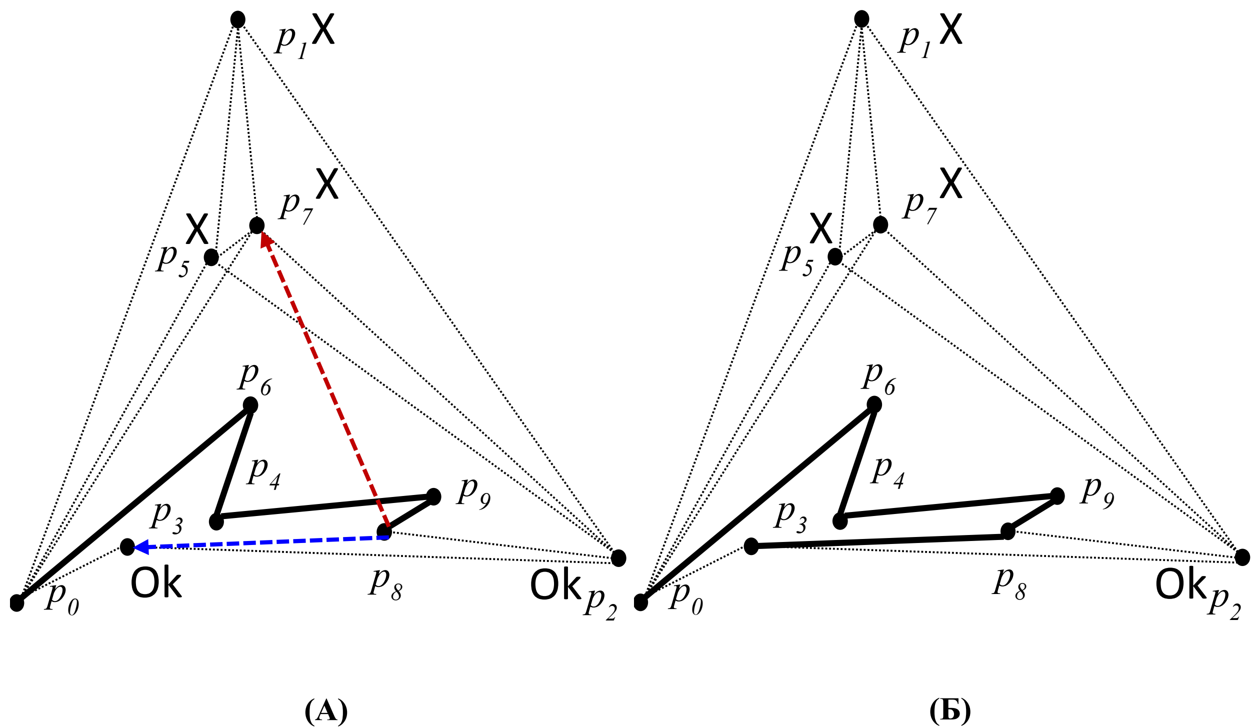


Рисунок 3.5. Пояснююча ілюстрація для алгоритму повного перебору з послідовною побудовою простого ланцюга і перевіркою на самоперетини. (А): з точки p_8 доступні точки p_3 та p_2 (відмічені “Ok”), решта точок утворюють ланцюг з самоперетином (відмічені літерою “X”). Правий рисунок (Б) показує наступний крок, після вибору p_3 для додавання у простий ланцюг.

Відповідний алгоритм реалізується за допомогою рекурсивної процедури з попереднього підрозділу додаванням відповідної перевірки (pointCanKeepSimplicityOfChainEx).

```
bool pointCanKeepSimplicityOfChainEx(
```

```
                        const size_t chainSize,
                        const size_t pIdx)
{
    if (chainSize >= 2){
        size_t lastPointIndex = chainSize - 1;
        size_t idxj = pidx[lastPointIndex];
        size_t idxidx = pidx[pIdx];

        if (segmentsCrossInsideBounds(pidx[MaxIdx], pidx[0], idxj, idxidx))
            return false;

        for (size_t i = 0; i < lastPointIndex; i++){
            size_t idxi = pidx[i];
            size_t idxi1 = pidx[i + 1];
            if (segmentsCrossInsideBounds(idxi, idxi1, idxj, idxidx))
                return false;
        }
    }
    return true;
}

//Рекурсивна процедура породження усіх можливих перестановок
//індексів точок з відбором тих, які утворюють прості многокутники
//Перебирає лише «праві» перестановки. При виборі наступного індексу
//точки перевіряє, чи буде ланцюг простий після додавання цієї точки
void makeAllPermutationsWithSimpleBacktracking(size_t m){
    if (m == points.size()){
        if (pointCanKeepSimplicityOfChainEx(MaxIdx, m)){
            Points polygon;
            for (size_t i = 0; i<points.size(); i++)
                polygon.push_back(points[pidx[i]]);
            polygons.push_back(polygon);
        }
    }
    else{
        for (size_t i = m; i < points.size(); i++){
            if (pointCanKeepSimplicityOfChainEx(m, i))
```

```
            {
                //зміст блоку є таким, як в процедурі
                //makeAllPermutationsHalved
                //з точністю до заміни рекурсивного виклику
                //на makeAllPermutationsWithSimpleBacktracking
            }
        }
    }
}
```

Рисунок 3.6. Рекурсивний алгоритм повного перебору з послідовною побудовою простого ланцюга і перевіркою на самоперетини на кожному кроці рекурсії (для бектрекінгу).

3.5 Повний перебір та граф взаємної видимості вільних точок

Як зазначалося вище, кількість можливих простих многокутників на заданій множині точок оцінюється як $|SP(S)| = O(b^N)=\Omega(a^N)$, де a, b – деякі константи. Алгоритми, які наведені в попередніх підрозділах, мають, практично, факторіальну складність. Тому поставимо, наступну ціль оптимізації – знайти алгоритм, який забезпечує розв’язок Задачі 1 і 2, принаймні, зі складністю, що наближається до $O(P(N)b^N)$, де $P(N)$ – деякий поліном від N. Відповідно, для Задачі 3 «Випадкове породження» бажано мати алгоритм з поліноміальною часовою складністю.

Матеріал, який викладений в цьому підрозділі, розширяє підхід, який вперше стисло описаний в [7] і деталізований в [8]. Автори роботи [7] користуються терміном “list of available edges” («список доступних ребер») для позначення об’єктів, які ми в даній роботі називаємо «графи взаємної видимості вільних точок» та розглядають їх поверхово. На нашу думку, графи взаємної видимості вільних точок грають фундаментальну роль в розв’язанні Задач 1 – 3, тому повинні бути вивчені більш глибоко.

Як видно з означень в підрозділі 3.1, граф взаємної видимості вільних точок є геометричним графом загального типу, який утворюється при

послідовній побудові простого ланцюга та, відповідно, при видаленні частини ребер повного геометричного графу, в якому спочатку кожна точка з’єднана з іншою ребром-відрізком, а потім в ньому видаляються ті ребра, які перетинають простий ланцюг, або дотикаються до його внутрішніх вершин.

Фактично, на кожному кроці побудови простого ланцюга, при додаванні до нього наступної точки, вирішується така задача:

Задача 4 («існування Гамільтонового шляху без самоперетинів на геометричному графі») Чи існує на геометричному графі Гамільтонів шлях, який з’єднує задані дві точки, обходячи усі інші, та не має самоперетинів (в нашому випадку це початок та кінець простого ланцюга - *див.* Рисунок 2.3 з підрозділу 2.1).

Відомо, що для графів загального типу подібна задача є NP-повною. Для повного геометричного графу довільної скінченої множини точок на площині завжди існує і простий Гамільтонів шлях, і простий цикл без самоперетинів [24]. Гамільтонів простий цикл в геометричному графі є простим многокутником.

Очевидно, що задача породження випадкового простого многокутника зводиться до Задачі 4.

Наведемо усі відомості, що прямо стосуються Задачі 4.

По-перше, для довільного геометричного графу відомо наступне.

Теорема 3.2 [10]. У кожного геометричного графа, який має n вершин та, принаймні, $\binom{n}{2} - \sqrt{\frac{n}{2}}$ ребер існує Гамільтонів шлях.

Стосовно графу взаємної видимості вільних точок справедливі наступні твердження [8].

Твердження 3.1 [8]. Якщо тільки два ребра інцидентні деякій вершині графу взаємної видимості вільних точок $VG_m(S, k)$, то обидва ребра повинні бути частиною результуючого простого многокутника $P(S, \sigma_k)$, де σ_k – відповідна k-та перестановка (це безпосередньо витікає з Означення 1.7

простого многокутника); такі ребра назвемо *обов'язковими ребрами* графу взаємної видимості вільних точок (скорочено «обов'язкові ребра»).

Твердження 3.2 [8]. Якщо точка має два обов'язкових ребра, то усі ребра, які мають дану точку за вершину, вже не можуть бути доступні для включення до простого ланцюга і, відповідно, до простого многокутника, що будується (це також безпосередньо витікає з Означення 1.7 простого многокутника).

Твердження 3.3 [8]. Усі ребра $VG_m(S, k)$, які перетинають одне з обов'язкових ребер повинні бути видалені.

Твердження 3.4. Перевірка умов та відповідні процедури прорідження графу $VG_m(S, k)$, що наведені в Твердженнях 3.1-3.3, сукупно потребують не більше $O(N^2)$ часу, якщо для кожного ребра-відрізка, який однозначно визначений довільною парою точок, в $O(N^4)$ зберігається список інших ребер-відрізків, які перетинають його всередині.

Доведення. Дійсно, максимальна кількість ребер графу $VG_m(S, k)$, що повинні бути перевірені на умови Тверджень 3.1-3.3, складає $N(N-1)/2<N^2$. При цьому $O(N^4)$ пам'яті потрібні для організації наступної структури даних: для кожного відрізка-ребра, який визначається парою індексів точок вхідної множини i, j елемент E_{ij} – це список інших відрізків-ребер, які його перетинають. Видалення ребра може бути виконане за $O(1)$, якщо список ребер організовано, як масив пар індексів кінцевих точок ребер – остання в масиві пара переставляється на місце тієї, що видаляється та лічильник елементів списку зменшується на 1. Обидві операції потребують $O(1)$ часу ■.

Наступні три леми наводяться з додатковими номерами, як вони позначаються в [8] та префіксом ТА, але формульовані в позначеннях та термінах даної дисертації без втрати їх основної суті (без доведення).

Лема 3.2 (ТА-2.9 [8]). Кожна вільна точка повинна мати принаймні два инцидентних ребра, інакше, якщо існують точки з меншою кількістю инцидентних ребер, то Гамільтонів шлях без самоперетинів між вхідною та вихідною точками не існує в $VG_m(S, k)$, і, відповідно, простий многокутник не може бути побудований.

Лема 3.3. (ТА-2.10 [8]). Тільки одна точка, яка інцидентна вхідній точці $VG_m(S, k)$ (тобто є вторим кінцем одного і того ж самого ребра, що і остання додана до простого ланцюга точка) може мати два інцидентні ребра, які доступні для включення в простий ланцюг, інакше Гамільтонів шлях без самоперетинів між вхідною та вихідною точками не існує в $VG_m(S, k)$, і, відповідно, простий многокутник не може бути побудований.

Лема 3.4 (ТА-2.11 [8]). Точки, які належать опуклій оболонці *CH(S)* повинні включатися до простого ланцюга у тій самій відносній послідовності, в якій вони розташовані на опуклій оболонці, інакше Гамільтонів шлях без самоперетинів між вхідною та вихідною точками не існує в $VG_m(S, k)$, і, відповідно, простий многокутник не може бути побудований.

Тобто, Леми ТА 2.9 – 2.11 є, фактично, необхідними умовами існування Гамільтонового шляху без самоперетинів між вхідною та вихідною точками $VG_m(S, k)$ і можуть бути використані для бектрекінгу в рекурсивній процедурі для розв’язку Задач 1-3, як це описано в [8], при цьому загальна складність усіх перевірок не перевищує $O(N^2)$.

Практична реалізація відповідного алгоритму показує, що він виявляється значно швидшим за алгоритм з попереднього Підрозділу 3.4, але, наприклад, при рішенні Задачі 3 (породження випадкових простих многокутників) виникає багато ситуацій, які приводять до непередбачуваного часу виконання.

Зауважимо, що ні Теорема 3.2, ні Леми ТА 2.9 – 2.11 не забезпечують повного охоплення усіх можливих конфігурацій графу взаємної видимості вільних точок $VG_m(S, k)$ та, відповідно, розв’язок Задачі 4 за поліноміальний час.

Більш того, перетворимо вираз для кількості ребер в Теоремі 3.2 наступним чином:

$$\binom{n}{2} - \sqrt{\frac{n}{2}} = \frac{n(n-1)}{2} - \sqrt{\frac{n}{2}} = \frac{n(n-1)}{2}\left(1 - \frac{1}{n}\sqrt{\frac{2}{n}}\right) \approx \frac{n(n-1)}{2} \text{ при } n \gg 1 \quad (3.3)$$

З (3.3.) видно, що навіть при малій кількості точок в графі $VG_m(S, k)$ достатня умова, що сформульована Теоремою 3.2, практично, непридатна, оскільки в більшості *реальних* конфігурацій точок вона не виконується (має суто теоретичний інтерес). Однак, з урахуванням того, що гранична кількість ребер для виконання умови Теореми 3.2 може бути обчислена наперед та збережена в $O(N)$ пам’яті (в масиві з довільним доступом), то питання її використання в практичній реалізації не є критичним (часова складність такої перевірки $O(1)$). Така перевірка може бути у нагоді на перших кроках виконання процедури побудови простого ланцюга $C_m(S, k)$ ($m=1, 2, 3$).

Розширимо список необхідних і достатніх умов існування Гамільтонового шляху без самоперетинів, які з’єднують вхідну і вихідну точки графу $VG_m(S, k)$.

Варто зробити зауваження, що хоча, як зазначено в [7], додавання додаткових перевірок для бектрекінгу на кожний крок рекурсивної процедури обходу дерева можливих варіантів може потенційно погіршити час виконання алгоритмів для розв’язок Задач 1 і 2, але для Задачі 3 це однозначно виправдано. Річь у тому, що кожна пропущена перевірка при породженні випадкового простого многокутника для достатньо великих N може привести алгоритм до таких конфігурацій точок, при яких час виконання стає, фактично, факторіальним від кількості вільних точок. Окрім цього, існує деякий розмір N_0 вхідної множини S, при якому стає суттєвою перевага у відсіканні тупикових варіантів для усіх $N>N_0$, навіть при тому, що розширені перевірки для бектрекінгу займають додатковий час.

Почнемо поповнення списку необхідних та достатніх умов з аналізу початкового стану графу взаємної видимості вільних точок $VG_0(S, k)$. Оскільки

$VG_0(S, k)$ – повний геометричний граф, утворений точками-вершинами S, однаковий для будь-якого k, то для спрощення будемо позначати його *VG(S)*.

В роботі [8] не врахована одна важлива спрощуюча перед-обробка графу *VG(S)* –видалення діагоналей многокутника, що утворюється вершинами опуклої оболонки. Для множин, в яких багато точок на опуклій оболонці і множина внутрішніх точок відносно невелика, це суттєво впливає на прискорення алгоритму обходу. Рисунок 3.6 показує різницю в початковому стані між графом із діагоналями та без них.

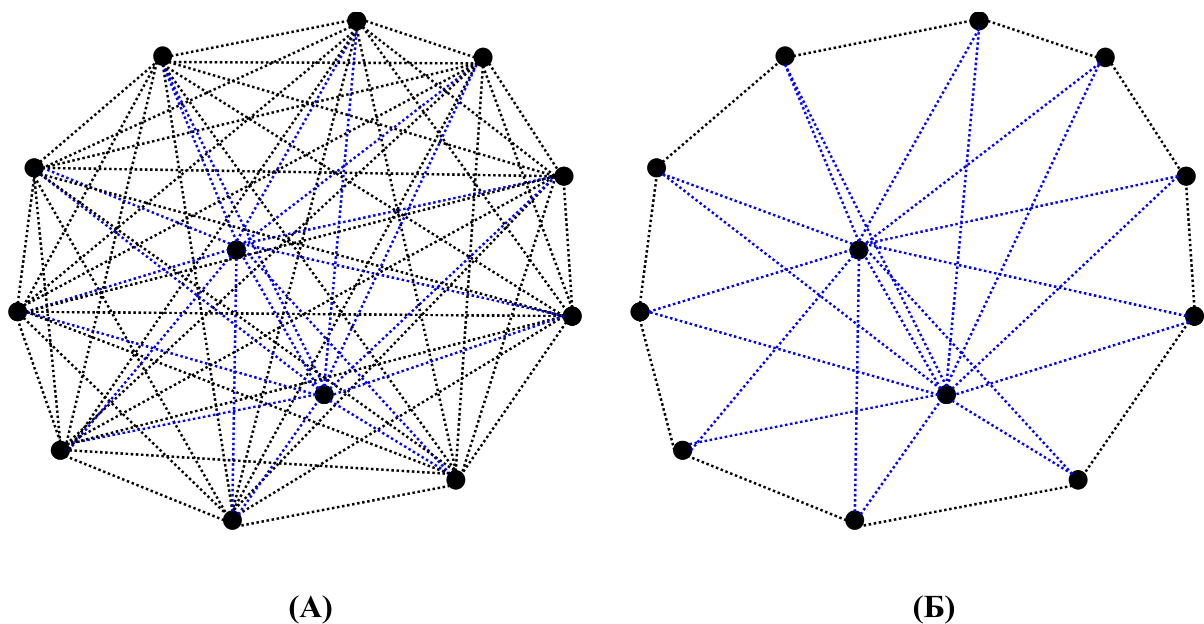


Рисунок 3.6. Приклад спрощення графу взаємної видимості вільних точок шляхом видалення його діагоналей на етапі попередньої обробки та аналізу вхідної множини точок S: (А) – початковий стан повного геометричного графу VG(S); (Б) – стан графу після видалення діагоналей – проріджений повний геометричний граф $VG_0(S)$.

Означення 3.2. Діагоналями повного геометричного графу *VG(S)* будемо називати такі ребра-відрізки, що з'єднують будь-яку пару точок-вершин опуклої оболонки *CH(S)*, які не є сусідніми одна для іншої.

Означення 3.3. *Проріджeним повним геометричним графом $VG_0(S)$* будемо називати геометричний граф, який отриманий з повного геометричного графу шляхом видалення усіх його діагоналей (див. Рисунок 3.6).

Означення 3.4. Назвемо максимальну кількість відрізків-ребер в проріджeному повному графі $VG_0(S)$, що можуть бути ребрами простого многокутника, побудованого на усіх точках S, *потенціалом E(S)* множини точок *S*.

Теорема 3.2. Діагоналі *VG(S)* не можуть бути ребрами будь-якого простого многокутника, що включає усі точки множини *S*. (Або в альтернативному формулюванні: діагоналі повного геометричного графу не можуть бути ребрами, які входять в Гамільтонів цикл без самоперетинів).

Доведення. Припустимо протилежне, що існує діагональ, яка входить до складу простого многокутника та яка з'єднує дві точки $p_i, p_j \in S$, що, у свою чергу, є несусідніми вершинами опуклої оболонки: $p_i, p_j \in CH(S)$. Із Означення 1.8 простого многокутника безпосередньо витікає, що жодна з пар непослідовних сторін не мають спільної точки. Окрім цього для простих многокутників відомо, що кожна з вершин простого многокутника, включаючи обрані точки p_i та p_j, мають два і тільки два суміжних ребра.

Відрізок-діагональ $\overline{p_i p_j}$ ділить опуклу область обмежену опуклою оболонкою *CH(S)* на дві опуклі області R_1 та R_2,(Рисунок 3.7, А), які мають спільні точки тільки на відрізку $\overline{p_i p_j}$ (тобто цей відрізок є перетином цих областей). По відношенню до ребра $\overline{p_i p_j}$ ребра нашого простого многокутника, які суміжні для точок p_i, та p_j можуть бути орієнтовані за чотирьома варіантами взаємної, та, відповідно, належності до R_1 та R_2 (Рисунок 3.7, А-Г). Розглянувши кожен з варіантів А і В без втрати загальності, та використавши той факт, що в простому многокутнику з кожної точки множини *S* можна потрапити через суміжні ребра в будь-яку іншу точку множини в обох напрямках обходу, отримуємо, що існує одне (варіант (В) і

(Г)) або два (варіант (А) і (Б)) ребра, вершини якого належать різним областям R_1 та R_2, які, у свою чергу мають спільні точки тільки на $\overline{p_i p_j}$, тобто це ребро буде мати спільну точку з відрізком $\overline{p_i p_j}$. Це суперечить тому, що многокутник простий і, відповідно, доводить те, що діагональ не може бути ребром простого многокутника, який включає усі точки множини S у якості своїх вершин ■.

