Метод вирішення задачі пошуку зіркового многокутника з мінімальною площею

А.І. Красіков, студент 3 курсу, групи МІ-32

Анотація. У роботі розв'язана задача пошуку зіркового многокутника з мінімальною площею. Розроблено та реалізовано наближений алгоритм вирішення цієї задачі методом рандомізації, та точний алгоритм на основі вичерпного перебору всіх зіркових многокутників зі складністю $O(N^6)$.

Abstract. In this paper, the problem of finding a star polygon with a minimum area is solved. An approximate randomized algorithm for solving this problem, and an exact algorithm, based on an exhaustive search of all star polygons with complexity $O(N^6)$, are developed and implemented.

ЗМІСТ

ВСТУП

Поставлена задача є прикладом більшого класу задач, а саме пошуку многокутника з тими чи іншими характеристиками, що має мінімальну площу серед усіх таких многокутників. Такі задачі часто постають перед дослідниками, особливо під час розв'язання різного роду задач оптимізації.

Завдання мінімізації не рідкість у галузі обчислювальної геометрії (див., наприклад, [1–3]).

В даній роботі многокутник має ті характеристики, що

1. є зірковим;
2. його вершинами є всі точки заздалегідь заданої множини точок.

Щодо конкретної задачі, пов’язаної з зірковими многокутниками, то тут можна навести роботи на кшталт [4], де серед інших питань в розділі 2.1 розглядаються алгоритми побудови випадкових зіркових многокутників, як і в роботі [5]. В розділі 2.2 роботи [6] також розглянуті деякі евристики щодо побудови зіркових многокутників. В цих роботах наведені також деякі оцінки щодо складності поставленої задачі, такі як оцінка кількості ядер для зіркових многокутників на множині з N точок, яка дорівнює $O(N^4)$. В роботі [4] показано, що існує найкращий алгоритм побудови всіх зіркових многокутників на множині з N точок, який виконує цю задачу за час $O(N^5 \log N)$.

ОСНОВНА ЧАСТИНА

Постановка задачі

Нехай задано фіксовану множину S з N точок на площині. Необхідно розробити алгоритм генерації зіркових многокутників, вершинами яких є всі точки заданої множини S, і визначити многокутник найменшої площі.

Основні поняття та визначення

Зірковий многокутник (англ. star-shaped polygon) у даному випадку — це такий многокутник, для якого існує хоча б одна точка усередині, з якої видно всі його вершини. Іншими словами, всі відрізки, побудовані з цієї точки до всіх вершин многокутника лежатимуть усередині області, обмеженої многокутником [7]. Це визначення у 1975 році ввів М. Шеймос [8].

Для фіксованої множини точок S існує скінченна множина зірчастих многокутників, які можна побудувати на даній множині точок. У кожного варіанта є своє ненульове *ядро* (англ. polygon kernel), тобто така область усередині многокутника, з якої видно всі його вершини [7,8]. Як показано в роботах [4–6], кількість можливих зіркових многокутників складає $O(N^4)$.

Зоряний многокутник можна побудувати із будь-якої початкової точки, що знаходиться всередині *опуклої оболонки* тієї самої хмари точок (CH(S)), в тому числі з будь-якою з точок, що входить до вихідної множини S. Така точка називається *центром зіркового многокутника*; вона буде знаходитися всередині ядра цього многокутника (фактично там, де ми встановимо центр нашого зірчастого многокутника, і буде його ядро).

У деяких випадках *центр* є єдиною точкою цього ядра (згідно з термінологією, використаною у статті [5], такі многокутники (у яких ядро має нульову площу) вважаються *виродженими* (англ. degenerate)).

ПРАКТИЧНА ЧАСТИНА

Розробка алгоритму генерації зіркових многокутників, вершинами яких є всі точки заданої множини

Таким чином, рішення першої частини поставленої задачі — *розробити алгоритм генерації зіркових многокутників, вершинами яких є всі точки заданої множини S* — витікає безпосередньо з означення зіркового многокутника, та того, що його можна побудувати для будь-якої точки всередині опуклої оболонки.

Отже, за наявності точки O всередині опуклої оболонки, яку розглядаємо як точку ядра, зірковий трикутник будується за допомогою такого алгоритму.