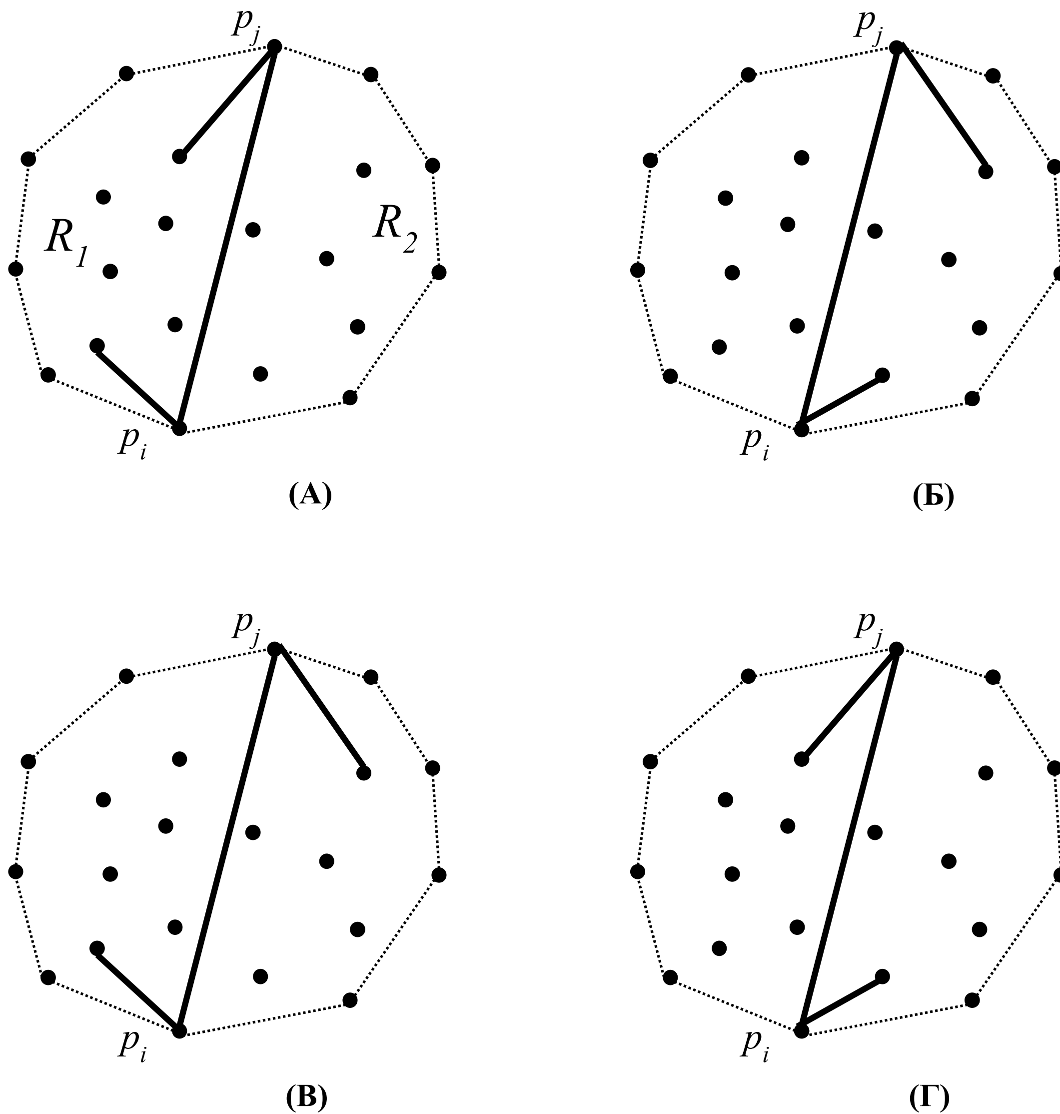


Рисунок 3.7. Ілюстрація для Доведення Теореми 3.2.

Наслідок 3.1. Якщо h – кількість вершин опуклої оболонки $CH(S)$, то для потенціалу множини з N точок S справедлива наступна формула:

$$E(S) = \frac{N(N-1) - h(h-3)}{2} \quad (3.4)$$

Доведення. Кількість ребер в *VG(S)*, повному геометричному графі, $\frac{N(N-1)}{2}$, кількість його діагоналей $\frac{h(h-1)}{2} - h = \frac{h(h-3)}{2}$. Формула (3.4) витікає безпосередньо з Теореми 3.2 та означень вище ■.

Зауважимо, що з викладеного вище можна зробити ще один важливий висновок: для тих множин, опукла оболонка яких є трикутником довершений граф співпадає з повним графом, тобто $VG(S) = VG_0(S)$. Тому оптимальні конфігурації точок, які максимізують кількість простих многокутників для заданої кількості вхідних точок, необхідно шукати саме серед них.

Для подальшого просування до ефективної процедури побудови простих многокутників з відсіканням тупикових варіантів (бектрекінгу) нам знадобляться деякі відомості з теорії графів, що пов'язані із задачами зв'язності. У якості основного джерела для означень та тверджень будемо використовувати публікацію Р. Седжвіка [60]. Оскільки геометричні графи з точки зору наступного розгляду є окремим випадком графів загального виду, то в термінології ми не будемо робити уточнень та конкретизації спеціально для геометричних графів, вважаючи, що все викладене стосується їх також.

Означення 3.5. (18.1 [60]) *Мостом* (*bridge*) в графі називається ребро, після видалення якого граф розпадається на два не пов'язаних між собою під-графи. Граф, якій не має мостів, називається реберно-зв'язаним (*edge-connected*).

Означення 3.6. (18.2 [60]) *Точка сполучення* (articulation point) графу є вершина, в разі видалення якої зв'язний граф розпадається на два графи, що не перетинаються.

Означення 3.7. (18.3 [60]) Граф називається *двозв'язним* (biconnected), якщо кожна пара вершин з'єднана двома шляхами, які не перетинаються.

Підграфи графу, які являються двозв'язними називаються *двозв'язними компонентами графу* (*biconnected components*).

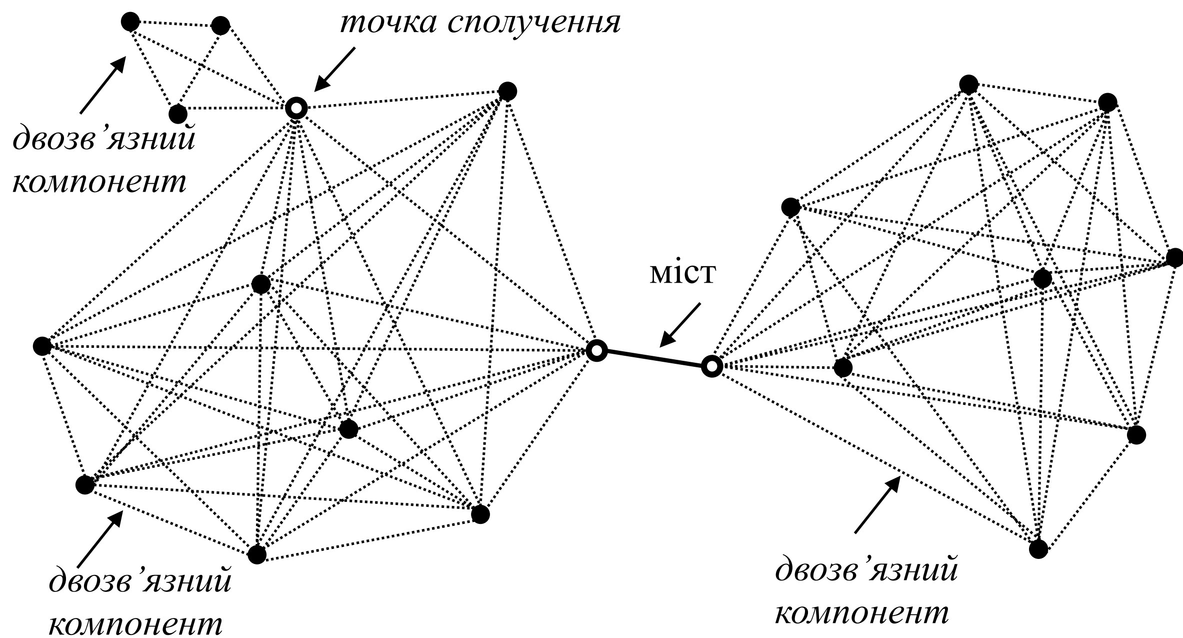


Рисунок 3.8. Приклад довільного геометричного графу з мостами, точками сполучення та двозв'язними компонентами. Примітка: кінцеві точки-вершини мостів також є точками сполучення.

Твердження 3.5. (Властивість 18.6 [60]) Мости графу знаходяться за лінійний час від суми кількості ребер та вершин.

Твердження 3.6. (Властивість 18.8 [60]) Точки сполучення та двозв'язні компоненти графу знаходяться за лінійний час від суми кількості ребер та вершин.

Тепер повернемося до процедури послідовного нарощування простого ланцюга $C_m(S, k)$ і пов'язаного з нею графу взаємної видимості вільних точок $VG_m(S, k)$, який отримується шляхом видалення відрізків-ребер, що перетинаються з простим ланцюгом $C_m(S, k)$, чи дотикаються до нього (окрім кінцевих точок).

Послідовно застосовуючи Твердження 3.1-3.3 до $VG_m(S, k)$ ми можемо створити його проріджену версію $\widetilde{VG}_m(S,k)$, в якій видалені усі ребра, про які

заздалегідь відомо, що вони не можуть входити до складу простого многокутника, що будується (див. Твердження 3.1-3.3 та Леми 3.2 і 3.3). Введемо відповідне означення.

Означення 3.8. *Проріджиним графом взаємної видимості вільних точок* $\widetilde{VG}_m(S,k)$ називається граф, який отриманий з графу взаємної видимості вільних точок $VG_m(S, k)$ шляхом видалення відрізків-ребер, що не можуть входити до складу простого ланцюга і приводити до завершення процедури побудови простого многокутника. Алгоритм, за яким видаляються такі зайві відрізки-ребра, будемо називати *процедурою проріджиння графу* $VG_m(S, k)$.

Примітка: Означення 3.8 узагальнює поняття проріджиння, що введене для повних геометричних графів, на довільний геометричний граф, що пов'язаний зі своїм простим ланцюгом через критерії видимості та придатності до використання в простому многокутнику. Зазначимо також, що деякі критерії та процедури, що викладені нижче можуть бути використані для пошуку Гамільтонових шляхів в довільних геометричних графах, а деякі суттєво залежать від вказаного контексту процедури побудови простого ланцюга.

Наведемо загальну процедуру повного перебору з використанням графу взаємної видимості вільних точок.

```
//
// Рекурсивна процедура повного перебору з послідовною побудовою
// простого ланцюга та відсіканням тупикових варіантів
// викликом процедури проріджиння та аналізу топології
// графу взаємної видимості вільних точок
// pointCanKeepSimplicityOfChain
//
void exchasitveSearchWithBacktrackingByVG(
      size_t m, BooleanArray64 pointIsVisibleFromChainHead){
    if (m == sizeOfPointSet) {
        if (pointIsVisibleFromChainHead[0]){
            //обробити отриманий простий многокутник
        }
```

```
}
else{
    size_t prevIdx = m - 1;
    pointIsInTheChain[pidx[prevIdx]] = true;
    for (size_t i = m; i < sizeOfPointSet; i++){
        size_t nextPoint = pidx[i];
        if ( pointIsVisibleFromChainHead[nextPoint] ){
            BooleanArray64 nextVisiblePoints(false);
            SafePairsArray edgesOfVisibilityGraphToRestore;
            if ( pointCanKeepSimplicityOfChain(
                 m, i, edgesOfVisibilityGraphToRestore,
                 nextVisiblePoints) ) {
                if (i == m){
                    exchasitveSearchWithBacktrackingByVG(
                        m + 1, nextVisiblePoints);
                }
                else{
                    size_t d = MaxIdx - nextPoint;
                    if ((d > 1) || (d == 0 && plPermuted)){
                        size_t v = pidx[m];
                        pidx[m] = nextPoint;
                        pidx[i] = v;
                        exchasitveSearchWithBacktrackingByVG(
                            m + 1, nextVisiblePoints);
                        pidx[i] = nextPoint;
                        pidx[m] = v;
                    }
                    else if (d == 1){
                        plPermuted = true;
                        size_t v = pidx[m];
                        pidx[m] = nextPoint;
                        pidx[i] = v;
                        exchasitveSearchWithBacktrackingByVG(
                            m + 1, nextVisiblePoints);
                        pidx[i] = nextPoint;
                        pidx[m] = v;
                        plPermuted = false;
```

```
                        }
                    }
                }//if (pointCanKeepSimplicityOfChain…))
                restoreVisibilityGraph(edgesOfVisibilityGraphToRestore);
            }//if(pointIsVisibleFromChainHead[nextPoint])       }
        pointIsInTheChain[pidx[prevIdx]] = false;
    }
}
```

Рисунок 3.9. Псевдокод алгоритму повного перебору з використанням графу взаємної видимості вільних точок.

Процедура на Рисунку 3.9 є модифікованою версією процедури на Рисунку 3.6 з попереднього підрозділу. На кожний наступний крок рекурсії ми передаємо тимчасову для кожного кроку структуру даних `pointIsVisibleFromChainHead`, яка містить інформацію про доступність точки з певним індексом для додавання її у простий ланцюг у якості наступної. Відповідно, цикл, хоча і проходить по всіх можливих індексах, на відміну від варіанту обходу з попереднього підрозділу, виконується лише для доступних точок.

Процедура `pointCanKeepSimplicityOfChain`, як і раніше, виконує перевірку, чи може точка з поточним індексом основного циклу рекурсивного обходу і надалі підтримувати ланцюг простим. Якщо відповідь «так», то виконується вхід до наступного рівня рекурсії і точка додається до ланцюга. Але основний зміст процедури `pointCanKeepSimplicityOfChain` в даному контексті повністю базується на обробці графу взаємної видимості, яка включає його прорідження та аналіз топології. Вона також готує вказану в попередньому параграфі структуру даних – масив індексів доступних точок (видимих з голови простого ланцюга). Далі розкриємо деталі цієї процедури, для чого сформулюємо усі необхідні твердження.

Нагадаємо, що питання, чи може робитися наступний крок рекурсії, залежить від того, чи існує Гамільтонів шлях на графі видимості вільних точок

між його вхідною (головою простого ланцюга) та вихідною (хвостом простого ланцюга) точками. У свою чергу, це питання тісно пов'язано із задачами про зв'язність графу.

Лема 3.5. Припустимо, що граф *$VG_m(S, k)$* містить інші вільні точки-вершини окрім вхідної та вихідної точок, тоді ребро, що з'єднує вхідну і вихідну точки не може бути ребром Гамільтонового шляху без самоперетинів між вхідною та вихідною точкам та ребром простого многокутника, що будується.

Доведення. Безпосередньо витікає з означень простого многокутника і простого Гамільтонового шляху: якщо включаєм ребро між вхідною та вихідною точками, то отримаємо замкнений простий шлях (цикл), який не є повністю охоплюючім: він включає тільки частину точок і вже ніяка точка не може бути додана до простого ланцюга ■.

Лема 3.6. Якщо вхідна точка-вершина графу взаємної видимості вільних точок (голова простого ланцюга) є точкою сполучення *$VG_m(S, k)$* (або $\widetilde{VG}_m(S, k)$), то не існує Гамільтонового шляху між вхідною та вихідною точками та, відповідно, побудова простого многокутника не може бути завершена (Рисунок 3.10, А).

Доведення. Припустимо протилежне, що вхідна точка є точкою сполучення і існує Гамільтонів шлях від вхідної точки до вихідної. Ця точка сполучення графа ділить його принаймні на дві компоненти. Для компоненти, яка не містить вихідну точку, Гамільтонів шлях без самоперетинів, за визначенням включає кожну точку-вершину цієї компоненти і має два суміжні ребра на кожній з вершин, включаючи точку сполучення. Теж саме виконується і для компоненти, яка містить вихідну точку, оскільки вхідна точка є спільною для обох компонентів. Тобто точка сполучення має принаймні три суміжних ребра, що суперечить припущенню, що шлях Гамільтонів і не має самоперетинів (кожна точка повинна мати рівно два суміжних ребра) ■.

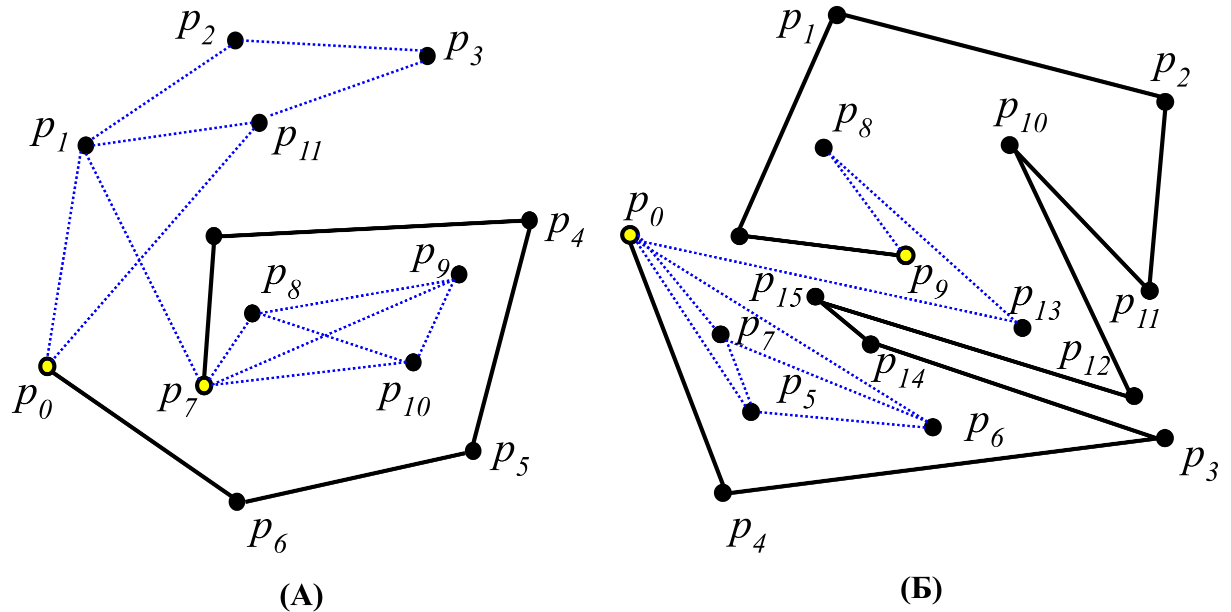


Рисунок 3.10. Приклади конфігурацій, в яких точки входу і виходу є точками сполучення (Леми 3.6 та 3.7): (А) – p_7, вхідна точка графу взаємної видимості вільних точок (голова простого ланцюга), є точкою сполучення; (Б) – p_0, вихідна точка графу взаємної видимості вільних точок (хвіст простого ланцюга), є точкою сполучення.

Лема 3.7. Якщо вихідна точка-вершина графу взаємної видимості вільних точок (хвіст простого ланцюга) є точкою сполучення $VG_m(S, k)$ (або $\widetilde{VG}_m(S,k)$), то не існує Гамільтонового шляху між вхідною та вихідною точками та, відповідно, побудова простого многокутника не може бути завершена (Рисунок 3.10, Б).

Доведення. Для вихідної точки доведення проводиться аналогічно Лемі 3.6 ■.

Означення 3.9. Припустимо, що точки входу і виходу графу взаємної видимості вільних точок $VG_m(S, k)$ не є точками сполучення $VG_m(S, k)$, тоді, якщо будь-який шлях без циклів (повторних входів на ті самі точки) від вхідної до вихідної точки проходить через деяку точку сполучення і при цьому вона є

також точкою сполучення для деякого компонента, який не містить жодне з ребер таких шляхів, то такий компонент називається *термінальним*.

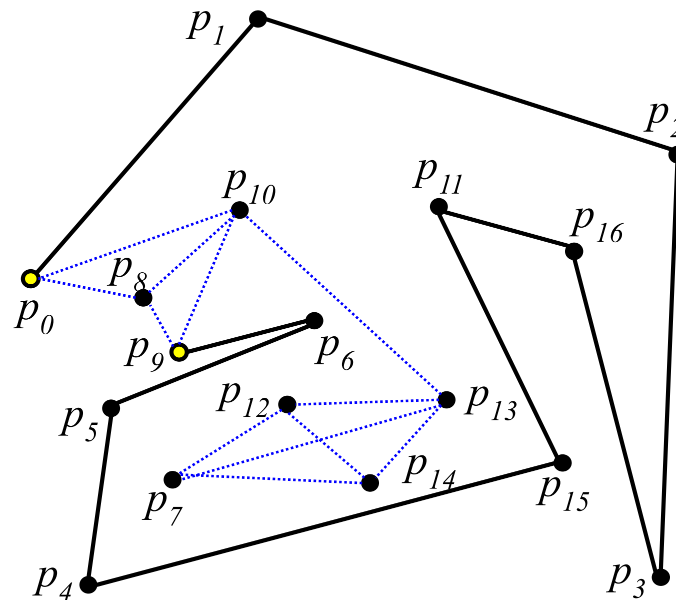


Рисунок 3.11. Приклад- ілюстрація для Означення 3.9 та Леми 3.8 конфігурації, що містить термінальний компонент, який утворюється точками p_7, p_{13}, p_{12}, p_{14}. Усі можливі шляхи без циклів, що проходять через точку сполучення p_{10}: шлях 1 - $p_9 \to p_8 \to [\boldsymbol{p_{10}}] \to p_0$; шлях 2: шлях 1 - $p_9 \to [\boldsymbol{p_{10}}] \to p_8 \to p_0$; шлях 3 (найкоротший) - $p_9 \to [\boldsymbol{p_{10}}] \to p_0$.

Лема 3.8. Якщо в графі взаємної видимості вільних точок існує термінальний компонент, то не існує Гамільтонового шляху між вхідною та вихідною точками та, відповідно, побудова простого многокутника не може бути завершена (Рисунок 3.11).

Доведення. Витікає безпосередньо з означення термінального компонента і означення Гамільтонового шляху без самоперетинів – точка сполучення, яка є єдиною загальною точкою з графом взаємної видимості вільних точок обов’язково повинна входити до складу простого ланцюга. З іншого боку, вона має, принаймні, три інцидентних ребра, які повинні бути ребрами Гамільтонового шляху без самоперетинів. Це суперечить тому, що кожна вершина такого шляху може мати тільки два суміжних ребра ■.

Рисунок 3.12 показує усі основні можливі варіанти приєднання термінальної компоненти. Зауважимо, що інші компоненти, які, у свою чергу можуть бути приєднані до термінальної компоненти, не є суттєвими з точки зору аналізу графу видимості вільних точок на його «гамільтоновість». Якщо ввести узагальнений тип вершин, який об'єднує точки сполучення, точку виходу і входу, то з усіх варіантів (А), (В)-(Д) видно, що у тому випадку, коли існує двозв'язна компонента, яка містить три вершини такого узагальненого типу, Гамільтонів шлях між точкою входу і виходу не існує.

Також, якщо існує деяка точка-вершина, яка є точкою сполучення для більше ніж двох компонент (Рисунок 3.12, Б), то це також приводить до неможливості побудови Гамільтонового шляху між точкою входу і виходу.

Означення 3.10. Точки-вершини входу і виходу, сполучення двозв'язних компонент графу взаємної видимості вільних точок $VG_m(S, k)$ будемо узагальнено називати точками *зчеплення* графу $VG_m(S, k)$.

Виходячи з цього, процедура перевірки, чи граф взаємної видимості вільних точок не містить термінальних компонент, може бути наступною.

Робимо ідентифікацію усіх двохзв'язних компонент, мостів та точок сполучення будь-яким з відомих алгоритмів. В даній роботі, наприклад, використовується модифікована реалізація алгоритму з [65], яка базується на пошуку в глибину (стандартні контейнери STL замінені на масиви фіксованого розміру). Для кожного індексу-номеру двозв'язної компоненти ведемо лічильник точок зчеплення, а для точок зчеплення – лічильник компонент, які вони зчіплюють. При цьому простий ланцюг, також, можна умовно вважати окремою двозв'язною компонентою: лічильники, які відповідають точкам-вершинам входу і виходу виставляються на початку в значення 1. Як тільки лічильник точок зчеплення для певної компоненти або лічильник компонент для точок зчеплення перевищує значення 2, то припиняємо роботу алгоритму і повертаємо результат, що Гамільтонів шлях для даного графу видимості вільних точок не існує.

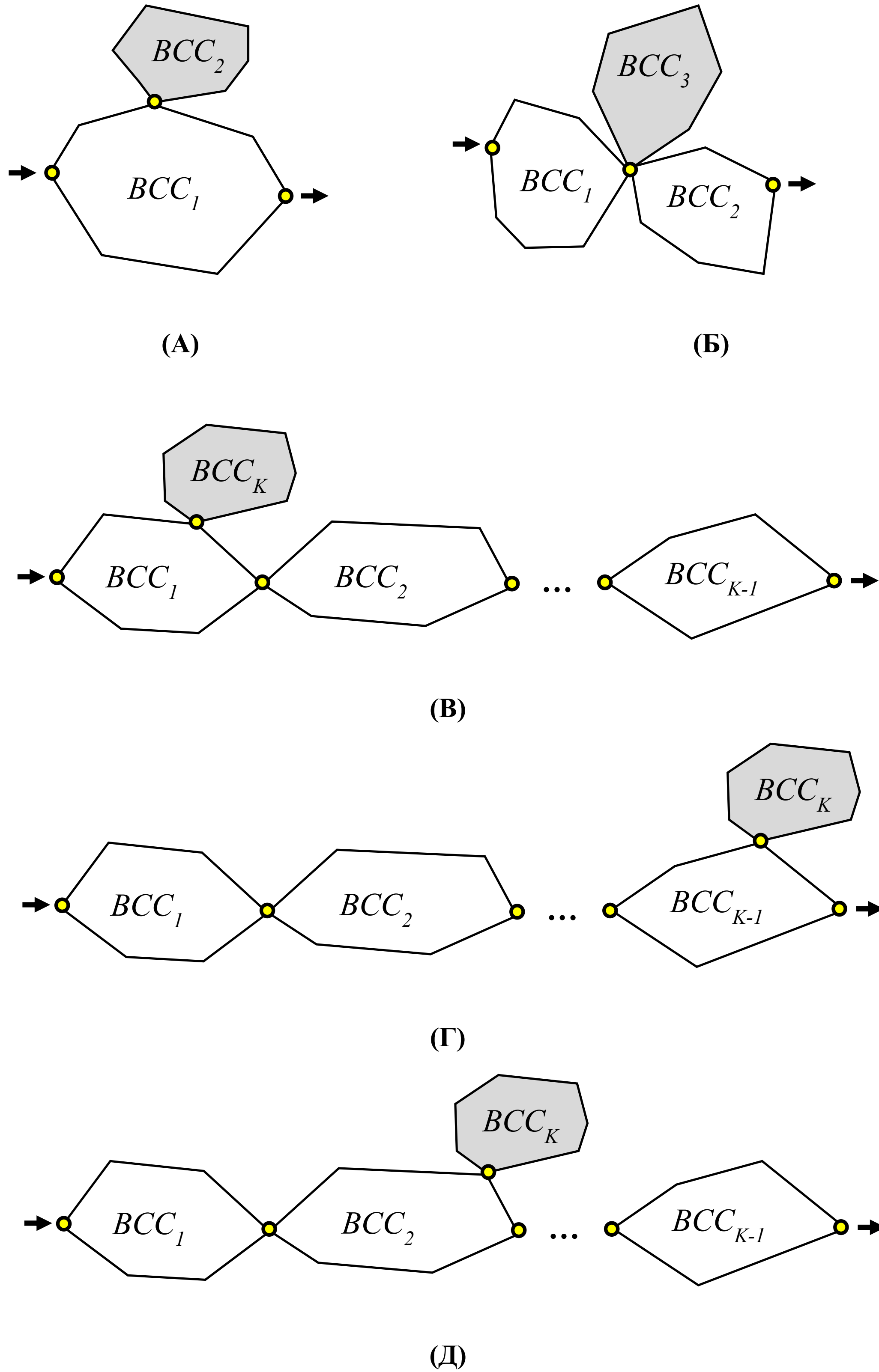


Рисунок 3.12. Усі можливі варіанти приєднання термінальної двозв'язної компоненти

Використовуючи означення точок зчеплення, можна провести ще одне наступне узагальнення (по аналогії з Лемою 3.5) для прорідження графу видимості вільних точок.

Лема 3.9. Якщо деяка двозв'язна BCC_n компонента графа видимості вільних точок містить дві точки зчеплення p_i та p_j, і в компоненті існує більше одного ребра, то ребро $\overline{p_i p_j}$, яке сполучає точки зчеплення, не може бути ребром Гамільтонового шляху без самоперетинів між вхідною та вихідною точкам та ребром простого многокутника, що будується (Рисунок 3.13)

Доведення. Припустимо протилежне, що ребро $\overline{p_i p_j}$, яке сполучає точки зчеплення, входить до складу простого многокутника. За умовою в двозв'язній компоненті існують інші ребра. Оскільки компонента двозв'язна, кількість ребер, що суміжні для точок зчеплення не менше 2. Але для них, крім цього, є суміжними ребра або з іншої компоненти, або з простого ланцюга, який ми будуємо. Тобто кількість суміжних ребер наших вершин-точок зчеплення не менше 3. Це суперечить тому, що ребро $\overline{p_i p_j}$ входить до складу простого многокутника, в якому для кожної вершини-точки повинно бути рівно 2 суміжних ребра ■.

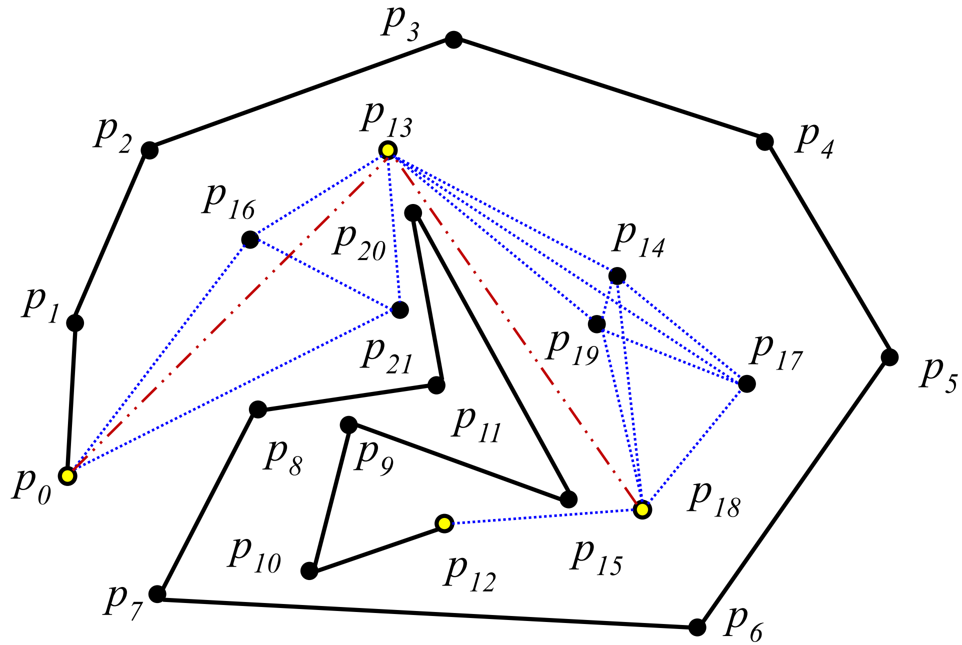


Рисунок 3.13. Приклад-ілюстрація для Означення 3.10 та Леми 3.9. Точки p0, p13, p18, p12 – точки зчеплення (p0 – вхідна, p12 – вихідна, p13, p18 – точки сполучення графу взаємної видимості вільних точок); штрих-пунктирні лінії показують ребра графу, які не можуть бути ребрами простого многокутника).

Припустимо тепер, що граф взаємної видимості вільних точок має $K>1$ двозв'язних компонент, які, у свою чергу мають тільки по дві точки зчеплення, і в якому немає термінальних компонент (див. Рисунок 3.14). Для такого випадку справедливою є наступна Лема.

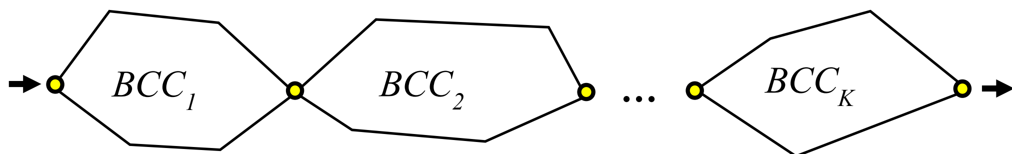


Рисунок 3.14. Граф взаємної видимості вільних точок без термінальних компонент.

Лема 3.10. Якщо граф взаємної видимості вільних точок складається з *K>1* двозв'язних компонент, кожна з яких має рівно дві різних точки зчеплення, що зв'язують її із іншими компонентами графу, або кінцями простого ланцюга (для вхідної та вихідної точки), і в цьому графі існує, принаймні, один Гамільтонів шлях від вхідної до вихідної вершини-точки, який завершує побудову простого многокутника (тобто, для кожної i-ої двозв'язної компоненти BCC_i між точками зчеплення даної компоненти існує H_i Гамільтонових шляхів, $H_i > 0$, принаймні, $H_i = 1$) , то для цього графу існує $\prod_{i=1}^{K} H_i$ Гамільтонових шляхів, що завершують побудову простого многокутника.

Доведення. Безпосередньо витікає з умов леми та незалежності шляхів між точками зчеплення всередині двозв'язних компонент та загально-відомих комбінаторних властивостей комбінації елементів незалежних множин ■.

Означення 3.11. *Невиродженою* двозв'язною компонентою графа взаємної видимості вільних точок будемо називати таку компоненту, яка містить не менше чотирьох точок-вершин. Відповідно, *виродженою* компонентою будемо називати таку двозв'язну компоненту графа взаємної видимості вільних точок, яка містить три або дві точки-вершини.

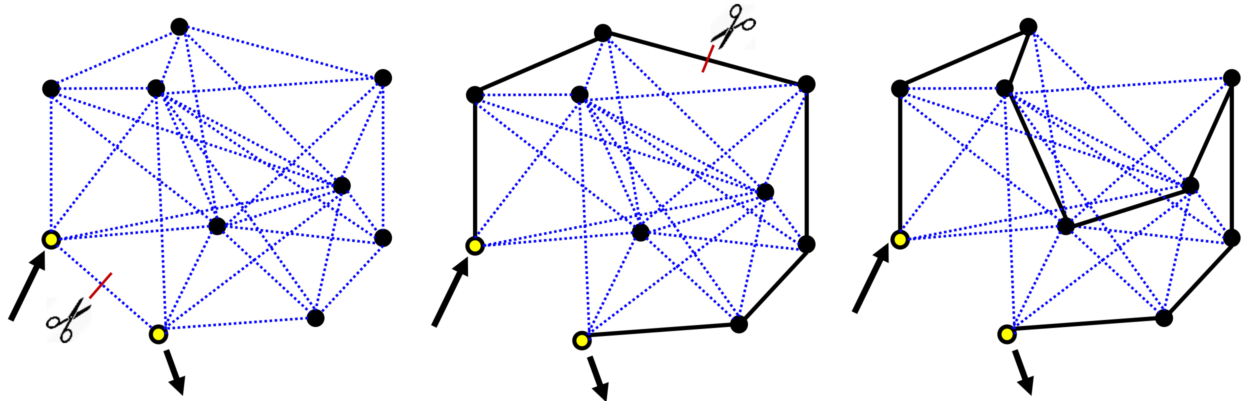
Примітка. Лема 3.10 є, по-перше, дуже корисною для скорочення великої кількості повторних перевірок необхідних умов Гамільтоновості графу взаємної видимості вільних точок, що задаються розглянутими вище Лемами. Коли ми повертаємося із першого успішного проходу по рекурсії, нам вже відомо, що існує Гамільтонів шлях без самоперетинів, який закінчує побудову простого многокутнику і перевірка для інших варіантів шляхів з точки входу, або з першої нетривіальної компоненти за точкою входу, є надлишковою. Тому, по-друге, ми можемо розбити задачу на само-подібні, але менш об'ємні задачі пошуку усіх можливих варіантів шляхів всередині двозв'язних компонентів.

Очевидно, що для компонентів з двома та трьома точками існує тільки один Гамільтонів шлях. У випадку двох точок – ця компонента є мостом; у

випадку трьох точок – згідно з Лемою 3.9 ребро-відрізок між двома з трьох точок не може бути ребром Гамільтонового шляху.

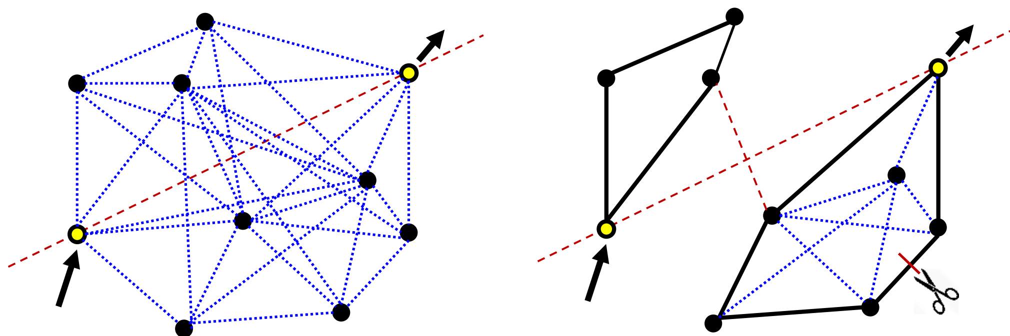
Лема 3.11. Якщо деяка підмножина ребер невиродженої двозв'язної компоненти графа взаємної видимості вільних точок співпадає з ребрами опуклої оболонки множини точок-вершин, а точки зчеплення – це вершини опуклої оболонки, то Гамільтонів шлях в такій компоненті завжди існує.

Доведення. Розглянемо спочатку варіант, коли точки зчеплення нетривіальної двозв'язної компоненти графа взаємної видимості вільних точок є сусідніми вершинами опуклою оболонки множини усіх точок-вершин цієї компоненти, тобто мають спільне ребро. За Лемою 3.9 таке ребро не може бути ребром Гамільтонового шляху, тому ми його видаляємо. Далі для решти точок-вершин виконуємо рекурсивну процедуру нарощування опуклими оболонками (Див. підрозділ 4.2), яка завжди приводить до успішного завершення побудови простого ланцюга (Рисунок 3.15, А).



Крок 1 *Крок 2* *Крок 3*

(A)



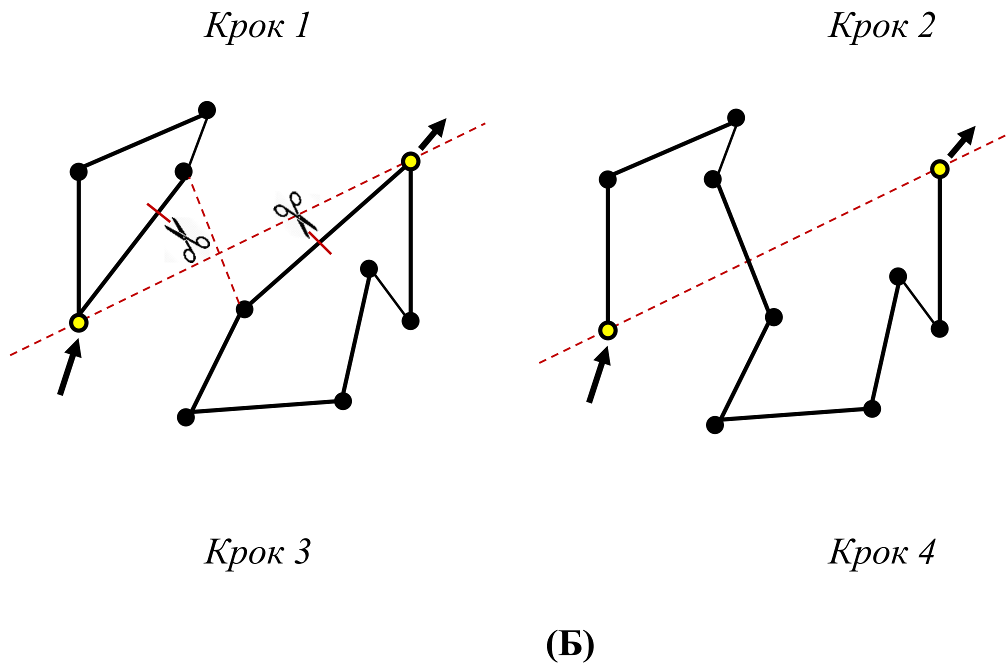


Рисунок 3.15. Ілюстрація для Доведення Леми 3.11. (А) - випадок, коли точки зчеплення і сусідніми вершинами на ОО компоненти): застосування методу вирізання ребер та нарощування простого ланцюга опуклими оболонками; (Б) – випадок коли точки зчеплення не є сусідніми вершинами.

Для варіанта, коли точки зчеплення не є сусідніми на опуклій оболонці, проводимо пряму між точками зчеплення. Ця пряма розділяє множину точок-вершин компоненти на дві під-множини, які є, у свою чергу, також опуклими.

Без втрати загальності, з точок зчеплення видаляємо по одному суміжному ребру опуклої оболонки, які лежать по різні сторони поділяючої прямої.

На отриманих підмножинах точок і ребер будуємо дві опуклі оболонки, кожна з яких включає точку одну точку зчеплення та під-множину точок, що лежить по іншу сторону від видаленого ребра, яке було суміжне. Якщо підмножина має тільки одну точку, то пропускаємо наступний крок.

Далі для кожної з отриманих опуклих оболонок, що охоплює свою підмножину вільних точок, також, як і в попередньому випадку, виконуємо рекурсивну процедуру видалення ребер та нарощування ланцюга простими

оболонками, починаючи з будь-якого ребра первинної опуклої оболонки нашої компоненти, до повного завершення замкненого простого ланцюга.

Після завершення процедури для кожної з підмножин видаляємо два протилежні ребра-відрізки, що є суміжними для точок зчеплення та не входять до складу первинної опуклої оболонки компоненти, у разі коли підмножина має більше однієї точки. Якщо підмножина має тільки одну точку, то беремо ребро опуклої оболонки, яке суміжно відповідній точці зчеплення. Внутрішні кінці обраних ребер (точки, що не є точками зчеплення), з'єднуємо ребром та отримуємо простий ланцюг без самоперетинів (Гамільтонів шлях), які з'єднує точки зчеплення (Рисунок 3.15, Б) ■.

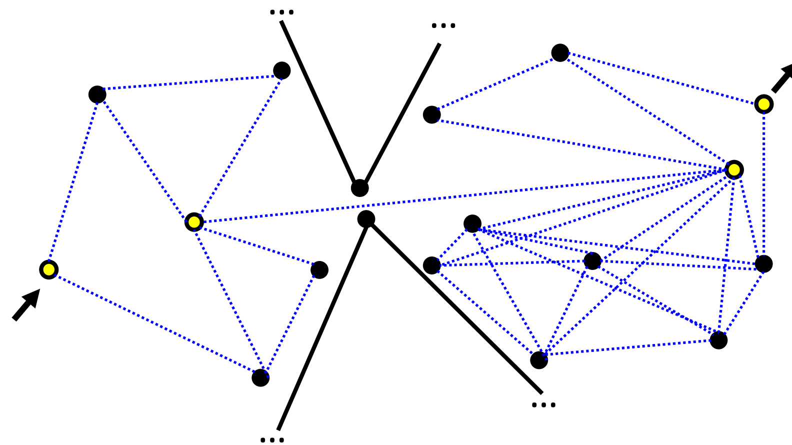


Рисунок 3.16. Контр-приклад, який демонструє випадок при якому точка зчеплення не є вершиною ОО своєї компоненти і Гамільтонів шлях неможливий.

Примітка. Зауважимо, що вимога Леми 3.11, щоб точки зчеплення були вершинами опуклої оболонки компоненти є суттєвою. Рисунок 3.16 показує відповідний контр-приклад, в якому компоненти з одного боку є опуклими, а з іншого їх точки зчеплення (сполучення у даному випадку), не є вершинами опуклих оболонок своїх компонент.

Наведемо, ще одне твердження, яке може бути використане для проріджения графа взаємної видимості вільних точок.

Лема 3.12. Жодне ребро графу взаємної видимості вільних точок, що перетинає міст цього графу, не може бути ребром Гамільтонового шляху без самоперетинів між вхідною та вихідною точкам та ребром простого многокутника, що будується.

Доведення. Міст містить єдине ребро, яке сполучає точки зчеплення, тому це ребро обов’язково повинно входити в будь-який Гамільтонів шлях графу взаємної видимості вільних точок, і, відповідно, простого многокутнику, який будується. Тому безпосередньо із визначення простого многокутника витікає, що жодне інше ребро не може перетинати міст ■.