Алгоритм побудови зіркового многокутника для заданого центру O.

1. Для кожної точки P множини S знаходимо кут, що відповідає вектору $\overrightarrow{OP}$. Якщо точка O є елементом множини S, вважаємо цей кут рівним 0.
2. Сортуємо всі точки за обчисленими кутами; у разі збігу кутів для визначеності приймаємо точку, що лежить ближче до точки O, як попередню для точки, що лежить далі.
3. З’єднуємо всі точки множини S у зазначеному порядку (остання точка з’єднується з першою), одержуючи в такий спосіб замкнений зірковий многокутник (властивість зірковості виконується, виходячи з методу побудови: всі вершини видні з точки O, тож така точка центру існує, виходячи з обраного методу побудови).

Оскільки кут обчислюється за час $\Omega(1)$, усі кути обчислюються за час $\Omega(N)$, сортування виконується за час $O(N \log N)$ — часова складність цього алгоритму становить $O(N \log N)$. Схематично алгоритм показано на рис. 1.

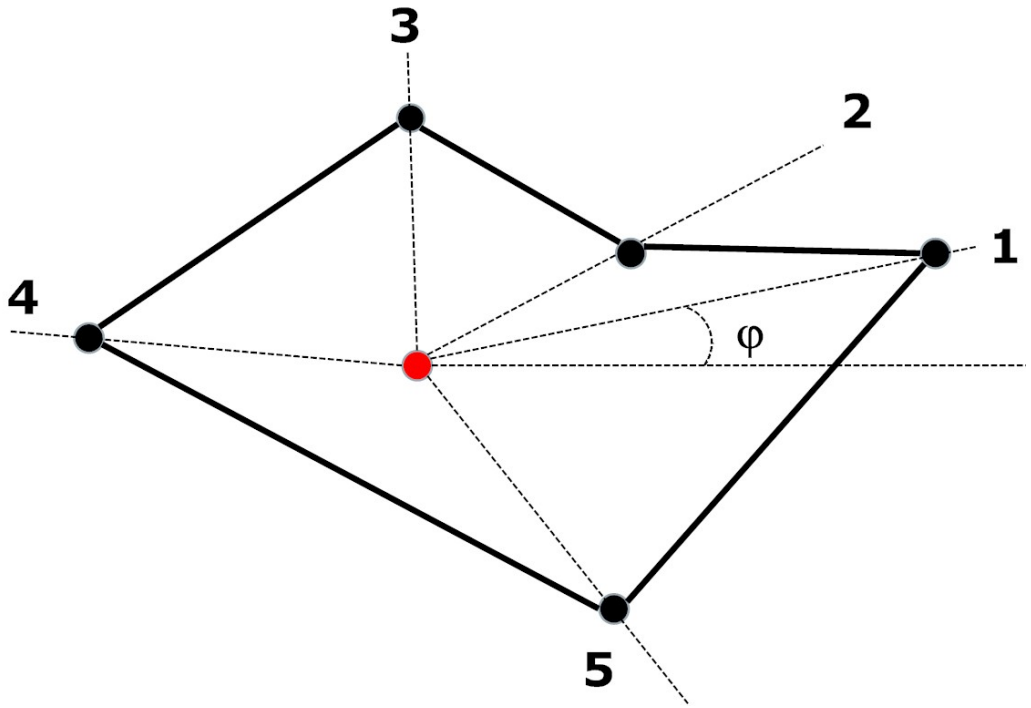

Рисунок 1. Алгоритм побудови зіркового многокутника

Центральна точка зіркового многокутника може бути будь-якою точкою, що знаходиться в опуклій оболонці множини *S*. За необхідності опуклу оболонку точок в роботі будуємо з використанням добре відомого алгоритму Грехема [9,10]. Точка центру O може бути обрана довільно; єдине обмеження — вона повинна знаходитися в межах опуклої оболонки. Оскільки ця задача вирішується ще й в алгоритмі пошуку зіркового многокутника мінімальної площі, опишемо відповідний алгоритм тут.