Важливою необхідною умовою є також перевірка мінімальної двозв’язності графу взаємної видимості вільних точок, яка безпосередньо витікає із визначення Гамільтонового циклу без самоперетинів в геометричному графі (тобто простого многокутника).

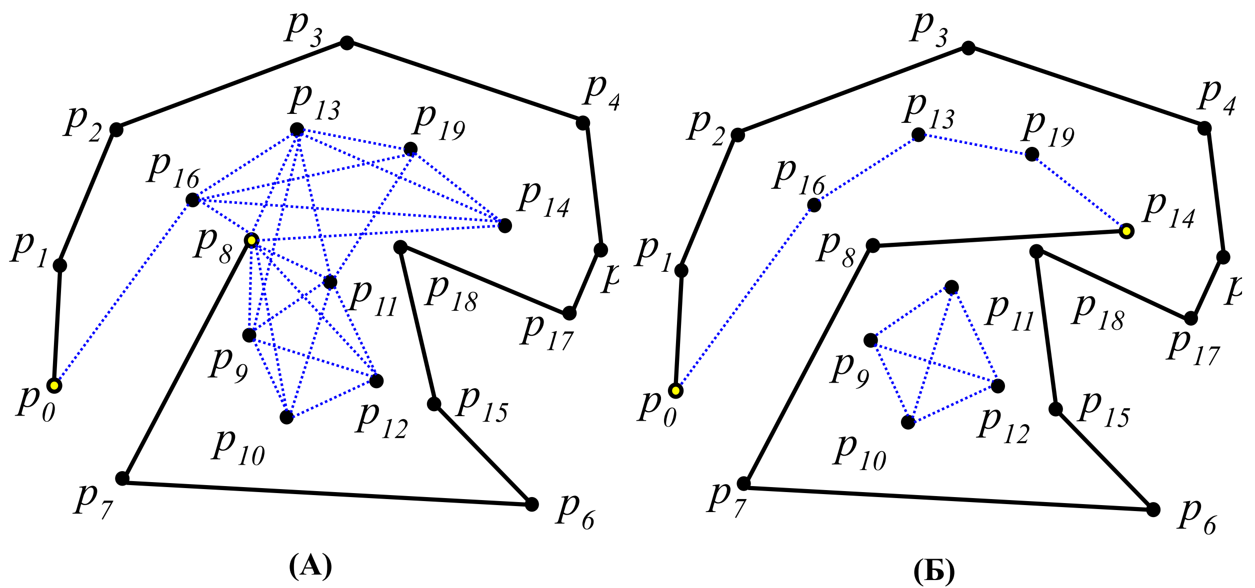


Рисунок 3.17. Приклади конфігурації, в якій граф взаємної видимості вільних точок при додаванні наступного ребра розпадається на дві компоненти.

Лема 3.13. Граф взаємної видимості вільних точок не повинен мати вершин зі ступенем менше ніж 2 та бути розщепленим на дві чи більше окремих компонент, інакше то не існує Гамільтонового шляху між вхідною та вихідною точками та, відповідно, побудова простого многокутника не може бути завершена (Рисунок 3.17).

Зауважимо, що перевірка графу на те, що він не має декількох незв’язних компоненти, може бути виконана за допомогою пошуку в ширину: BFS - breadth first search (Рисунок 3.18).

```
bool terminalComponentExists;
size_t idxq[MaxSize];
int ord[MaxSize];
size_t qhead;
size_t qtail;

inline void push(size_t idx){
	idxq[qtail] = idx;
	qtail++;
}
inline size_t pop(){
	size_t result = idxq[qhead];
	qhead++;
	if (qhead == qtail) qhead = qtail = 0;
	return result;
}
inline bool qIsNotEmpty(){
	return qhead != qtail;
}

bool isConnected(size_t numOfPoints, size_t chainSize, size_t pEntryPoint)
{
	for (size_t i = 0; i < numOfPoints; i++) ord[i] = -1;
	int cnt = 0;
	qtail = qhead = 0;
	push(pEntryPoint);
```

```
        ord[pEntryPoint] = cnt;
        while (qIsNotEmpty()){
                size_t idx = pop();
                for (size_t i = 0; i < degree[idx]; i++){
                        size_t child = vertices[idx][i];
                        if (ord[child] == -1){
                                push(child);
                                ord[child] = cnt++;
                        }
                }
        }
        if (numOfPoints - chainSize != cnt){
                return false;
        }
        return true;
}
```

Рисунок 3.18. Алгоритм перевірки зв'язності графу за допомогою пошуку в ширину.

Підсумуємо леми, що викладені у даному розділі у вигляді наступної теореми, справедливість якої безпосередньо витікає з Твердження 3.4, 3.5, 3.6 та того факту, що усі операції зі списками ребер не перевищують розміру самого списку $O(N^2)$.

Теорема 3.3. Загальна складність процедур proріджeння графу взаємної видимості вільних точок і перевірок необхідних та достатніх умов можливості завершення побудови простого многокутника, що викладені у Лемах 3.5-3.12 не перевищує $O(N^2)$.

Висновки до Розділу 3

В Розділі 3 були розглянуті точні розв'язки для Задач 1-3 методом повного (вичерпного) перебору усіх можливих значущих перестановок індексів точок та різними підходами до оптимізації цього методу.

Відсікання дерева варіантів в рекурсивній процедурі послідовного нарощування простого ланцюга, що представлена в Розділі 3, дозволяє отримувати точний розв'язок для майже в два рази більших множин точок, ніж було відомо раніше. Часова складність перевірки можливості існування Гамільтонового шляху в графі взаємної видимості вільних точок на кожному кроці рекурсивної процедури побудови простого ланцюга не перевищує $O(N^2)$, але при умові виділення $O(N^4)$ пам'яті.

У відомих на момент даної написання дисертаційної роботи джерелах (див. [8]) для точного розв'язку Задач 1-3 розглядається тестові набори даних розміром не більше 15 точок. Обчислювальні експерименти з використанням реалізацій алгоритмів, що описані в Розділі 3 показують, що для найбільш складних конфігурацій точок з опуклою оболонкою-трикутником та випадково розташованими точками всередині розмір вхідної множини може становити до 20 точок; для випадково згенерованих точок з рівномірним розподілом імовірності по обох координатах – до 30 точок в залежності від балансу точок на опуклій оболонці та всередині многокутника.

Зауважимо, що точне розв'язок NP-повних та NP-складних задач навіть для такого відносно невеликого розміру вхідних даних дає можливість більш якісно досліджувати поведінку наближених алгоритмів розв'язку таких задач, порівнюючи отримані розв'язки з точними.

Окрім цього, як було зазначено в Розділі 1, точне значення оптимальних конфігурацій відомо лише для 6 точок. Представлені методи точного розв'язку Задачі 1 дозволяють прискорити пошук оптимальних конфігурацій точок для більших N.

РОЗДІЛ 4. МЕТОДИ ПОРОДЖЕННЯ ОКРЕМИХ ТИПІВ МНОГОКУТНИКІВ

4.1 Побудова зіркових многокутників

Оскільки зіркові многокутники і методи розв'язку Задач 1-3 для цього типу геометричних об'єктів є найбільш глибоко вивченими (див. [8], [45]) надамо опис відповідних процедур лише оглядово.

Для повного розв'язку Задачі 1-3 з оцінкою $O(N^4)$ для пам'яті, що потрібна для діаграми еквівалентності зіркового розбиття, виконується наступна загальна процедура.

По-перше, для вхідної множини точок будується діаграма еквівалентності зіркових розбиттів, яка утворюється усіма променями, що направлені на зовнішню сторону від відрізків, які сполучають усі можливі пари точок, уздовж прямої, що утворена кожною парою.

Кожна область еквівалентності – це опуклий многокутник, будь-яка внутрішня точка якого може служити у якості центру зіркового розбиття. Тому, по-друге, для простоти обираємо центроїд будь-яких трьох точок з опуклої оболонки кожної області у якості центру зіркового розбиття.

Нарешті, відносно обраного центру зіркового розбиття сортуємо усі точки вхідної множини за значенням полярного кута променю, який проходить із центру розбиття до кожної з точок. Отримана послідовність точок – зірковий многокутник для обраної області еквівалентності.

Оскільки максимальна кількість областей еквівалентності складає $O(N^4)$, а сортування N точок за полярним кутом можна виконати за $O(N\log N)$, то загальна складність повного розв'язку Задач 1 та 2 складає $O(N^5\log N)$.

Відповідно, якщо виконане обчислювання діаграми еквівалентності зіркових розбиттів, то вибір індексу області еквівалентності виконується за O(1) часу, та побудова зіркового многокутника на цій області - за час $O(N\log N)$.

4.2 Побудова простих многокутників нарощуванням опуклих оболонок

Оскільки, як було зазначено вище, розмір множини можливих простих многокутників зростає експоненційно з ростом розміру вхідної множини точок, то для практичної реалізації Задач 1-3 використовуються різні евристичні підходи та методи, що були оглядово розглянуті у Розділі 1. У даному підрозділі представлено новий евристичний метод розв’язку Задач 1-3.

Спочатку дамо неформальний опис методу. На першому кроці знаходиться опукла оболонка *CH*(*S*). Далі множина точок, що складається з фрагменту поточного контуру та почергово, в залежності від кроку, зовнішніх чи внутрішніх вільних точок, охоплюється опуклою оболонкою з приєднанням її нового фрагменту до цього контуру. Тобто частина «вільних» точок, які опиняються на опуклій оболонці, приєднуються до многокутника, що будується. Для решти «вільних» точок процес повторюється до тих пір, доки усі вони не опиняться у складі многокутника. На Рисунку 4.1 показано приклад виконання процедури побудови простого многокутника нарощуванням опуклої оболонки.

Опишемо формально основні кроки алгоритму, опускаючи несуттєві деталі:

1. Побудувати опуклу оболонку *CH(S)* множини *S* (складність *O(NlogN)*) та прийняти її за поточний контур многокутника ***polygon***, що будується, а також за поточний фрагмент контуру ***contourFragmen*** для наступного кроку, окремо зберегти множину внутрішніх «вільних» точок ***freePooints*** (*S**CH*(*S*)) та контур ***polygon***;
2. Виконати рекурсивну процедуру ***doCutAndGrow*** для ***polygon***, ***contourFragmen*** та ***freePooints***:

2.1 Якщо всі точки множини ***S*** вже в поточному контурі многокутника ***polygon***, вийти з рекурсії з кінцевим результатом;

2.1.1 Цикл по всім можливим ребрам ***subFragment(i, i+1)*** поточного фрагменту контуру ***contourFragmen***;

2.1.2 Побудувати опуклу оболонку ***CH***(***S****i*) на точках ***subFragment****(i, i+1)* та ***freePooints***, запам'ятовуючи нові «вільні» точки опуклої оболонки ***localFreePoints****(i, i+1)*;

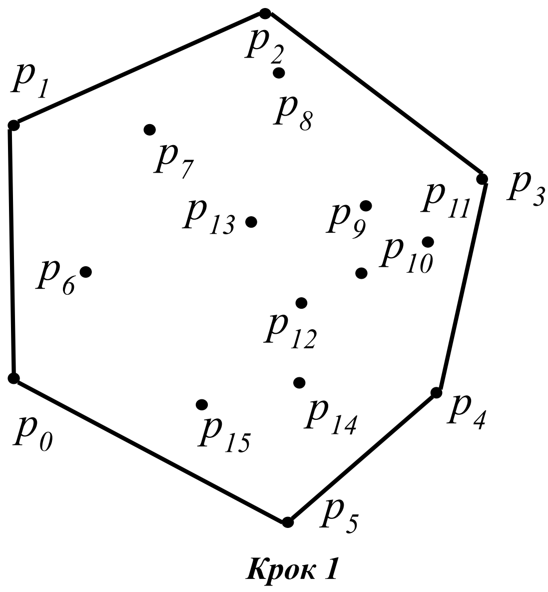
2.1.3 Приєднати фрагмент ***CH***(***S****i*)***subFragment****(i, i+1)* до поточного контуру ***polygon*** многокутника, що будується.

2.1.4 Перевірити ***цільовий предикат відбору*** для даного контуру та виконати рекурсивно процедуру ***doCutAndGrow*** для ***polygon***, ***subFragment****(i, i+1)*, ***localFreePoints****(i, i+1)*, інакше перейти до наступного кроку циклу.

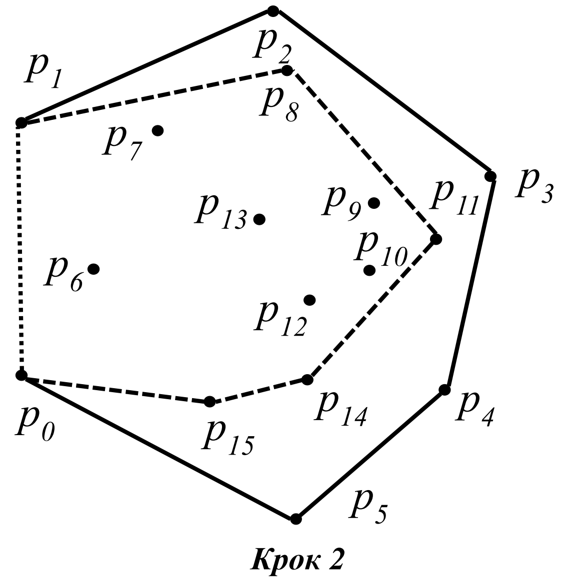
В залежності від рівня рекурсії «вільні» точки почергово опиняються зовні чи всередині многокутника, що будується. Алгоритм умовно «нарощує» контур опуклими оболонками, при цьому рекурсивна процедура в обох випадках залишається універсальною.

Теорема 4.1. Алгоритм побудови випадкового простого многокутника методом рекурсивного нарощування опуклих оболонок на видаленому ребрі простого ланцюга, що будується, має складність O(*NlogN*) і потребує O(*N*) пам'яті.

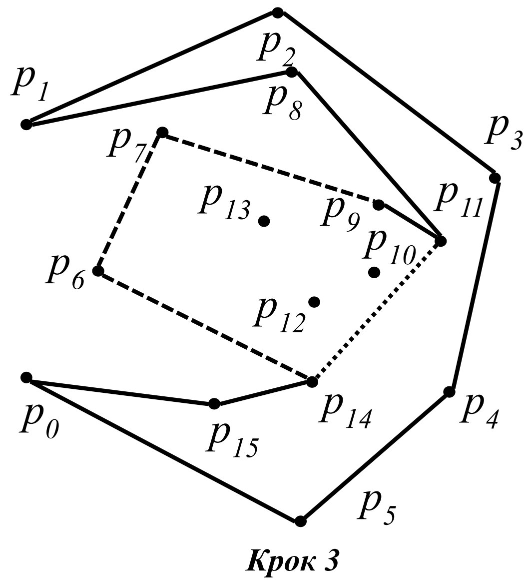
Доведення. Часова складність кроку 1 (побудова опуклої оболонки) складає *O(NlogN)*. При цьому, на кроці попередньої обробки множини точок список їх індексів розташовується у порядку зростання абсциси. Для цього списку потрібно *O(N)* пам'яті. Побудова опуклої оболонки на кожному наступному кроці рекурсії з використанням вже відсортованого списку точок займає *O(N)* часу. Глибина рекурсії не може перевищувати *N*. Отже, загальна часова складність алгоритму O(*NlogN*) та загальна потреба пам'яті становить *O(N)* ■.



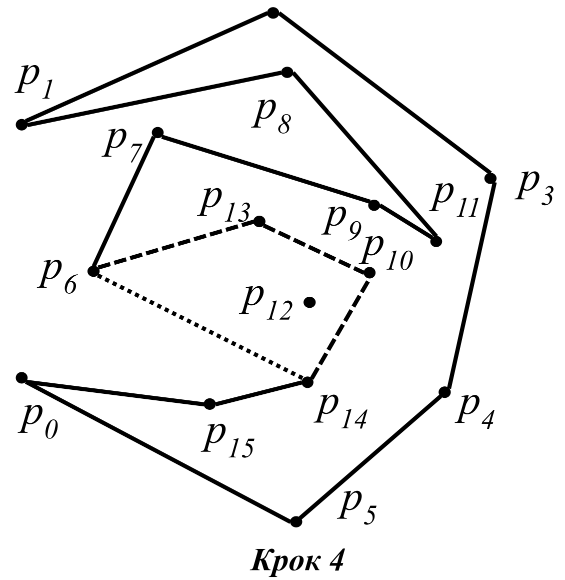
Крок 1



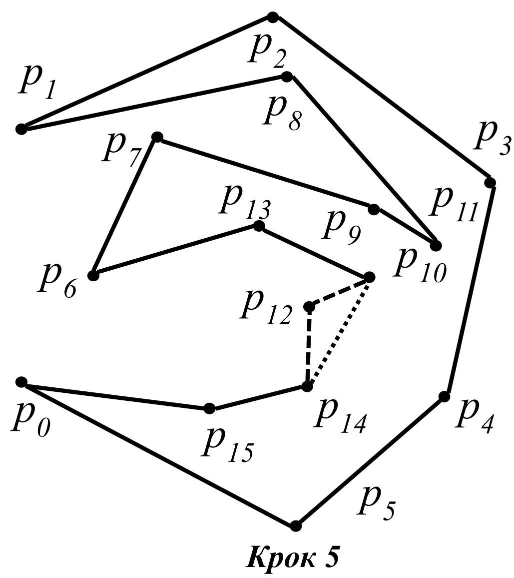
Крок 2



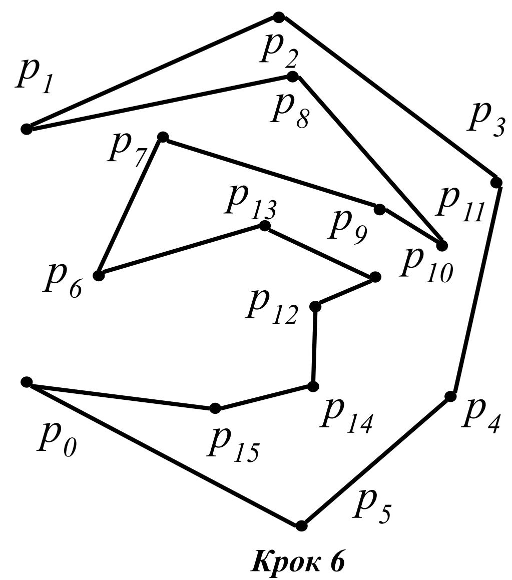
Крок 3



Крок 4



Крок 5



Крок 6

Рисунок 4.1. Побудова простого многокутника нарощуванням ОО.

Зауважимо, що оскільки описаний алгоритм оперує тільки множинними операціями та опуклими оболонками, він а) дозволяє зробити узагальнення на Евклідові простори довільної розмірності $\boldsymbol{E^d}$ з точністю до заміни фрагментів та суб-фрагментів суміжними гранями-ділянками поверхні многогранника, що будується, та їх об'єднаннями; б) представляє із себе схему конструктивного доведення існування простого многогранника для множини точок в $\boldsymbol{E^d}$.

Теорема 4.2. Для будь-якої заданої скінченої множини точок Евклідового простору розмірності d>2 існує простий поліедр (многогранник), який містить усі точки цієї множини у якості своїх вершин (тобто, в просторі $\boldsymbol{E^d}$ для будь якої множини S з $N>d$ точок у загальній позиції, в існує простий многогранник, який включає усі точки множини S як вершини.)

Доведення. У якості опорної обираємо будь яку з граней *CH(S)* для наступного кроку рекурсії. Наприклад, як у випадку зі спіральним многокутником, ми можемо обирати першу грань в структурі даних, що містить усі грані поточного фрагменту опуклої оболонки. Для кожного *(k+1)*-го кроку рекурсії, якщо все ще є вільні точки, то вони, за побудовою на кроці *k*, є внутрішніми точками опуклої оболонки. При побудові на них опуклої оболонки наступного кроку множини точок перетинаються з нею тільки на опорній грані, яка видаляється з поверхні многогранника, що будується. Тобто отримане об'єднання граней не має самоперетинів і гарантовано завершується додаванням частини поверхні оболонки, що побудована на останньому рівні рекурсії ■.

Хоча представлений підхід, як і більшість попередніх відомих евристичних алгоритмів, забезпечує генерацію тільки деякої підмножини многокутників з усіх можливих варіантів, на відміну від них, дозволяє природне узагальнення для просторів довільних вимірів $\boldsymbol{E^d}$, та може використовуватися, як основа конструктивного доведення існування простого d-вимірного многогранника.

Крім того, деякі конфігурації такі, як спіральні, є окремими випадками запропонованого алгоритму при певних значеннях його параметрів. Для отримання спірального многокутника на кожному кроці рекурсії необхідно

обирати для видалення та нарощування опуклої оболонки те з ребер, яке примикає до попередньо обраного ребра з визначеної сторони обходу.

Зауважимо, що наведений метод також може служити для побудови простих ланцюгів без самоперетинів.

4.3 Квітко-подібні многокутники і метод їх породження

Розглянемо Рисунок 4.2, на якому наведено приклад простого многокутника, що побудований на трикутниках утворених зірковим розбиттям з точкою *q* у якості центру розбиття. За q може бути обраний центроїд області еквівалентності зіркового розбиття. Такий тип многокутників будемо називати квітко-подібні (або спрощено «квіткові»).

Очевидною властивістю квітко-подібних многокутників є існування точки всередині многокутника, яку можна з’єднати з усіма вершинами опуклою оболонки *CH(S)* відрізками, що не перетинають жодне з ребер многокутника.