Алгоритм перевірки, чи знаходиться точка в межах опуклої оболонки

Вхідні дані: вектор точок опуклої оболонки такий, що за побудовою перша точка в ньому має найменшу координату *x*, та координато точки *P*.

1. Якщо координата *x* точки *P* менша за координату *x* першої точки оболонки, точка *P* лежить поза оболонкою.
2. Інакше для всіх точок оболонки крім першої обчислюємо кути, під якими вони видимі з першої точки, та відповідний кут для точки *P*.
3. Якщо кут для точки *P* більше найбільшого або менше найменшого з обчислених кутів, точка *P* лежить поза оболонкою.

4. Користуючись відсортованістю кутів верши оболонки бінарним пошуком (алгоритм `upper_bound` стандартної бібліотеки) знаходимо пару вершин, між якими проходить луч, проведений від першої вершини оболонки через точку P, та дивимось за допомогою орієнтованої площі трикутника (див. наступний підрозділ), чи знаходиться точка P до перетину з відповідним ребром, чи за ним.

Час роботи алгоритму — $\Omega(N)$ за рахунок обчислення кутів; якщо ж кути попередньо обчислені, час роботи — $\Omega(\log N)$.

Обчислення площі многокутника

Обчислення площі многокутника виконується з використанням орієнтованих площин трикутників [11,12], на які розбивається многокутник:

$$S = \frac{1}{2}\begin{vmatrix} x_1 & y_2 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

Для спрощення обчислень будуємо трикутники відносно початку координат, тобто для многокутника $P_1P_2\ldots P_n$ це трикутники OP_1P_2, OP_2P_3 и т.д., де O — точка початку координат. Завдяки такому спрощенню, коли при цьому початок координат і дві точки кінців відрізку P_1 і P_2 утворюють трикутник, його орієнтована площа дорівнює

$$S = \frac{1}{2}\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} = \frac{x_1y_2 - x_2y_1}{2}.$$

тобто формула включає координати тільки двох точок, бо координати третьої точки — (0,0). Абсолютне значення суми таких площ трикутників дає площу всього многокутника. Час обчислення становить $\Omega(N)$.

Пошук зіркового многокутника найменшої площі

Для рішення цієї частини завдання було розроблено два алгоритми — рандомізований алгоритм із застосуванням методу Монте-Карло [13], який забезпечує швидке обчислення, але не гарантує його точності, і регулярний

алгоритм вичерпного перебору всіх зіркових многокутників із визначенням многокутника найменшої площі.

Рандомізований алгоритм

Рандомізований алгоритм, на розробку якого наштовхнули роботи [4,5], ґрунтується на генерації випадкової множини з M точок усередині опуклої оболонки, які розглядаються як центри для побудови зіркових многокутників. Під час побудови чергового зіркового многокутника за описаним вище алгоритмом обчислюємо його площу, як вказано в попередньому розділі, порівнюємо із найменшим значенням площі, знайденим до цього моменту, і якщо площа нового многокутника менша, то запам’ятовуємо нове мінімальне значення площі для чергових порівнянь, а відповідний многокутник — як послідовність вершин, що його складають.

В якості першого значення площини для пошуку мінімуму беремо площину опуклої оболонки, перевищити яку не може жоден зірковий многокутник. Це витікає з того факту, ще весь він повністю покривається опуклою оболонкою (за її визначенням).

При реалізації алгоритму одноразово будується опукла оболонка множини точок для подальшої перевірки входження в неї випадковим чином генерованих точок. Генерація випадкових точок виконується за час $\Omega(1)$, перевірка приналежності точки до області всередині оболонки — за час $O(N)$ (виходячи з максимально можливої кількості точок оболонки), генерація одного зіркового многокутника для цієї точки — за час $O(N \lg N)$. Таким чином, час роботи всього алгоритму становить $O(MN \lg N)$.

На жаль, рандомізований алгоритм не може гарантувати знаходження многокутника найменшої площі, хоча й дає, як буде показано далі, досить непогані результати.

Алгоритм вичерпного перебору всіх зіркових многокутників

Цей алгоритм будує всі можливі зіркові многокутники, обчислює і порівнює їхні площі в пошуках многокутника з найменшою площею.