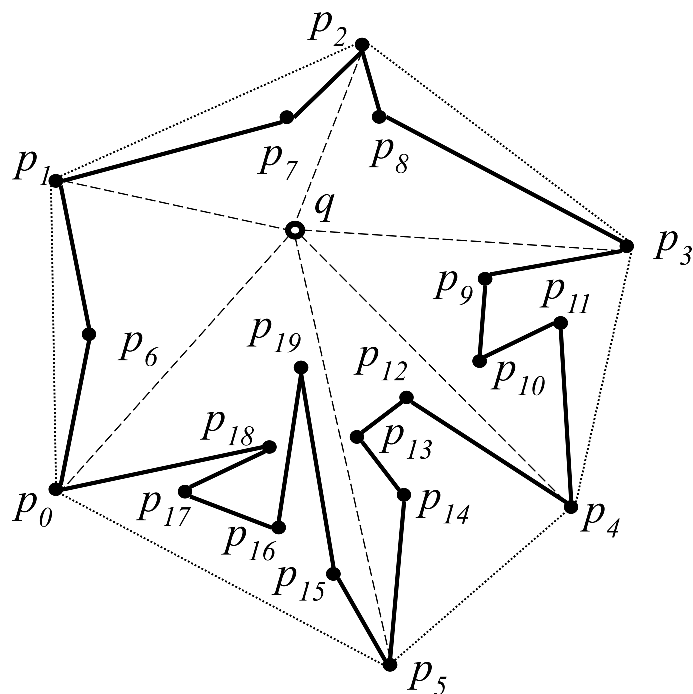


Рисунок 4.2. Приклад квітко-подібного многокутника.

На відміну від зіркових многокутників, квітко-подібний многокутник утворюється об'єднанням *довільних* простих ланцюгів, що: а) спираються на дві сусідні вершини опуклої оболонки, б) включають внутрішні точки, які належать трикутнику, що утворений парою вершин опуклої оболонки і центром зіркового розбиття. При цьому, діаграма еквівалентності будується не на всіх парах точок, а тільки на усіх можливих парах точок опуклої оболонки і внутрішніх точок (назвемо таке розбиття «діаграма еквівалентності зіркових розбиттів для квітко-подібних многокутників»).

Теорема 4.3. Кількість усіх простих квітко-подібних многокутників для даної області еквівалентності R_k дорівнює $\prod_{i=1}^{h} N_i$, h – кількість вершин опуклої оболонки, N_i – кількість можливих простих ланцюгів для трикутника T_i k-го зіркового розбиття.

Доведення. Безпосередньо витікає із незалежності побудови простих ланцюгів на окремих трикутниках зіркового розбиття ■.

Наслідок 4.1. Якщо для даної множини точок існує K областей еквівалентності зіркових розбиттів для квітко-подібних многокутників, то загальна кількість квітко-подібних многокутників дорівнює $\sum_{k=0}^{K-1} \prod_{i=1}^{h(k)-1} N_i$.

Метод породження кітко-побідних многокутників полягає у наступному.

1) Будуємо опуклу оболонку множини точок *CH(S)*.

2) Будуємо діаграму еквівалентності зіркових розбиттів на усіх можливих парах «вершина опуклої оболонки – внутрішня точка»

3) Обираємо область еквівалентності та її центроїд q та визначаємо належність внутрішніх точок до трикутних областей, що утворюються ребрами опуклої оболонки і центроїдом.

4) Для кожної групи точок в трикутних областях виконуємо породження простого ланцюга (наприклад, методом нарощування опуклих оболонок з попереднього розділу або іншим).

5) об'єднуємо окремі ділянки ланцюга, отримуючи простий многокутник.

4.4 Побудова простих многокутників нарощуванням трикутниками

Розглянемо ще один метод породження випадкових простих многокутників довільного типу.

В роботі [8] описано евристику Steady Growth, суть якої полягає в поступовому нарощуванні контуру, починаючи з трикутника, побудованого на випадково обраних трьох точках, який не містить інших точок вхідної множини. На кожному наступному кроці алгоритму Steady Growth відшукується випадкова точка, яка з одним із ребер поточного контуру утворює трикутник, що не містить інших точок. Ця точка обирається таким чином, щоб вона була зовнішньою до опуклої оболонки множини точок поточного контуру простого многокутника, що будується. У цьому випадку гарантується, що існує принаймні одне ребро контуру, кінці якого видимі з обраної точки.

Дана евристика Steady Growth, як показано в [8], хоча і гарантує завершення побудови випадкового простого многокутника, не забезпечує повного покриття усієї множини можливих простих многокутників. З іншого боку, якщо не використовувати зазначені вище обмеження для вибору наступної точки, то можливі ситуації, коли залишаються вільні точки, що не можуть бути з’єднанні з будь-яким ребром контуру (див. Рис. 4.3).

Однак, на практиці, при генерації множини випадкових простих многокутників із заданою кількістю елементів у більшості випадків достатньо, щоб ймовірність зустріти такий випадок була відносно малою. Процедура породження простого многокутника в такому разі видає нульовий результат і перезапускається знову.

У якості альтернативи, якщо алгоритм виконаний як рекурсивна процедура, може застосовуватися бектрекінг – повернення на попередній крок з вибором іншої наступної точки. Але у цьому випадку процедура стає недетермінованою.

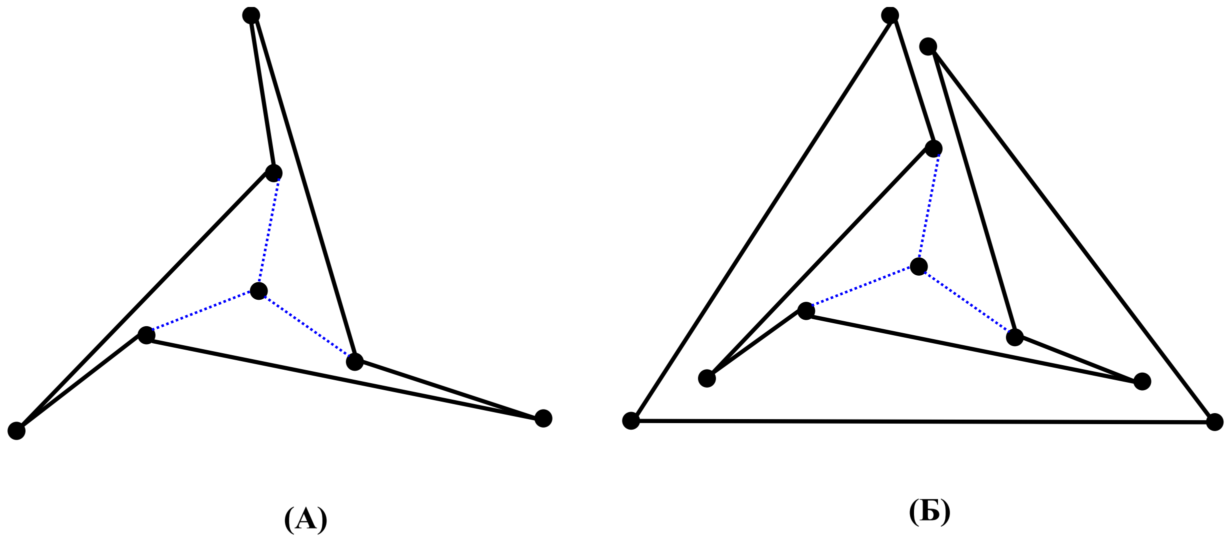


Рисунок 4.3. Приклади конфігурацій, в яких: (А) – внутрішня вільна точка, (Б) – зовнішня вільна точка недосяжні з обох кінців жодного ребра простого контуру.

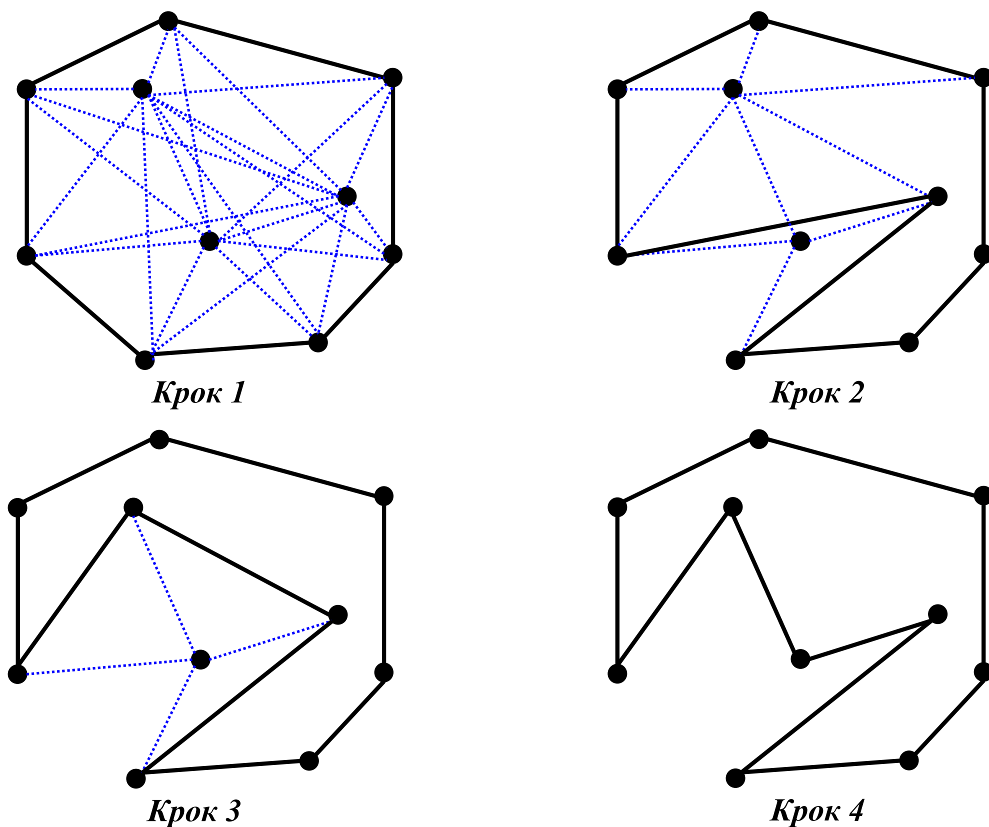


Рисунок 4.4. Приклад побудови простого многокутника нарощуванням трикутниками.

На Рисунку 4.4 наведено приклад кроків побудови випадкового простого многокутника методом нарощування контуру трикутниками. Суть метода

полягає у тому, що, починаючи з опуклої оболонки, будується контур простого многокутника шляхом визначення списку вільних точок як зовні, так і всередині замкнутого простого контуру, що будується, за умовою, що ці точки можуть утворювати трикутник на будь-якому ребрі контуру, який не перетинає цей контур. На кожному наступному кроці обирається випадкова точка з такого списку, або точка, вибір якої відповідає умовам відбіркового предикату, наприклад, мінімізує периметр поточного контуру.

Метод побудови простих многокутників нарощуванням трикутниками складається з наступних кроків:

1) Будуємо опуклу оболонку множини точок *CH(S)*.
2) До тих пір поки список вільних точок не порожній, або неможливо вільну точку приєднати до контуру:

   Виконуємо цикл:

   a. Створити порожній список пар «ребро-вільна точка»
   b. Перевірити усі пари «ребро-вільна точка» на перетин сторонами трикутника простого контуру, що будується
   c. Якщо перетину немає, то додаємо пару до списку пар
   d. Із списку пар «ребро-вільна точка» обираємо наступну точку для додавання у простий контур (з використанням відбіркового предиката)
3) Перевірити чи порожній список вільних точок, якщо так, то вийти з результатом «успіх», якщо список непорожній – з результатом «невдача»

Теорема 4.4. Якщо час виконання відбіркового предикату $O(P(N))$, де $P(N)$ – деяка поліноміальна функція від N, то розв’язку Задачі 3 (або відповідь про відсутність розв’язку), може бути отримане за $O(P(N)N^3)$ часу з використанням $O(N^2)$ пам’яті.

Доведення. По-перше, зауважимо, що список пар «ребро-вільна точка» має максимальний розмір $O(N^2)$. Внутрішній цикл (пункт b) у найгіршому випадку

потребує $O(N^2)$ часу. Основний зовнішній цикл виконується максимум $N\text{-}h$ разів, де h – кількість точок на опуклій оболонці, мінімальне значення якої $h=3$. Таким чином, загальний час виконання процедури побудови випадкового простого многокутника із заданими відбірковим предикатом властивостями $O(P(N)N^3)$ ■.

Зауважимо, що на практиці час виконання процедури, що наведена вище можна покращити, якщо, по-перше зробити попереднє обчислення перетинів усіх пар відрізків, а також, для кожного відрізка зберігати список відрізків, які його перетинають всередині.

Варто відзначити, що метод породження простих многокутників, є узагальненням підходу, який запропонований в роботі [] для наближеного розв’язку NP-повної задачі пошуку простого многокутника мінімальної площі.

Висновки до Розділу 4

В даному розділі запропонований новий клас простих многокутників квітко-подібні, що є над-класом зіркових многокутників. На відміну від зіркових многокутників, на заданій скінченій множині точок, кількість можливих квітко-подібних многокутників у найгіршому випадку зростає експоненційно.

Розділом 4 запропоновано два нових евристичних метода породження простих многокутників: метод нарощування опуклих оболонок і методо нарощування трикутниками.

Перевагами метода нарощування опуклими оболонками є швидкодія та гарантоване завершення для будь-якої вхідної множини точок у загальному розташуванні. У той самий час, недоліком даного методу є обмежене покриття множини можливих простих многокутників на заданій множині точок. Для практичних застосувань квітко-подібні многокутники, що будуються з

використанням діаграми еквівалентності зіркових розбиттів частково вирішують проблему обмеженого покриття при вирішенні Задачі 3.

Метод породження простих многокутників нарощуванням простого контуру трикутниками, починаючи з опуклої оболонки, вирішує проблему повного покриття – він дозволяє породжувати *усі можливі прості многокутники* досить ефективно. Основним недоліком методу є, по-перше, його недетермінованість, а по-друге, те, що він не забезпечує рівномірну імовірність породження окремих простих многокутників, оскільки один і той самий простий многокутник може бути побудований декількома варіантами послідовності нарощування простого контуру. У цьому легко можна впевнитися, якщо розглянути обернену процедуру – послідовне вивільнення вершин простого многокутника шляхом з’єднання вершин, що розташовані через одну. Але на практиці обидва недоліки майже несуттєві.

Підсумовуючи, в залежності від критичності вимог швидкодії чи якнайбільшого покриття множини можливих простих многокутників, може обиратися один із запропонованих в даному розділі методів породження даного типу геометричних об’єктів. Для просторів довільної розмірності метод нарощування простих оболонок, що запропонований в Розділі 4, є єдиним відомим автору методом породження випадкових поліедрів на заданій множині точок при умові, що усі точки входять у поліедр у якості його вершин.

РОЗДІЛ 5. ПРАКТИЧНА РЕАЛІЗАЦІЯ

5.1 Загальна архітектура програмної реалізації генератора геометричних об'єктів

Загальна архітектура програмної реалізації генератора геометричних об'єктів представлена на наступному Рисунку 5.1.

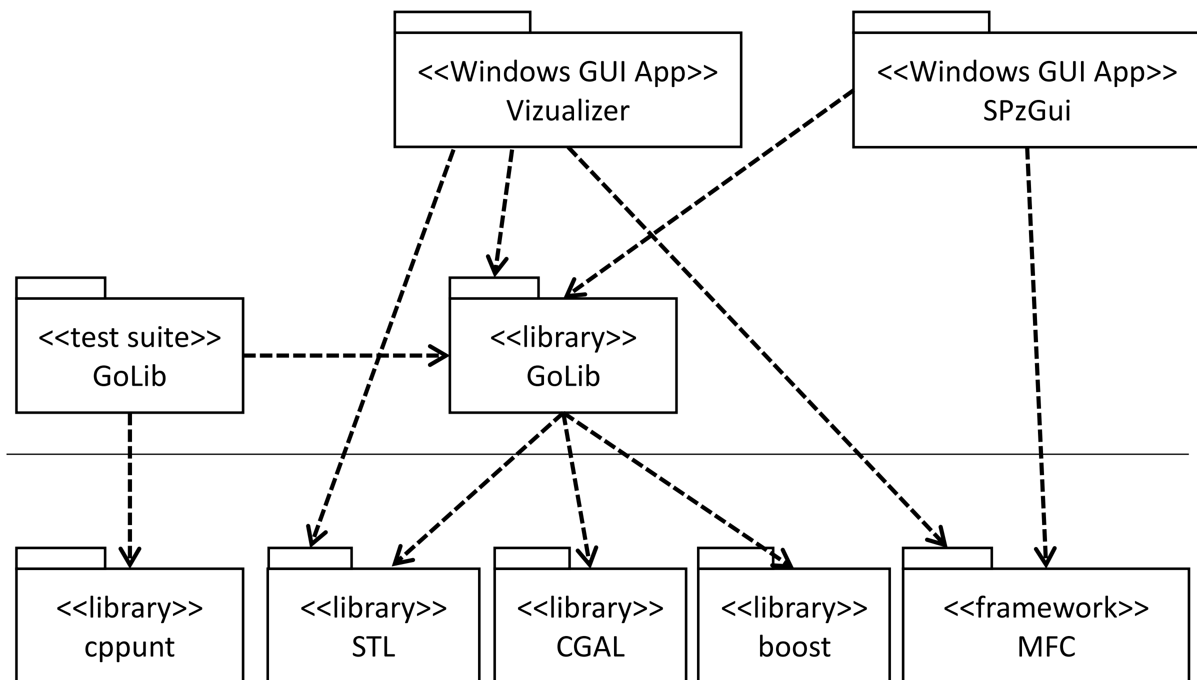


Рисунок 5.1. Загальна архітектура програмної реалізації генератора геометричних об'єктів.

Генератор геометричних об'єктів реалізовано як мультиалгоритмічну платформу на мові С++ у середовищі Microsoft Visual Studio Community 2015.

Платформа складається із наступних функціональних пакетів.

1) Бібліотека GoLib реалізації алгоритмів породження (генерації) простих многокутників різними методами, що викладені в дисертаційній роботі, а також частини методів, що описані у різних джерелах, та з якими виконувалося порівняння.
2) Пакет модульних тестів для перевірки коректності роботи алгоритмів та базових елементів реалізацій таких як обчислення площини, побудова опуклої оболонки та подібних.

3) Програмне забезпечення з графічним інтерфейсом користувача SPzGUI, яке дозволяє в інтерактивному режимі вводити тестові набори точок, виконувати навігацію по отриманим множинам геометричних об'єктів.
4) Програмне забезпечення з графічним інтерфейсом користувача Visualizer, основна задача якого – навігація по графам видимості вільних точок на різних кроках виконання алгоритмів.

Для реалізації перелічених пакетів було використано наступні готові бібліотеки та каркасне програмне забезпечення:

- стандартна бібліотека шаблонів STL;
- Бібліотека шаблонів boost (використовується бібліотекою CGAL);
- Реалізація побудови розбиття площини на області еквівалентності з бібліотеки CGAL;
- Каркас для побудови графічних Windows-застосунків MFC;
- Бібліотека для побудови модульних тестів cppunit.

Зауважимо, що при реалізації алгоритмів повного перебору з Розділу 3 для прискорення роботи алгоритмів використовувалися структури даних фіксованого розміру з константним часом виконання усіх основних операцій. Оскільки для більшості випадків порядок елементів в списках не має значення для наступних кроків розглянутих алгоритмів, то видалення елементу було реалізовано, як копіювання останнього елементу списку в комірку елементу, що видаляється, та зменшення кількості елементів на одиницю. Відповідно, додавання елементу реалізовано, як додавання елементу у кінець списку та збільшення лічильника кількості елементів на одиницю.

5.2 Формат вхідних даних

Формат вхідного файлу генератора геометричних об'єктів наступний:

N – кількість точок вхідної множини

X_0 Y_0 X_1 Y_1 … X_{N-1} Y_{N-1} – координати точок (розділені пропусками)

Приклад:

30

326 567 422 277 642 141 919 119 1143 156 1492 287 1519 657 1429 966 1036 1030 630 1023 399 870 1351 469 1229 331 1257 629 1278 783 1131 524 1053 647 1071 779 1148 928 969 320 889 738 836 418 845 587 754 246 867 916 641 359 652 541 693 833 475 494 535 709

5.3 Формат вихідних даних.

Формат вихідного файлу з множиною простих многокутників, що породжений на заданій множині точок має наступну структуру:

N – кількість точок вхідної множини

X_0 Y_0 X_1 Y_1 … X_{N-1} Y_{N-1} – координати точок (розділені пропусками)

SP – кількість простих многокутників

I_0 I_1 … I_{N-1} – індекси точок вхідної множини, що є вершинами простого многокутника (розділені пропусками, кожен набір індексів вершин простого многокутника починається з нового рядка).