Проблема побудови всіх можливих ядер для заданої множини точок розглядається, наприклад, в [4], але він достатньо складний. До того ж нас цікавить в першу чергу побудова не ядер, а саме зіркових многокутників для пошуку їх площини. Звісно, ці задачі дуальні і тісно пов’язані одна з одною, але в даній роботі вирішено ґрунтуватися на многокутниках та будувати ядра, виходячи з них.

Розроблений алгоритм ґрунтується на тому, що для кожного зіркового многокутника можна побудувати його ядро — многокутник, що являє собою перетин усіх напівплощин, які визначаються ребрами зіркового многокутника. Продовження ребер та ядро многокутника, показаного на рис. 1, наведено на рис. 2.

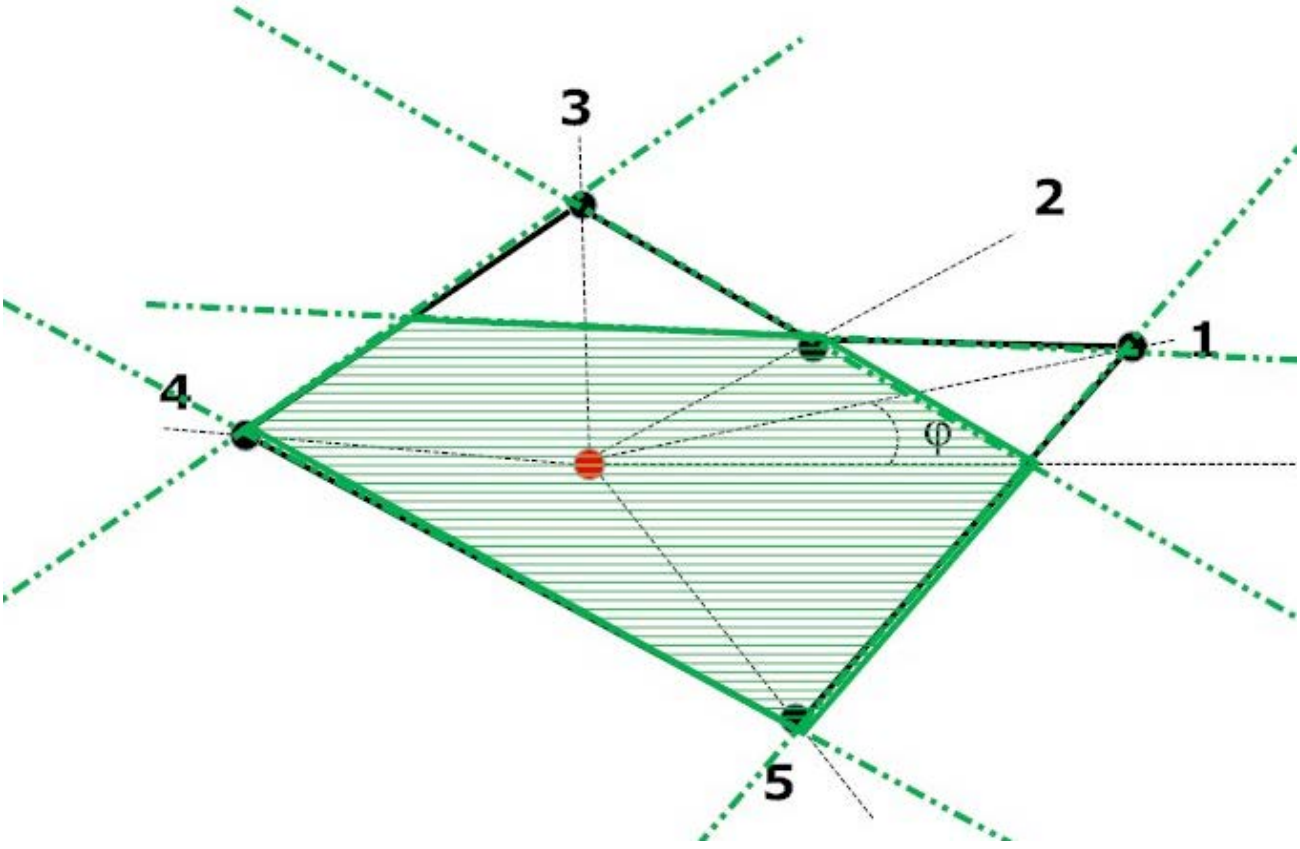

Рисунок 2. Побудова ядра зіркового многокутника, наведеного на рис. 1

При цьому під час переходу в нове, сусіднє ядро через деяке ребро вихідного ядра зірковий многокутник, відповідний новому ядру, відрізняється від зіркового многокутника вихідного ядра тим, що змінюється напрямок його ребра, що відповідає ребру ядра, яке перетинається.