Приклад (див. Рисунок 5.2):

11
479 508 492 277 683 127 987 114 1249 335 1266 626 1064 765 786 780 588 687 975 530 766 361
59
0 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 9 8 10
0 1 2 3 4 5 6 7 9 10 8
0 1 2 3 4 5 6 9 7 8 10
0 1 2 3 4 5 6 9 7 10 8
0 1 2 3 4 5 6 9 10 7 8
0 1 2 3 4 5 9 6 7 8 10
0 1 2 3 4 5 9 6 7 10 8
0 1 2 3 4 5 9 6 10 7 8
0 1 2 3 4 5 9 10 6 7 8
0 1 2 3 4 9 5 6 7 8 10
0 1 2 3 4 9 5 6 7 10 8
0 1 2 3 4 9 5 6 10 7 8
0 1 2 3 9 4 5 6 7 8 10
0 1 2 3 9 4 5 6 7 10 8

0 1 2 3 9 4 5 6 10 7 8
0 1 2 9 3 4 5 6 7 8 10
0 1 2 9 3 4 5 6 7 10 8
0 1 2 9 3 4 5 6 10 7 8
0 1 9 10 2 3 4 5 6 7 8
0 8 7 6 5 4 3 2 9 10 1
0 8 7 6 5 4 3 9 2 1 10
0 8 7 6 5 4 3 9 2 10 1
0 8 7 6 5 4 3 9 10 2 1
0 8 7 6 5 4 9 3 2 1 10
0 8 7 6 5 4 9 3 2 10 1
0 8 7 6 5 4 9 3 10 2 1
0 8 7 6 5 4 9 10 3 2 1
0 8 7 6 5 9 4 3 2 1 10
0 8 7 6 5 9 4 3 2 10 1
0 8 7 6 5 9 4 3 10 2 1
0 8 7 6 5 9 4 10 3 2 1
0 8 7 6 5 9 10 4 3 2 1
0 8 7 6 9 5 4 3 2 1 10
0 8 7 6 9 5 4 3 2 10 1
0 8 7 6 9 5 4 3 10 2 1
0 8 7 6 9 5 4 10 3 2 1
0 8 7 6 9 5 10 4 3 2 1
0 8 7 6 9 10 5 4 3 2 1
0 8 7 9 6 5 4 3 2 1 10
0 8 7 9 6 5 4 3 2 10 1
0 8 7 9 6 5 4 3 10 2 1
0 8 7 9 6 5 4 10 3 2 1
0 8 7 9 6 5 10 4 3 2 1
0 8 9 7 6 5 4 3 2 1 10
0 8 9 7 6 5 4 3 2 10 1
0 8 9 7 6 5 4 3 10 2 1
0 8 9 7 6 5 4 10 3 2 1
0 8 9 7 6 5 10 4 3 2 1
0 9 8 7 6 5 4 3 2 1 10
0 9 8 7 6 5 4 3 2 10 1
0 9 8 7 6 5 4 3 10 2 1
0 9 8 7 6 5 4 10 3 2 1
0 9 8 7 6 5 10 4 3 2 1
0 9 1 2 3 4 10 5 6 7 8
0 9 1 2 3 10 4 5 6 7 8
0 9 1 2 10 3 4 5 6 7 8
0 9 1 10 2 3 4 5 6 7 8
0 9 10 1 2 3 4 5 6 7 8

5.3 Приклади роботи генератора геометричних об’єктів та засобів візуалізації

Рисунок 5.2. Простий многокутник N=11 із прикладу вихідних даних.

Приклад:

На Рисунку 5.3 показано знімки екранів застосунку SPzGui з випадковим многокутником, що породжений на множині точок з прикладу вище (у попередньому підрозділі), а також роботу візуалізатора графів видимості вільних точок – Рисунок 5.4.

Рисунок 5.5 демонструє роботу візуалізатора діаграми еквівалентності зіркових розбиттів та алгоритму пошуку зіркового многокутника мінімальної площі.

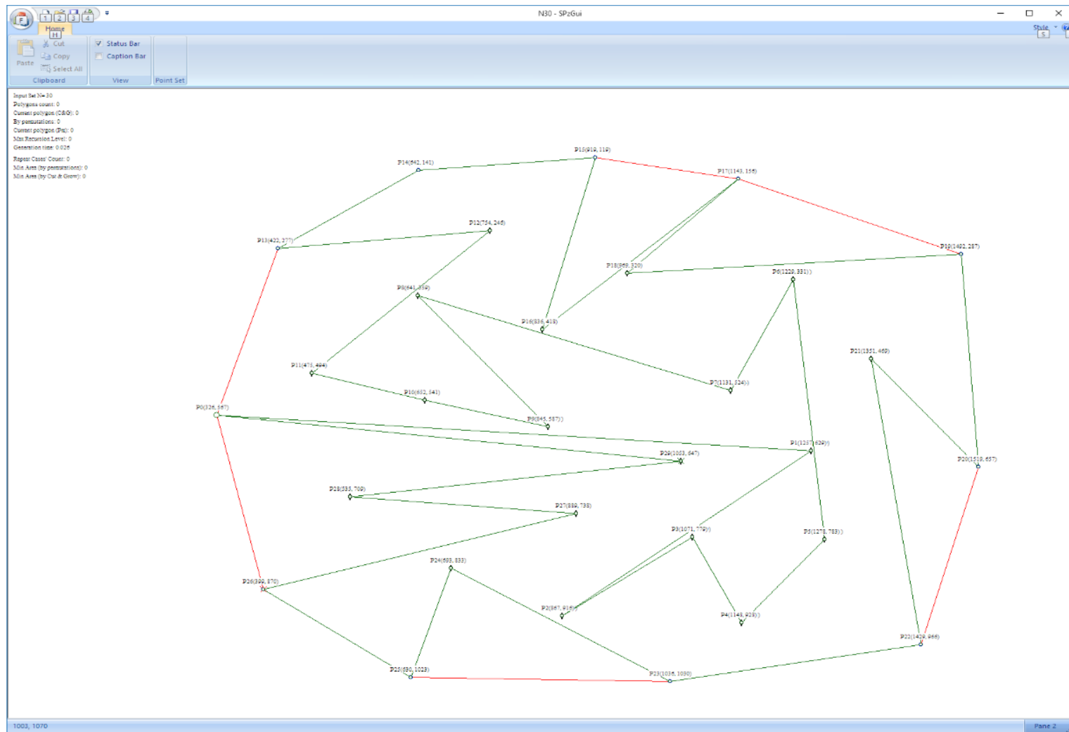


Рисунок 5.3. Приклад роботи програмної реалізації генератора геометричних об'єктів. Червоним кольором показані ребра ОО.

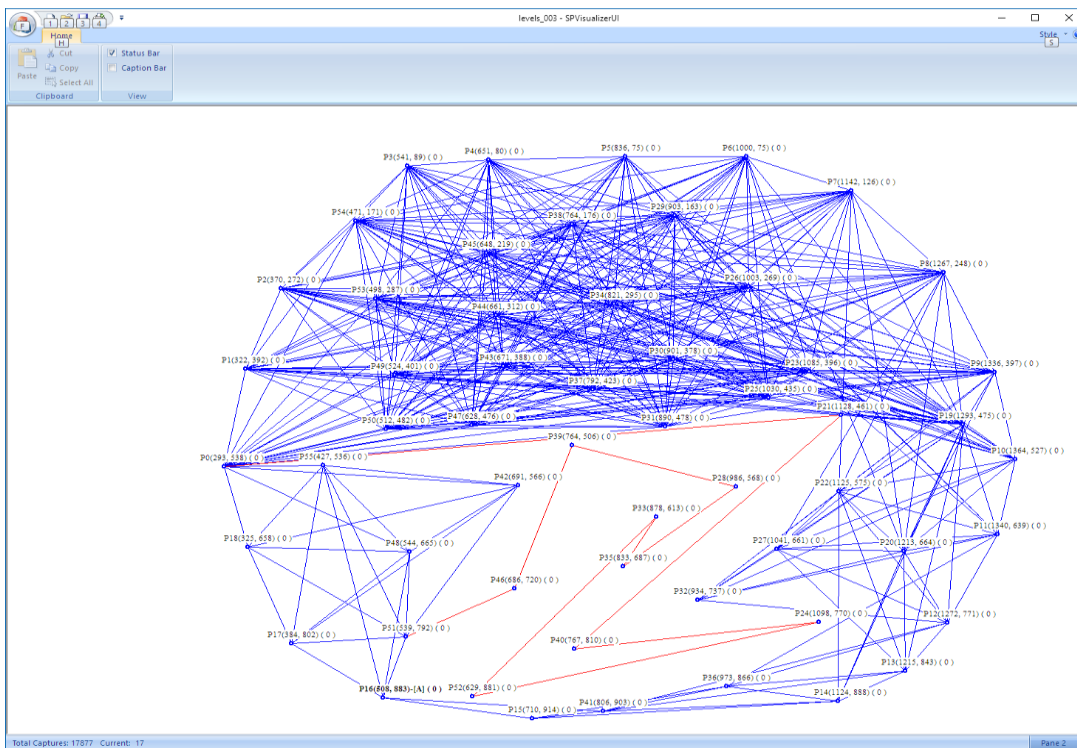


Рисунок 5.4. Приклад роботи візуалізатора графів видимості вільних точок. Простий ланцюг – червоним; граф видимості – синім кольором.

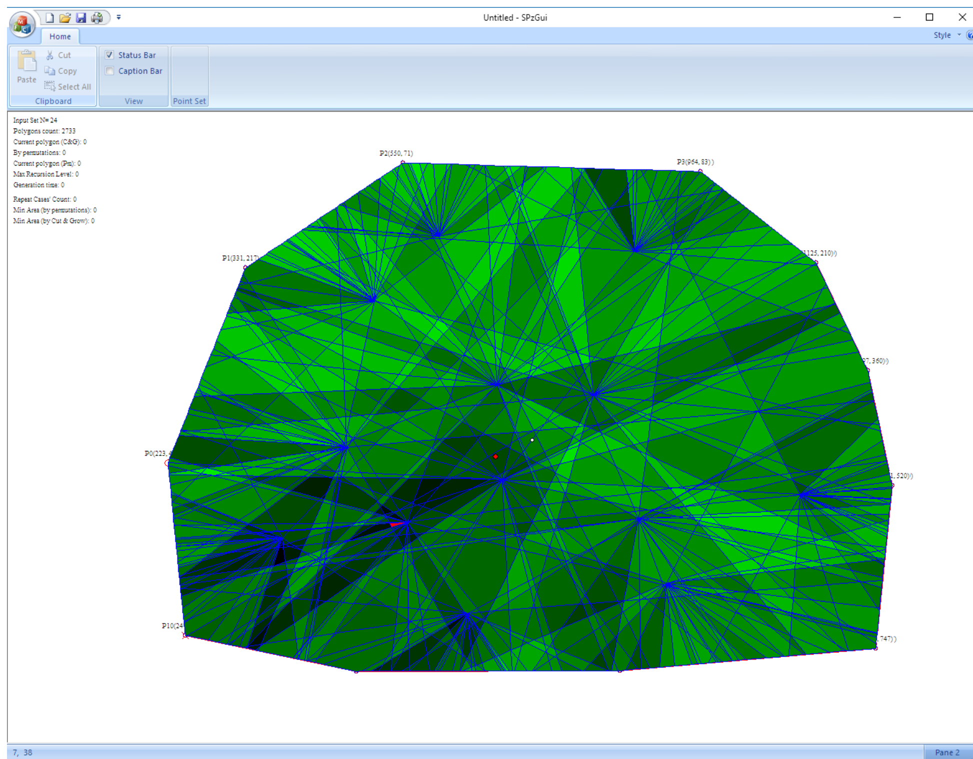


Рисунок 5.5. Приклад роботи візуалізатора діаграми еквівалентності зіркових розбиттів. Градація кольору відповідає площі зіркового многокутника, що утворюється відповідною областю еквівалентності: темніший колір означає меншу площу. Червоний многокутник відповідає області еквівалентності, що породжує зірковий многокутник мінімальної площі. Області еквівалентності обмежені синіми відрізками.

5.4 Застосування точного розв'язку Задачі 1 для отримання точного значення оптимальної конфігурації при N=7

В Розділі 1 (див. Таблиця 1.3) наведено відомі на сьогодні оцінки точного значення максимально можливої кількості простих многокутників, які можуть бути утворені при заданій кількості вхідних точок. Як видно з таблиці точне значення відомо тільки для $N=6$.

З використанням діаграми еквівалентності зіркового розбиття, точного рішення Задачі 3 і даних для усіх можливих типів розташування для $N=6$ (див. Розділ 2) можна показати, що оцінка $max|SP|=92$, максимального можливої кількості простих многокутників для $N=7$ є точною. Доведення виконуються вичерпним обчисленням усіх можливих типів розташувань, які можна отримати додаванням 7-мої точки до множини з 6 точок (див. Рисунок 5.6).

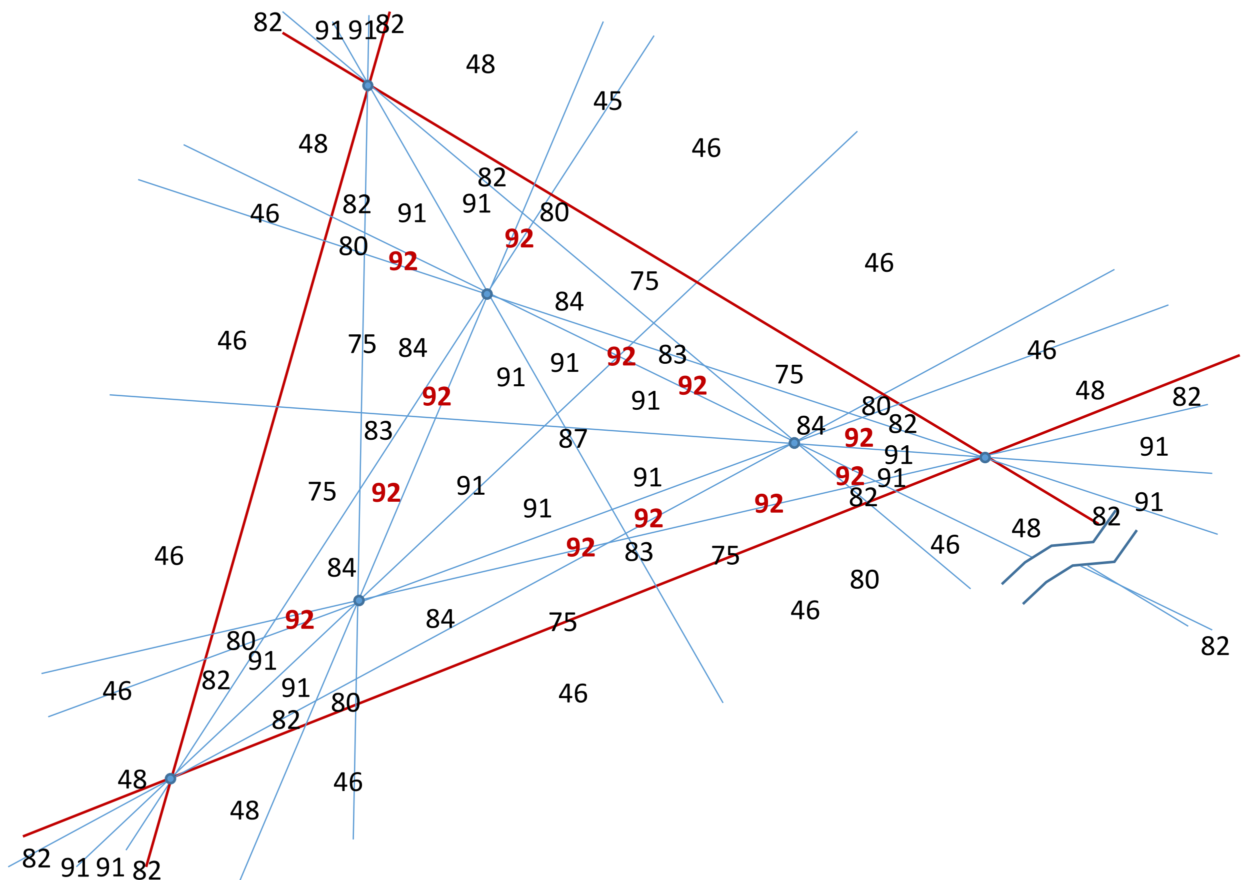


Рисунок 5.6. Застосування діаграми еквівалентності та точного розв’язку Задачі 3 для отримання точного рішення задачі про оптимальне розташування для N=7 (див. також Додаток Д-А.4).

5.5 Застосування точного розв’язку Задачі 1 для оцінювання жадібного алгоритму знаходження простого многокутника мінімальної площі

В роботі [49] запропоновано наближений (жадібний) алгоритм пошуку простого многокутника найменшої площі. У якості прикладу практичного застосування точного рішення для Задачі 1, в даній дисертаційній роботі виконано оцінку ефективності вказаного жадібного алгоритму.

Для заданої множини точок відшукуються: а) точний розв’язок задачі про простий многокутник мінімальної площі з використанням підходу, що описаний в Підрозділі 3.5, при цьому відбірковий предикат, який відповідає на питання чи є площа отриманого простого многокутника мінімальною, перевіряється при кожному завершенні побудови простого многокутника; б) наближений розв’язок задачі – з використанням процедури, що наведена в [49]. Відповідні результати, точний та наближений, порівнюються.

З Таблиць Д-А.1 та Д-А.2 у Додатку А можна бачити, що із збільшенням кількості точок у вхідній множині існує очікувана тенденція зменшення ефективності наближеного розв’язку, що отриманий з використанням жадібного алгоритму.

Примітки:

1) оригінальна процедура була реалізована авторами роботи [49] на Java. В даній роботі з метою уніфікації та можливості інтеграції в мульти-алгоритмічне середовище, жадібний алгоритм [49] був реалізований на мові C++ (див. відповідний код процедури у Додатку А, Рисунок Д-А.3);

2) автори [49] не навели доведення того факту, що простий многокутник можна построїти для будь-якої вхідної множини точок – питання про існування рішення для даного алгоритму залишається відкритим.

Для розглянутих розмірів вхідної множини точок (N = 16…22) середнє співвідношення наближеного і точного розв’язків задачі про простий многокутник мінімальної площі коливається в діапазоні 1,2…1,6.

Висновки до Розділу 5

Розділом 5 продемонстровано основні режими роботи практичної реалізації генератора геометричних об’єктів.

Реалізація генератора геометричних об’єктів у вигляді мульти-алгоритмічної платформи дозволяє уніфікувати формати вхідних та вихідних даних, а також взаємодію з застосунками для породження геометричних об’єктів і візуалізації результатів та проміжних (допоміжних) структур даних.

Представлення вихідних даних у простому і компактному текстовому форматі дозволяє їх використання у якості вхідних даних для тестування довільних алгоритмів, що обробляють прості многокутники.

На прикладі оцінки ефективності жадібного алгоритму знаходження простого многокутника мінімальної площини показано практичну цінність отриманого в даній роботі точного розв’язку Задачі 1 для певного діапазону розмірів вхідної множини точок.

ВИСНОВКИ

Головним результатом дисертаційної роботи є побудова мультиалгоритмічної платформи-генератора геометричних об’єктів на площині, що розв’язує задачу породження простих многокутників певних типів із визначеними властивостями на заданих скінчених множинах точок.

Основні результати дисертації.

1. Вперше застосовано поняття діаграми еквівалентності зіркових розбиттів, та областей еквівалентності для аналізу вхідних множин точок на площині та процедур породження декількох типів простих многокутників.
2. Досліджено та суттєво розширено перелік необхідних і достатніх умов існування Гамільтонового шляху в таких геометричних графах і відповідно, перевірки можливості завершити ланцюг побудовою простого многокутника. Умови існування Гамільтонового шляху базуються на аналізі графу взаємної видимості вільних точок. Загальна складність усіх перевірок умов обмежена $O(N^2)$ при умові використання $O(N^2)$ пам’яті. Отримано прискорення алгоритму шляхом скорочення об’єму обчислювань з плаваючою точкою на перевірки перетинів відрізків ціною $O(N^4)$ пам’яті, в якій зберігається попередньо оброблена інформація про перетині усіх пар відрізків.
3. Розроблено метод послідовного нарощування простого ланцюга з відсіканням дерева варіантів для реалістичних розмірів вхідної множини, який, фактично, наблизив складність обчислювань до $O(P(N)a^N)$, де $P(N)$ – поліноміальна функція від N; a – деяка константа.
4. Розроблено метод підрахунку усіх можливих простих многокутників на заданій скінченій множині точок, який суттєво прискорює процедури шляхом рекурсивного розбиття на під-задачі пошуку кількості

Гамільтонових шляхів на двозв'язних компонентах та перемноження окремих результатів.

5. Розроблено евристичний метод породження простих многокутників нарощуванням простих оболонок на видаленому ребрі простого ланцюга, що будується. Доведено, що метод є одним із самих ефективних на сьогодні, маючи часову складність *O(NlogN),* та потребуючи *O(N)* пам'яті. Показано, що деякі з відомих типів простих многокутників, зокрема спіральні, можуть бути отримані, як окремий випадок запропонованого методу.
6. Узагальнено метод нарощування простих оболонок для евклідових просторів довільної розмірності.
7. Доведено існування простого поліедру для скінченої множини точок у, так званому, загальному розташуванні, коли жодна з *d+1* точок не знаходяться в одній гіперплощині.
8. Вперше досліджено властивості нового типу простих многокутників – квітко-подібних (квіткових).
9. Розроблено відповідну процедуру породження квітко-подібних многокутників з використанням діаграми еквівалентності зіркових розбиттів.
10. Узагальнено на простори більших розмірностей жадібний метод пошуку простого многокутника мінімальної площини, який включає усі точки вхідної множини у якості вершин: жадібний метод пошуку простого поліедра мінімального об'єму.
11. Виконано практичну реалізацію генератора геометричних об'єктів у вигляді мульти-алгоритмічного середовища та засобів візуалізації множин простих многокутників, діаграм еквівалентності зіркових розбиттів та графів взаємної видимості вільних точок. Формати вхідних та вихідних даних розроблені таким чином, щоб уможливити просте використання множин точок і відповідних породжених многокутників для перевірки алгоритмів обчислювальної геометрії.

12. Застосовано практичну реалізацію генератора геометричних об’єктів до верифікації низки алгоритмів єдиного алгоритмічного середовища розв’язування комплексу задач обчислювальної геометрії, що розробляється на кафедрі математичної інформатики факультету кібернетики Київського національного університету імені Тараса Шевченка та задач, які розв’язуються в рамках проектів «Rectilinear Crossing Number Project» і «ComPoSe: Combinatorics on Point Sets and Arrangements of Objects» [2]. Зокрема, за допомогою точного розв’язку Задачі 1 виконано оцінку ефективності жадібного алгоритму пошуку простого многокутника мінімальної площі представленому в публікації [49]. Доведено з використанням точного розв’язку Задачі 1 та діаграми еквівалентності зіркових розбиттів, що максимальна кількість можливих простих многокутників для будь-якої множини з 7 точок не може перевищувати 92. Шляхом вичерпного обчислювального експерименту показано, що для інтервалу $N=4...7$ виконується наступне: кожна оптимальна конфігурація для N+1 точок може бути отримана з оптимальної конфігурації для N точок з використанням діаграми еквівалентності зіркових розбиттів для вичерпного перебору усіх можливих типів розташування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ahsan M., Cheruvatur S., Gamboni M., Garg A., Grishin O., Hashimoto S., Jermsurawong J., Toussaint G. T., Zhang L., A Simple Algorithm for Computing a Spiral Polygonization of a Finite Planar Set /M. Ahsan, S. Cheruvatur, M. Gamboni, A. Garg, O. Grishin, S. Hashimoto, J. Jermsurawong, G. T. Toussaint, L. Zhang // In Proceedings of The 24th Annual Fall Workshop on Computational Geometry, October 2014. – P.5-6.
2. Aichholzer O., Enumerating Order Types for Small Point Sets with Applications / O. Aichholzer // Режим доступу:
3. Ajtai M., Crossing-free subgraphs. / M. Ajtai, V. Chvátal, M. M. Newborn, and E. Szemerédi. // In Theory and Practice of Combinatorics, volume 12 of Annals of Discrete Mathematics and volume 60 of North-Holland Mathematics Studies. – 1982. – P. 9-12.
4. Akl S. A lower bound on the maximum number of crossing-free Hamiltonian cycles in a rectilinear drawing of Kn. / S. Akl. // Ars Combinatoria, volume 7 – 1979. – P. 7–18.
5. Nourollah A., A Genetic Based Algorithm to Generate Random Simple Polygons Using a New Polygon Merge Algorithm / Ali Nourollah, Mohsen Movahedinejad // World Academy of Science, Engineering and Technology; International Journal of Computer, Electrical, Automation, Control and Information Engineering Vol 9, No 1, 2015. – P.230-236.
6. Alt H., On the number of simple cycles in planar graphs. / H. Alt, U. Fuchs, K. Kriegel. // Combinatorics, Probability & Computing, volume 8, number 5, 1999. – P. 397–405.
7. Auer T., Heuristics for the generation of random polygons / T. Auer, M. Held // In Proceedings of the 8th Canadian Conference on Computational Geometry, Ottawa, Ontario. – 1996. – P. 38-43.
8. Auer T., Heuristics for the generation of random polygons / T. Auer // Dimplomarbeit zur Erlangung des Diplomingemieurgrades an der

Naturwissenschaftlichen Fakultät der Universität Salzburg. – 1996 //Режим доступу:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.6026&rep=rep1&type=pdf

9. Buchin K., On the number of cycles in planar graphs / Kevin Buchin, Christian Knauer, Klaus Kriegel, André Schulz, Raimund Seidel // in Proceedings of the 13th International Computing and Combinatorics Conference. – 2007. – P. 97–107.
10. Černý J., Noncrossing Hamiltonian paths in geometric graphs / Jakub Černý, Zdeněk Dvořák, Vít Jelínek, Jan Kára // Discrete Applied Mathematics 155, – 2007. – P. 1096–1105.
11. Chazelle B. An Optimal Algorithm for Intersecting Line Segments in the Plane. / B. Chazelle, H. Edelsbrunner // Journal of the Association for Computing Machinery. Vol. 39. No. 1, – 1992. – P. 1-54.
12. Dailey D., Constructing Random Polygons / David Dailey, Deborah Whitfield // SIGITE '08 Proceedings of the 9th ACM SIGITE conference on Information technology education, 2008 - P. 119-124.
13. Dean J. A., Recognizing polygons, or how to spy / James A. Dean, Andrzej Lingas, Jörg-Rüdiger Sack // The Visual Computer, Volume 3, Issue 6, November 1988. – P. 344-355
14. Demaine E.D., The Open Problems Project. / Demaine E.D., J. S. B. Mitchell, J. // Режим доступу: http://cs.smith.edu/~orourke/TOPP/P16.html#Problem.16
15. Demaine E., Simple Polygonizations./ Demaine E.// http://erikdemaine.org/polygonization/
16. Deneen L., Polygonizations of point sets in the plane / Linda Deneen, Gary Shute // Discrete & Computational Geometry, Volume 3, Issue 1. March 1988. – P. 77-87.

17. Deneen L., Polygonizations of point sets in the plane / Linda Deneen, Gary Shute // Discrete & Computational Geometry, volume 3, number 1, 1988, pages 77–87.
18. Denny M., Encoding a triangulation as a permutation of its point set / M. Denny, C. Sohler // Proceedings of the 9th Canadian Conference on Computational Geometry. – 1997. – P. 39–43.
19. Dumitrescu A., On two lower bound constructions / Adrian Dumitrescu // Proceedings of the 11th Canadian Conference on Computational Geometry, Vancouver. – 1999.
20. Epstein P., Generating geometric objects at random / Peter Epstein // Master Degree Thesis, Carleton University, School of Computer Science, 1992. – P. 1-151.
21. Fekete S. P., On simple polygonalizations with optimal area / S. P. Fekete // Discrete & Computational Geometry, volume 23, number 1, 2000, pages 73–110.
22. García A., Lower bounds on the number of crossing-free subgraphs of Kn /A. García, M. Noy, A. Tejel // Proceedings of the 7th Canadian Conference on Computational Geoemtry. – 1995. – P. 97-102.
23. García A., A lower bound for the number of polygonizations of N points in the plane /A. García, J. Tejel// Ars Combinatoria, volume 49. – 1998. – P. 3–19.
24. Gritzmann P., Embedding a planar triangulation with vertices at specified points / P. Gritzmann, B. Mohar, J. Pach, R. Pollack // American Mathematical Monthly 98, – 1991. – P. 165–166.
25. Hayward R. B., A lower bound for the optimal crossing-free Hamiltonian cycle problem /Ryan B. Hayward// Discrete & Computational Geometry, volume 2, number 4. – 1987. – P. 327-343.
http://www.ist.tugraz.at/staff/aichholzer/research/rp/triangulations/ordertypes/

26. Hurtado F., On polygons enclosing point sets / F. Hurtado, C. Merino, D. Oliveros, T. Sakai, J. Urrutia, and I. Ventura // Graphs and Combinatorics, Volume 25, Issue 3, 2009. - P. 327-339.
27. Hurtado F., On Polyhedra Indued by Point Sets in Space / Ferran Hurtado, Godfried T. Toussaint, Joan Trias // Proceedings of 15th Canadian Conferene on Computational Geometry, Dalhousie University, Halifax, Nova Sotia, Canada, August 11-13, 2003. – P. 107-110.
28. Iwerks J., Spiral serpentine polygonization of a planar point set / Justin Iwerks, Joseph S. B. Mitchell // XIV Spanish Meeting on Computational Geometry, 27–30 June 2011. – P.146-150.
29. M. Damian, Connecting Polygonizations via Stretches and Twangs. / Damian M., Flatland R., O'Rourke J., Ramaswami S. // Theory of Computing Systems, 47(3), – 2010. – P. 674-695.
30. Mitchell J. S. B., Computational Geometry Column 42. A compendium of thirty previously published open problems in computational geometry is presented. / J. S. B. Mitchell, J. O'Rourke // International Journal of Computational Geometry and Applications, Vol. 11, No. 5. – 2001. – P. 573-582.
31. Mohammadi L., Generating Sunflower Random Polygons on a Set of Vertices / L. Mohammadi, A. Nourollah // Proceedings of the ICFCS, 2011. – P. 47-50.
32. Newborn M., Optimal crossing-free Hamiltonian circuit drawings of K_n /M. Newborn, W. O. J. Moser// Journal of Combinatorial Theory, Series B, volume 29. – 1980. – P. 13-26.
33. O'Rourke J., Generating Random Polygons. / J. O'Rourke, M. Virmani/ Smith College. Department of Computer Science, Northampton, MA. Technical Report (TR # 011), – 1991.
34. Pach J., Graphs drawn with few crossings per edge / János Pach, Géza Tóth // Combinatorica, volume 17, number 3. – 1997. – P. 427–439.

35. Papadimitriou C., The Euclidian Traveling Salesman Problem is NP-Complete / Christos Papadimitriou // Theoretical Computer Science 4, – 1977. – P. 237-244.
36. Rappaport D., On the complexity of computing orthogonal polygons from a set of points / David Rappaport // Technical Report SOCS-86.9, McGill University, Montréal, 1986.
37. Sachio T., Heuristics for Generating a Simple Polygonalization / Teramoto Sachio, Motoki Mitsuo, Uehara Ryuhei, Asano Tetsuo // Japan Advanced Institute of Science and Technology, IPSJ SIG Technical Report, January 2006. – P. 41-48.
38. Sadhu S., Random Polygon Generation using "GRP_AS" Heuristic / Sanjib Sadhu, Niraj Kumar // In Proceedings of 2013 Sixth International Conference on Contemporary Computing (IC3), August 8-10, 2013. – P.13-17.
39. Santos F., A better upper bound on the number of triangulations of a planar point set / Francisco Santos, Raimund Seidel // Journal of Combinatorial Theory, Series A, volume 102, number 1. – 2003. – P. 186-193.
40. Seidel R., On the number of triangulations of planar point sets. / Raimund Seidel // Combinatorica, volume 18, number 2. – 1998. – P. 297-299.
41. Sharir M., Counting triangulations of planar point sets / Micha Sharir, Adam Sheffer // arXiv:0911.3352. First version November 2009, revision January 2010.
42. Sharir M., Counting plane graphs: perfect matchings, spanning cycles, and Kasteleyn's technique / Micha Sharir, Adam Sheffer, Emo Welzl // arXiv:1109.5596, September 2011. [Originally announced during an invited talk by Emo Welzl at 23rd Canadian Conference on Computational Geometry, Toronto, Canada, August 2011.]
43. Sharir M., On the number of crossing-free matchings, cycles, and partitions / Micha Sharir and Emo Welzl // Manuscript. – July 2005.
44. Smith W. D., Studies in Computational Geometry motivated by mesh generation / W. D. Smith// Ph.D. Thesis, Princeton University. –1989.

45. Sohler C., Generating random star-shaped polygons / Christian Sohler // In Proceedings of the 11th Canadian Conference on Computational Geometry (CCCG'99), 1999.
46. Steinhaus H. One Hundred Problems in Elementary Mathematics. / H. Steinhaus // Dover Publications Inc. New York, 1964. – P. 85-86 (https://books.google.com.ua/books?id=IARoSNX7hE0C)
47. Taherkhani F., The generation of pseudo-triangulated spiral polygon using convex hull layers / F. Taherkhani, A. Nourollah // Proceedings of the Proceedings of the ICFCS, 2011. – P. 51-56
48. Tereshchenko V., Algorithm for Finding the Domain Intersection of a Set of Polytopes / V. Tereshchenko, S. Chevokin, A. Fisunenko // Original Research Article: Procedia Computer Science, Volume 18, Pages 459-464, Elsevier, 2013, ISSN:1877-0509, ISBN: 978-1-62748-825-9. (SCOPUS Author ID: 55433978800, A. Fisunenko).
49. Tereshchenko V., Generating a simple polygonizations / V. Muravitskiy, V. Tereshchenko // In Proceedings of 15th International Conference on Information Visualisation, 2011, – P. 502-506.
50. Tereshchenko V., Pilipenko S., Fisunenko A. Domain Triangulation between Convex Polytopes. // Original Research Article: Procedia Computer Science, Volume 18, Pages 2500-2503, Elsevier, 2013: ISSN: 1877-0509, ISBN: 978-1-62748-825-9 (SCOPUS Author ID: 55433978800, A. Fisunenko).
51. Tereshchenko V., Socolov O., Fisunenko A. Solving the Range Searching Problem for Region Bounded by a Convex Surface // *Proceedings of the 16th International Conference on Information Visualisation IV2012*, Montpellier, France, 11-13 July 2012, P. 491- 494, ISSN:1550-6037,ISBN: 978-1-4673-2260-7 (SCOPUS Author ID: 55433978800, A. Fisunenko).
52. Tomas A., Bajuelos A.. Generating Random Orthogonal Polygons. / Tomas A., Bajuelos A. // Current Topics in Artificial Intelligence. Lecture Notes in Computer Science, Volume 3040. - 2004. – P. 364-373

53. van Leeuwen J., Schoone A. A., Untangling a traveling salesman tour in the plane / Jan Van Leeuwen, Anneke A. Schoone // In Proceedings of 7th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'81), 1981.
54. Vasyl Tereshchenko. The system of common algorithmic space to create visual models of phenomena and processes. In Proc. International Conference: Applications of Computer and Information Sciences to Nature Research (ACISNR 2010). State University of New York at Fredonia, New York, USA May 5-7, 2010, P. 64-68.
55. Zhu C., Sundaram G., Snoeyink J., Mitchell J. S. B., Generating random polygons with given vertices / C. Zhu, G. Sundaram, J. Snoeyink, J. S. B. Mitchell // *Computational Geometry: Theory and Applications*, Volume 6, Issue 5, September 1996. – P. 277-290.
56. Анісімов А. В., Терещенко В. Н., Рекурсия и параллельные алгоритмы в задачах геометрического моделирования / А. В. Анісімов, В. Н. Терещенко // Кибернетика и системный анализ. – 2010 – №2.– С. 9-22.
57. Анісімов А. В., Терещенко В. М., Кравченко І. В. Основні алгоритми обчислювальної геометрії. Навчальний посібник. / А. В. Анісімов, В. М. Терещенко, І. В. Кравченко // Київський Національний університет імені Тараса Шевченка. Видавничо-поліграфічний центр «Київський університет» – 2002. – 81 с.
58. Препарата Ф., Шеймос М. Вычислительная геометрия: Введение. / Ф. Препарата, М. Шеймос // Пер. с англ. – М.: Мир, 1989. – 486 с.
59. Седжвик Р. Алгоритмы на С++. / Р. Седжвик // Пер. с англ. – М. ООО «И. Д. Вильямс», 2014. – 1056 с.
60. Седжвик Р., Алгоритмы на С++, анализ, структуры данных, сортировка, поиск, алгоритмы на графах / Р. Седжвик // Перевод с английского. – Москва, Издательский дом «Вильямс», 2014. – 1056. – С. 726-732.

61. Терещенко В. М., Фісуненко А. Л., Особливості розв'язання задач регіонального пошуку для d-вимірного випадку/ Терещенко В. М., Фісуненко А. Л.// Вісник Київського університету. Серія: фізико-математичні науки. – 2012. – №4. – С. 207-210.
62. Фісуненко А. Л., Модель мультиагентної системи обробки візуальної інформації для вирішення задач реконструкції / Фісуненко А. Л. // Вісник Київського університету. Серія: фізико-математичні науки. – 2012. – №3. – С. 261-264.
63. Фісуненко А. Л., Генерація простих многокутників вирізанням та нарощуванням опуклих оболонок / А. Л. Фісуненко // Вісник Київського національного університету імені Тараса Шевченка, серія фізико-математичні науки. – 2013. -спецвипуск. – С. 190-193.
64. Харченко В. С., Безопасность критических инфраструктур: математические методы анализа и обеспечения / Под. ред. Харченко В. С. // Министерство образования и науки Украины, Национальный аэрокосмический университет им. Н. Е. Жуковского «ХАИ». – 2011.– С. 490-523.
65. Швед Д., Поиск точек раздела, мостов и двусвязных компонент / Д. Швед // Режим доступу до статті: http://acm.mipt.ru/twiki/bin/view/Algorithms/ArticulationPointsAndBridges

ДОДАТКИ

Додаток А. Порівняння точного і наближеного розв'язків задачі про простий многокутник найменшої площі.

Таблиця Д-А.1*

Кількість точок вхідної множини, ***N***	Точок на ОО, *h*	Кількість простих многокутників, $/SP(S)/$	Площа простого многокутника мінімальної площі (точний розв'язок), S_{min}	Площа простого многокутника мінімальної площі (розв'язок жадібним алгоритмом), S_{greedy}	Відношення мін. площі, отриманої точним до площі жадібним алгоритмом, S_{min}/ S_{greedy}
1	**2**	**3**	**4**	**5**	**6**
16	9	125854	171192	196323	1.15
16	10	47101	295372	345389	1.17
17	9	508495	331858	345556	1.04
17	10	213671	327640	425249	1.30
18	9	2226086	236159	337642	1.43
18	10	818345	213589	224548	1.05
19	9	9567358	201426	258384	1.28
19	10	4046155	208838	301520	1.44
20	9	41288158	168924	270020	1.60
20	10	18400407	185214	235114	1.27
21	9	184898420	163544	234556	1.43
21	10	85119467	201439	248143	1.23
22	9	393135487	161256	220587	1.37
22	10	804779826	174647	285364	1.63

*Див. також Таблицю Д-А.3: результати обчислень, що надані в Таблиці Д-А.1, були отримані на тестових множинах точок з Таблиці Д-А.3.

Таблиця Д-А.2*

N=16						
h	8	7	6	5	4	3
$\|SP(S)\|$	322658	676983	1586474	3622814	9111930	18233275
S_{min}	176960	129078	111130	90325.5	63976.5	70134
S_{greedy}	268506	188031	174086	112009	84407.5	122578
S_{min}/S_{greedy}	1.52	1.46	1.57	1.24	1.32	1.75
N=17						
h	8	7	6	5	4	3
$\|SP(S)\|$	1290789	3105093	5931865	6369585	24334845	62211616
S_{min}	164506	113580	100605	162418	146769	70761
S_{greedy}	219598	159051	148424	174642	210705	82306
S_{min}/S_{greedy}	1.33	1.40	1.48	1.08	1.44	1.16
N=18						
h	8	7	6	5	4	3
$\|SP(S)\|$	5291989	13911273	26552115	66293008	122836043	281922573
S_{min}	167940	109357	122324	114116	95821	91750.5
S_{greedy}	215674	224003	212269	139537	208707	133615
S_{min}/S_{greedy}	1.28	2.05	1.74	1.22	2.18	1.46
N=19						
h	8	7	6	5	4	3
$\|SP(S)\|$	22188659	59816602	128556413	227067092	692189796	1854180859
S_{min}	150130	142384	125324	86896	70911.5	50635.5
S_{greedy}	210854	180430	198287	164609	88336	87274
S_{min}/S_{greedy}	1.40	1.27	1.58	1.89	1.25	1.72

****Примітка**:* Позначення величин в 1-му стовпці відповідають позначенням в Таблиці Д-А.1.

Таблиця Д-А.3. Тестові набори даних.

К-сть точок, *N*	К-ість Точок ОО, *H*	Кількість простих много-кутників, */SP(S)/*	Координати точок вхідної множини; Координати простого многокутника мінімальної площі (точний розв’язок) (X_0 Y_0 X_1 Y_1 … X_{N-1} Y_{N-1})
1	**2**	**3**	**4**
16	9	125854	421 463 570 244 876 91 1265 169 1440 445 1334 719 1083 919 750 910 484 751 1185 777 985 143 1004 569 959 385 914 477 787 527 548 580; 421 463 484 751 787 527 750 910 959 385 1083 919 1334 719 1440 445 1185 777 1004 569 985 143 1265 169 876 91 914 477 548 580 570 244
16	10	47101	333 464 496 238 769 120 1167 143 1429 387 1440 618 1289 947 965 1019 626 965 420 787 1235 706 959 176 956 511 945 378 774 529 548 643; 333 464 420 787 774 529 626 965 956 511 965 1019 959 176 1289 947 1440 618 1429 387 1235 706 1167 143 769 120 945 378 548 643 496 238
17	9	508495	317 693 398 398 628 141 865 66 1200 68 1490 189 1470 807 1179 945 705 943 1427 570 1177 654 1141 355 985 273 1010 584 865 754 697 422 574 655; 317 693 1010 584 398 398 985 273 628 141 865 66 1427 570 1200 68 1490 189 1470 807 1141 355 697 422 1177 654 865 754 1179 945 574 655 705 943
17	10	213671	422 306 662 104 1011 75 1412 121 1566 379 1580 691 1459 931 1014 1037 539 948 431 660 1370 656 1149 374 1065 566 888 719 787 611 736 448 743 622; 422 306 736 448 662 104 888 719 1065 566 1011 75 1370 656 1412 121 1566 379 1580 691 1459 931 1149 374 1014 1037 743 622 539 948 431 660 787 611

1	2	3	4
18	9	2226086	327 622 348 424 537 224 804 112 1327 235 1465 628 1286 966 779 1008 433 835 1249 664 1023 438 1142 788 926 512 932 655 969 875 777 423 775 745 597 543; 327 622 926 512 775 745 433 835 1249 664 779 1008 969 875 1286 966 1142 788 1465 628 932 655 1327 235 1023 438 777 423 804 112 597 543 537 224 348 424
18	10	818345	254 407 337 196 504 82 939 107 1175 374 1167 646 972 821 615 826 346 780 269 590 1012 528 937 312 647 265 710 554 636 431 803 713 440 341 468 661; 254 407 440 341 803 713 269 590 346 780 615 826 468 661 972 821 710 554 647 265 1167 646 1175 374 1012 528 939 107 937 312 504 82 636 431 337 196
19	9	9567358	330 491 511 191 952 39 1406 269 1529 587 1356 895 1010 1034 613 981 384 805 1219 222 1174 311 1098 460 1096 641 1115 832 1150 940 998 645 865 531 709 455 506 367; 330 491 511 191 709 455 1096 641 1098 460 1219 222 952 39 1406 269 1529 587 1174 311 1115 832 1356 895 1010 1034 613 981 1150 940 998 645 865 531 506 367 384 805
19	10	4046155	284 534 438 279 722 159 1132 171 1450 513 1372 915 918 1057 558 969 421 862 332 711 950 251 1004 608 940 695 1067 788 1269 873 822 622 788 537 768 719 468 781; 284 534 332 711 421 862 788 537 558 969 768 719 1269 873 918 1057 1372 915 1067 788 940 695 1004 608 1450 513 822 622 1132 171 722 159 438 279 950 251 468 781