Так, на рис. 2 при переході з ядра, заштрихованого зеленим, до сусіднього, незаштрихованого ядра в формі трикутника зверху, через продовження сторони 1-2, порядок вершин 1 та 2 зміниться, і зірковий многокутник для незаштрихованого ядра матиме вигляд 1-3-4-5-2-1.

Таким чином, ми знаємо зірковий многокутник для сусіднього ядра, а знаючи його, ми можемо побудувати і саме ядро.

Далі, розглядаючи ядра як вершини деякого графа, а суміжні ядра — як з’єднані ребром графа, ми можемо виконати обхід усіх вершин графа (тобто ядер, а отже, і зіркових многокутників) за допомогою звичайного алгоритму обходу в ширину [14], і знайти многокутник мінімальної площі.

При цьому побудова ядра виконується за час $O(N^2)$ — тому що N ребер зіркового многокутника перевіряються на перетин з $O(N)$ ребрами побудованого на цей час ядра.

Ми беремо в якості першого наближення ядра прямокутник, що визначається найменшими та найбільшими координатами множини точок, а потім на кожній ітерації алгоритму розглядаємо перетин ребра зіркового многокутника з ребрами ядра, відкидаючи ті його точки, які не входять до питомої напівплощини цього ребра, та додаючи в ядро точки перетину, якщо такі є.

Усього кількість таких ядер обмежена значенням $O(N^4)$ [4–6] — тому що їх не більше, ніж частин, на які ділять область усі можливі прямі (у найгіршому випадку всі вони перетинаються між собою). Кількість таких частин квадратично залежить від кількості прямих, а їх кількість, у свою чергу, квадратично залежить від кількості точок. Таким чином, розроблений алгоритм має час роботи $O(N^6)$ (тому що в цьому випадку ми не будуємо кожен зірковий многокутник заново). Пошук площі многокутника для даного ядра займає час $O(N)$, так що сумарний час пошуку площ $O(N^5)$ поглинається часом роботи частини побудови алгоритму.

Отже, розроблений алгоритм має час роботи $O(N^6)$.

Програмна реалізація розроблених алгоритмів

Програмна реалізація на C++ (Visual C++ 2022) використовує для візуалізації вільно розповсюджувану бібліотеку BGI-графіки для Windows [15]. Її можливостей цілком достатньо для візуалізації поставленого завдання, а наявність одночасно вікна графічного виведення (я використовував вікно розміром 900×900) і текстової консолі забезпечує зручний спосіб відладки.

Основна функціональність, пов'язана з побудовою зіркових многокутників, ядер тощо, винесена в окремий простір імен `poly` у файл `point.h`.

Сюди входять такі функції та класи.

Функція `atan3()` — аналог `atan2()`, але з діапазоном від 0 до 2π, використовується при побудові зіркового многокутника.

Структура `Point`, яка являє собою точку з координатами `x`, `y`, а також (для зручності роботи) номером, який точка отримує за допомогою статичного члена класу `N`, що відстежує кількість створених точок. Відкритий інтерфейс класу включає перевірку точок на рівність, упорядкування, орієнтовану площу трикутника з вершинами в точках `a` і `b` і на початку координат (для швидкого обчислення площі многокутників) і для перевірки кута повороту від двох точок до третьої (використовується під час побудови опуклої оболонки).

```
struct Point
{
    Point(double x = 0, double y = 0):x(x),y(y),n(N++){}

    double x = 0, y = 0;
    size_t n = 0;
    inline static int N = 0;

    static inline bool less(const Point& a, const Point& b);
    static inline bool cw(const Point& a, const Point& b, const Point& c);
    static inline bool ccw(const Point& a, const Point& b, const Point& c);
    static inline double S0(const Point& a, const Point& b);
};
inline bool operator == (const Point& a, const Point& b);
```