1	2	3	4
20	9	41288158	319 566 427 303 630 132 949 77 1328 334 1400 761 1184 1047 756 1044 454 847 1076 292 987 603 899 421 992 854 891 620 827 536 874 737 1051 1005 749 602 661 483 446 476; 319 566 661 483 827 536 454 847 749 602 1051 1005 756 1044 1184 1047 992 854 1400 761 874 737 1328 334 987 603 1076 292 891 620 949 77 899 421 446 476 630 132 427 303
20	10	18400407	276 591 383 334 606 121 910 81 1329 343 1376 647 1317 931 1048 1029 685 1024 395 860 1229 886 882 129 880 262 854 380 915 691 807 495 769 593 695 684 613 745 472 823; 276 591 613 745 395 860 685 1024 472 823 915 691 1229 886 1048 1029 1317 931 769 593 1376 647 807 495 1329 343 854 380 880 262 910 81 606 121 383 334 882 129 695 684
21	9	184898420	319 566 427 303 630 132 949 77 1328 334 1400 761 1184 1047 756 1044 454 847 1076 292 996 492 987 603 899 421 992 854 891 620 827 536 874 737 1051 1005 749 602 661 483 446 476; 319 566 661 483 827 536 454 847 749 602 1051 1005 756 1044 1184 1047 992 854 1400 761 874 737 1328 334 987 603 1076 292 996 492 891 620 949 77 899 421 446 476 630 132 427 303
21	10	85119467	296 450 444 214 731 67 1203 179 1436 455 1347 778 1098 993 660 987 428 847 327 662 1119 268 1026 320 966 433 897 576 826 342 880 758 815 651 721 507 912 962 609 397 468 325; 296 450 609 397 1119 268 327 662 1026 320 966 433 721 507 912 962 428 847 660 987 1098 993 880 758 1347 778 815 651 1436 455 897 576 1203 179 826 342 468 325 731 67 444 214

1	2	3	4
22	9	804779826	319 566 427 303 630 132 949 77 1328 334 1400 761 1184 1047 756 1044 454 847 1076 292 996 492 988 363 987 603 899 421 992 854 891 620 827 536 874 737 1051 1005 749 602 661 483 446 476; 319 566 661 483 899 421 454 847 749 602 1051 1005 756 1044 1184 1047 992 854 874 737 891 620 1400 761 987 603 996 492 1328 334 827 536 1076 292 949 77 988 363 446 476 630 132 427 303
22	10	393135487	296 450 444 214 731 67 1203 179 1436 455 1347 778 1098 993 660 987 428 847 327 662 1119 268 1026 320 966 433 897 576 856 456 826 342 880 758 815 651 721 507 912 962 609 397 468 325; 296 450 444 214 609 397 856 456 731 67 826 342 966 433 1026 320 1203 179 1436 455 1119 268 897 576 1347 778 721 507 1098 993 660 987 428 847 912 962 880 758 815 651 468 325 327 662

Додаток Д-А.4 (Доповнення до Рисунку 5.6)
Набір даних для оптимального розташування для N=6 (кількість можливих простих многокутників |SP(S)| = 29; див. Рисунок 5.6):
6
463 991 706 73 1476 565 855 348 1237 547 695 755

Примітка:

Області еквівалентності для пошуку оптимальної конфігурації для N=7 утворюються на гранях розташування, яке породжується шляхом об’єднання усіх відрізків та променів, що утворені перетинами прямих на усіх можливих парах точок. В області еквівалентності за 7-му точку може бути обрана довільна внутрішня точка грані, наприклад, центроїд будь-яких трьох вершин грані у випадку коли грань є простим многокутником. Якщо грань є розімкненою областю, то для обчислення внутрішньої точки обирається вершина кута і дві довільні точки на променях, що обмежують область еквівалентності. Для обраних трьох точок обчислюються координати центроїда, який і обирається як 7-ма точка. Для множини з отриманих 7-ми точок виконується Задача 3.

```
// Реалізація жадібного алгоритму пошуку простого многокутника мініальної площі з
// Tereshchenko V.,
// Generating a simple polygonizations/ V. Muravitskiy, V. Tereshchenko //
// In Proceedings of 15th International Conference on Information Visualisation, 2011, –
// P. 502-506.

#pragma once

#include <vector>
#include "utils.h"
#include "ConvexHullIncrementalAlgorithm.h"

namespace golib
{

	template <class Point>
	class GreedyMinAreaByTM
	{
		typedef std::vector<Point> Points;
		typedef ConvexHullIncrementalAlgorithm<Point> ConvexHullBuilder;

		Points list;
		Points edges;

		bool hasPointOrIntersects(Point& p1, Point& p2, Point& p3){
			int k = 0;
			for (int i = 0; i < edges.size(); i++){
				k = (i == edges.size() - 1) ? 0 : i + 1;
				CrossingType tt = crossing(edges[i], edges[k], p1, p3);
				if (tt == ctInBounds){
					return true;
				}
				if (tt == ctOnBounds){
					if (!(edges[i] == p1) && !(edges[i] == p3) &&
						!(edges[k] == p1) && !(edges[k] == p3))
						return true;
				}
				tt = crossing(edges[i], edges[k], p2, p3);
				if (tt == ctInBounds){
					return true;
				}
				if (tt == ctOnBounds){
					if (!(edges[i] == p2) && !(edges[i] == p3) &&
						!(edges[k] == p2) && !(edges[k] == p3))
						return true;
				}
			}
			for (int i = 0; i < list.size(); i++){
				if (pointBelongsToTriangle<Point>(p1, p2, p3, list[i]))
					return true;
			}

			return false;
		}

		void runGreedyAlgorithm(){
			if (edges.size() < 3)
				return;
			int  k, max = 0, after = 0;
			double maxSquare = -1.0;
			double tmpSquare;

			while (list.size() > 0)
			{
```

```
                        maxSquare = -1.0;
                        //Перебираємо усі ребра
                        for (int i = 0; i < edges.size(); i++){
                                k = (i == edges.size() - 1) ? 0 : i + 1;
                                //Перебираємо усі вільні точки
                                for (int j = 0; j < list.size(); j++) {
                                        Point p1 = edges[i];
                                        Point p2 = edges[k];
                                        Point p3 = list[j];
                                        if (((tmpSquare = area_of_triangle(p1, p2, p3))
> maxSquare) && !hasPointOrIntersects(p1, p2, p3)) {
                                                maxSquare = tmpSquare;
                                                max = j;
                                                after = i;
                                        }
                                }
                        }
                        Point mp = list[max];
                        list[max] = list[list.size() - 1];
                        list.pop_back();
                        Points::iterator it = edges.begin();
                        edges.insert(it + after + 1, mp);
                }
        }

    public:
        GreedyMinAreaByTM() {}

        Points& getMinAreaPolygon(Points& ps){
                list.clear();
                edges.clear();
                ConvexHullBuilder chb;
                chb.build(ps, edges, list);
                runGreedyAlgorithm();
                return edges;
        }
    };
}
=================================================================================
```

Примітки:

1) Процедура crossing визначає взаємне розташування відрізків:
2) ctInBounds – точка перетину відрізків всередині відрізків;
3) ctOnBounds – відрізки мають спільну точку-кінець відрізку;
4) Процедура pointBelongsToTriangle визначає чи належить трикутнику (перші три параметри процедури) задана точка (четвертий параметри процедури);
5) Процедура area_of_triangle обчислює площу трикутника, який задається 3-ма заданими точками (параметри процедури)

Рисунок Д-А.3. *Вихідний код основної процедури жадібного алгоритму пошуку простого многокутника мінімальної площі*.

Додаток Б. Вихідний код окремих частин практичної реалізації генератора геометричних об’єктів.

```
1   //****************************************************************************
2   // File: SimplePolygonGenerator.h
3   // Author: Andriy Fisunenko
4   // Description: Implementation of the main algorithm of exhaustive search
5   // with optimizations and backtracking techniques
6   //****************************************************************************
7   #pragma once
8
9   #include <windows.h>
10  #include <time.h>
11  #include <stdlib.h>
12  #include <vector>
13  #include <fstream>
14  #include "utils.h"
15  #include "Array16.h"
16  #include "ConvexHullIncrementalAlgorithm.h"
17  #include "PointsVisibilityGraph.h"
18  #include "BooleanArray64.h"
19  #include "Globals.h"
20  #include "PairsArray.h"
21
22  namespace golib
23  {
24     //частина даних та методів цього класу відкинута для скорочення;
25     template <class Point>
26     class SimplePolygonGenerator
27     {
28     protected:
29            typedef std::vector<Point> Points;
30            typedef ConvexHullIncrementalAlgorithm<Point> ConvexHullBuilder;
31            //It is estremally redundant comapring to STL's containers but it gives
    much faster code
32            CrossingType ct[MAX_POINTS][MAX_POINTS][MAX_POINTS][MAX_POINTS];
33            SafePairsArray pairs[MAX_POINTS][MAX_POINTS];
34            size_t pidx[MAX_POINTS];
35            size_t levelVisits[MAX_POINTS+2];
36            size_t blockingChainsOnLevel[MAX_POINTS];
37            size_t articulationPoints[MAX_POINTS];
38            size_t level;
39            bool pointIsHullVertex[MAX_POINTS];
40            bool pointIsInTheChain[MAX_POINTS];
41            size_t hullPointsInTheChain[MAX_POINTS];
42            size_t numOfHullPointsInTheChain;
43            inline void addPointToHullPointsInTheChain(size_t idx){
44                   hullPointsInTheChain[numOfHullPointsInTheChain] = idx;
45                   numOfHullPointsInTheChain++;
46            }
47            inline void removeLastAddedPointFromHullPointsInTheChain(){
48                   if (numOfHullPointsInTheChain > 0)
49                          numOfHullPointsInTheChain--;
50            }
51            bool compliesAuersLemma2_11(size_t idx){
52                   if (pointIsHullVertex[idx]) {
53                          size_t lastHullVertexIdx = hull.size() - 1;
54                          if (idx < lastHullVertexIdx){
55                                 //check either the last vertex of the hull is already
    in the chain and set a direction
56                                 //for checking conditions of Auer's Lemma 2.11
57                                 int offset = (pointIsInTheChain[lastHullVertexIdx]) ? 1
    : -1;
```

```
58                              if (!pointIsInTheChain[idx + offset])
59                              {
60                                     return false;
61                              }
62                       }
63                }
64                return true;
65         }
66         Points points, hull, interior;
67         size_t blockingChains;
68         size_t nonTrivialBccCounter;
69         PointsVisibilityGraph<MAX_POINTS> vgraph;
70         Points minimalAreaPolygon, randomPolygon;
71         std::vector<Points> polygons;
72         double minArea;
73         bool first, plPermuted;
74         size_t sizeOfPointSet, MaxIdx, MaxIdxPrior, count;
75         void makeAllPermutations3Halved(size_t m , BooleanArray64
   pointIsVisibleFromChainHead){
76                levelVisits[level]+=1;
77                level++;
78                if (m == sizeOfPointSet) {
79                       //if (pointCanKeepSimplicityOfChain(m, 0))
80                       if (pointIsVisibleFromChainHead[0]) {
81                              //...Do processing of obtained polygon
82                              count += 1;
83                       }
84                }
85                else{
86                       size_t prevIdx = m - 1;
87                       pointIsInTheChain[pidx[prevIdx]] = true;
88                       for (size_t i = m; i < sizeOfPointSet; i++){
89                              size_t nextPoint = pidx[i];
90                              if (pointIsVisibleFromChainHead[nextPoint]){
91                                     BooleanArray64 nextVisiblePoints(false);
92                                     SafePairsArray edgesOfVisibilityGraphToRestore;
93                                     if (pointCanKeepSimplicityOfChain(m, i,
   edgesOfVisibilityGraphToRestore, nextVisiblePoints)){

94                                            if (i == m){

95                                                   makeAllPermutations3Halved(m + 1,
   nextVisiblePoints);
96                                            }
97                                            else{
98                                                   size_t d = MaxIdx - nextPoint;
99                                                   if ((d > 1) || (d == 0 &&
   plPermuted)){
100                                                         size_t v = pidx[m];
101                                                         pidx[m] = nextPoint;
102                                                         pidx[i] = v;
103                                                         makeAllPermutations3Halved(m
   + 1, nextVisiblePoints);
104                                                         pidx[i] = nextPoint;
105                                                         pidx[m] = v;
106                                                  }
107                                                  else if (d == 1){
108                                                         plPermuted = true;
109                                                         size_t v = pidx[m];
110                                                         pidx[m] = nextPoint;
111                                                         pidx[i] = v;
112                                                         makeAllPermutations3Halved(m
   + 1, nextVisiblePoints);
113                                                         pidx[i] = nextPoint;
```

```
114                                                     pidx[m] = v;
115                                                     plPermuted = false;
116                                               }
117                                         }
                                      }
    restoreVisibilityGraph(edgesOfVisibilityGraphToRestore);
118                         }//if(enabledPoints[pidx[i]])
119                   }
120                   pointIsInTheChain[pidx[prevIdx]] = false;
121             }
122             level--;
123       }
124
125       inline void restoreVisibilityGraph(SafePairsArray& pairs)
126       {
127             for (size_t i = 0; i < pairs.size(); i++)
128                   vgraph.addEdge(pairs[i].idx1, pairs[i].idx2);
129       }
130
131       bool genIndexOfTheNextVisiblePointRandomly(size_t& idx,
  std::vector<size_t>& indices)
132       {
133             if (indices.size() == 0)
134                   return false;
135             if (indices.size() == 1) {
136                   idx = indices[0];
137                   indices.clear();
138             }
139             else
140             {
141                   size_t r = (size_t)(rand()%indices.size());
142                   idx = indices[r];
143
144                   indices[r] = indices[indices.size()-1];
145                   indices.pop_back();
146             }
147             return true;
148       }
149       void randomGenerationMove(
150              size_t m, BooleanArray64 pointIsVisibleFromChainHead){
151             levelVisits[level] += 1;
152             level++;
153             if (m == sizeOfPointSet){
154                   if (pointIsVisibleFromChainHead[0]){
155                         Points polygon;
156                         for (size_t i = 0; i < points.size(); i++){
157                               polygon.push_back(points[pidx[i]]);
158                         }
159                         polygons.push_back(polygon);
160                   }
161             }
162             else{
163                   size_t prevIdx = m - 1;
164                   pointIsInTheChain[pidx[prevIdx]] = true;
165
166                   std::vector<size_t> visiblePointIdx(sizeOfPointSet);
167                   size_t i;
168                   for (i = m; i < sizeOfPointSet; i++)
169                   {
170                         size_t nextPoint = pidx[i];
171                         if (pointIsVisibleFromChainHead[nextPoint])
172                               visiblePointIdx.push_back(i);
173                   }
174
```

```
175                         while(genIndexOfTheNextVisiblePointRandomly(i,
   visiblePointIdx) && polygons.size()==0) {
176                                 size_t nextPoint = pidx[i];
177                                 if (pointIsVisibleFromChainHead[nextPoint]){
178                                         BooleanArray64 nextVisiblePoints(false);
179                                         SafePairsArray edgesOfVisibilityGraphToRestore;
180
181                                         if (pointCanKeepSimplicityOfChain(m, i,
   edgesOfVisibilityGraphToRestore, nextVisiblePoints)) {
182                                                 if (i == m){
183                                                         randomGenerationMove(m + 1,
   nextVisiblePoints);
184                                                 }
185                                                 else{
186                                                         size_t d = MaxIdx - nextPoint;
187                                                         if ((d > 1) || (d == 0 &&
   plPermuted)) {
188                                                                 size_t v = pidx[m];
189                                                                 pidx[m] = nextPoint;
190                                                                 pidx[i] = v;
191
192                                                                 randomGenerationMove(m + 1,
   nextVisiblePoints);
193                                                                 pidx[i] = nextPoint;
194                                                                 pidx[m] = v;
195
196                                                         }
197                                                         else if (d == 1) //if (pidx[i] ==
   MaxIdxPrior){
198                                                                 plPermuted = true;
199                                                                 size_t v = pidx[m];
200                                                                 pidx[m] = nextPoint;
201                                                                 pidx[i] = v;
202                                                                 randomGenerationMove(m + 1,
   nextVisiblePoints);
203                                                                 pidx[i] = nextPoint;
204                                                                 pidx[m] = v;
205                                                                 plPermuted = false;
206                                                         }//the case if(pidx[i] == MaxIdx &&
   !plPermuted) is skipped
207                                                 }
208                                         }//if (pointCanKeepSimplicityOfChain(m, i,
   nextVisiblePoints))
209
      restoreVisibilityGraph(edgesOfVisibilityGraphToRestore);
210                                 }//if(enabledPoints[pidx[i]])
211                         }
212                         pointIsInTheChain[pidx[prevIdx]] = false;
213                 }
214                 level--;
215         }
216         inline bool thereIsAnArticulationPointOnTheHull(){
217                 for (size_t i = 1; i < hull.size(); i++) {
218                         if (vgraph.isArticulationPoint(i) &&
   vgraph.getVertexDegree(i)>2)
219                                 return true;
220                 }
221                 return false;
222         }
223         inline bool pointCanKeepSimplicityOfChain(const size_t chainSize, size_t
   nextPointIdx, SafePairsArray& edgesToRestore, BooleanArray64& nextVisiblePoints){
224                 size_t idx = pidx[nextPointIdx];
225                 size_t idxPrior = pidx[chainSize - 1];
226
```

```
227                 if (!compliesAuersLemma2_11(idx))
228                 {
229                       processBlockingChain();
230                       return false;
231                 }
232
233                 //Remove edges from the previous head of chain (entry point to
   visibility graph)
234                 //and store them for further restoring after processing pair pidx[m]
   and pidx[i]
235                 //1st segment of the simple chain is a special case (it is the only
   segment to remove from vgraph)
236                 if (idxPrior == 0) {
237                       if (vgraph.removeEdge(idxPrior, idx))
238                             edgesToRestore.addPair(idxPrior, idx);
239                 }
240                 else{
241                       //the head of the built simple chain cannot go strait to the
   tail without visiting all free points
242                       if (chainSize < points.size() - 1) {
243                             if(vgraph.removeEdge(0, idx))
244                                   edgesToRestore.addPair(0, idx);

245                       }
246                       for (size_t i = 0; i < vgraph.getVertexDegree(idxPrior); i++){
247                             if (vgraph.edgeExists(idxPrior,
   vgraph.getAdjacentVertex(idxPrior, i)))
248                                   edgesToRestore.addPair(idxPrior,
   vgraph.getAdjacentVertex(idxPrior, i));
249                       }
250                       vgraph.cleanAllEdgesAt(idxPrior);
251                 }
252                 //(to optimize by analysing situation of VG)
253                 //remove all edges intersecting our newly added edge
254                 SafePairsArray& intersectedSemgments = pairs[idx][idxPrior];
255                 for (size_t i = 0; i < intersectedSemgments.size(); i++){
256                       size_t from = intersectedSemgments[i].idx1;
257                       size_t to = intersectedSemgments[i].idx2;
258                       if (vgraph.removeEdge(from, to))
259                             edgesToRestore.addPair(from, to);
260                 }
261                 if (vgraph.getVertexDegree(0) == 0) {
262                       processBlockingChain();
263                       return false; // nothing to test further
264                 }
265                 vgraph.removeRedundantEdges(points.size(), idx, edgesToRestore);
266                 if (invalidVertexExists(idx)) {
267                       processBlockingChain();
268                       return false;
269                 }
270
271                 if (!vgraph.isConnected(points.size(), chainSize, idx))
272                 {
273                       processBlockingChain();
274                       return false;
275                 }
276                 vgraph.resetSearch(sizeOfPointSet);
277
278                 //*********** New section start (according Tomas Auers' paper)
279                 if(!vgraph.compliesAuersLemma2_9(points.size(), idx))
280                 {
281                       processBlockingChain();
282                       return false;
283                 }
```

```
284
285                 if (!vgraph.compliesAuersLemma2_10(idx))
286                 {
287                       processBlockingChain();
288                       return false;
289                 }
290                 //*********** End of New section end (according Tomas Auers' paper)
291                 vgraph.findArticulationPoints(idx);
292                 if (vgraph.isArticulationPoint(idx) || vgraph.isArticulationPoint(0)
    || vgraph.hasTerminalComponent()){
293                       processBlockingChain();
294                       return false;
295                 }
296                 size_t numOfAp = vgraph.getNumberOfArticulationPoints();
297                 vgraph.removeRedundantEdgeFromTheTail(edgesToRestore);
298                 bool nonTrivialCase = vgraph.getVertexDegree(0) != 1 &&
    vgraph.getVertexDegree(idx) != 1;
299                 //Test special case of vgraph geometry ("arrows" - splitting vgraph
    onto two parts)
300                 if (chainSize < points.size() - 2){
301                       SafePairsArray tailEdges;
302                       for (size_t i = 0; i < vgraph.getVertexDegree(0); i++){
303                             if (vgraph.edgeExists(0, vgraph.getAdjacentVertex(0,
    i)))
304                                   tailEdges.addPair(0, vgraph.getAdjacentVertex(0,
    i));
305                       }
306                       vgraph.cleanAllEdgesAtTail();
307                       vgraph.resetSearch(sizeOfPointSet);
308                       vgraph.findArticulationPoints(idx);
309                       restoreVisibilityGraph(tailEdges);
310                       if (vgraph.isArticulationPoint(idx)){
311                             processBlockingChain();
312                             return false;
313                       }
314                 }
315                 if (chainSize < points.size() - 2){
316                       for (size_t i = 2; i < sizeOfPointSet; i++){
317                             for (size_t j = 1; j < i; j++){
318                                   if (vgraph.getVertexDegree(i) == 2 &&
    vgraph.getVertexDegree(j) == 2 && vgraph.edgeExists(i, j)) {
319                                         size_t nextFor_i =
    (vgraph.getAdjacentVertex(i, 0) != j) ? vgraph.getAdjacentVertex(i, 0) :
    vgraph.getAdjacentVertex(i, 1);
320                                         size_t nextFor_j =
    (vgraph.getAdjacentVertex(j, 0) != i) ? vgraph.getAdjacentVertex(j, 0) :
    vgraph.getAdjacentVertex(j, 1);
321                                         if ((ct[i][nextFor_i][j][nextFor_j] ==
    ctInBounds) && (nextFor_i != idx) && (nextFor_j != idx) && (i != idx) && (j != idx)
    && (nextFor_i != 0) && (nextFor_j != 0)){
322                                               processBlockingChain();
323                                               return false;
324                                         }
325                                   }
326                             }
327                       }
328                 }
329                 vgraph.updateEnabledPoints(chainSize, idx, nextVisiblePoints);
330                 if (numOfAp > 0 && nonTrivialCase){
331                       nonTrivialBccCounter++;
332                 }
333                 return true;
334           }
```