Функції `convexHull` і `convexHullFull` будують опуклі оболонки множини точок за алгоритмом Грехема [9], виключаючи проміжні точки на одному відрізку оболонки і включаючи їх відповідно:

```
inline std::vector<Point> convexHull(const std::vector<Point>& a);
inline std::vector<Point> convexHullFull(const std::vector<Point>& a);
```

Функція

```
inline double Area(const std::vector<poly::Point>& a)
```

обчислює площу многокутника `a` з використанням орієнтованої площі трикутників.

Побудову зіркового многокутника з вершинами в точках `a` навколо точки `pt` шляхом сортування всіх вершин за кутом і відстанню від точки `pt` виконує функція

```
inline std::вектор<poly::Point> starPoly(const poly::Point& pt,
                                         const std::vector<poly::Point>& a)
```

Функція

```
inline bool inHull(const poly::Point& p, const std::vector<poly::Point>& h)
```

перевіряє знаходження точки `p` усередині опуклої оболонки `h`, перевіряючи, де перебуває точка `p` на промені, що проходить через неї і виходить із точки `h[0]` — до або після перетину з відрізком оболонки (використовується в методі Монте-Карло).

Функції

```
std::vector<poly::Point> BoundRect(const std::vector<poly::Point>& poly);
double leftOf(const poly::Point& p, const poly::Point& begin,
                                    const poly::Point& end);
poly::Point cross(double Lp, const poly::Point& p,
                  double Lq, const poly::Point& q);
```

є допоміжними функціями під час побудови ядра зіркового многокутника. `BoundRect` будує початкове наближення ядра у вигляді прямокутника, `leftOf` обчислює, чи перебуває точка `p` ліворуч або праворуч від прямої, яка проходить через точки `begin` і `end` (обчислюючи орієнтоване значення відстані до прямої, більше за 0, якщо точка лежить ліворуч. Функція `cross` знаходить точку перетину відрізка `pq` з прямою, що проходить через точки `begin` і `end`, за значеннями координат самих точок і значень, обчислених функцією `leftOf`.

Точки вершин ядра описуються структурою

```
struct KernelVertex
{
    poly::Point p;
    size_t b,e;
};
```

у якій крім самої точки записуються номери точок ребра зіркового многокутника, яке призвело до утворення цієї вершини ядра.

Саме ядро являє собою структуру

```
struct StarKernel // Ядро зіркового полігона
{
    std::vector<poly::Point> s; // Полігон
    std::vector<poly::KernelVertex> k; // Ядро для s
    std::vector<std::pair<size_t,size_t>> ixs;
    std::size_t hashvalue;
```

яка містить зірковий многокутник `s` з парами номерів вершин для його ребер `ixs` (воно і значення хеша `hashvalue` забезпечує можливість зберігання ядра в контейнері `unordered_set` мови C++). Конструктор

```
StarKernel(const StarKernel& kernel, size_t edge)
```

дозволяє перейти до сусіднього ядра для зіркового многокутника, отриманого шляхом розвороту ребра, записаного у вершині `edge` ядра.

Функція

```
std::vector<poly::KernelVertex> buildKernel(
            const std::vector<poly::Point>& starPoly)
```

будує ядро для зіркового многокутника `starPoly`, проходячи всіма ребрами многокутника з відсіканням відповідних областей ядра допоміжною функцією

```
std::vector<poly::KernelVertex> clipBound(
        const std::vector<poly::KernelVertex>& bound,
        const std::vector<poly::Point>& star,
        size_t b, size_t e)
```

яка для ядра `bound` і многокутника `star` виконує відсікання ребром многокутника, яке визначається точками `b` і `e`.

Сама програма дуже проста. Вона заповнює вектор попередньо заданими або випадковими точками (їх кількість задається в командному рядку; за замовчуванням — 25), обчислює для них опуклу оболонку та її площу і розміщує випадковим чином 100000 точок, будуючи для них зіркові многокутники та запам’ятовуючи ті, що мають найбільшу та найменшу площу.

Попередньо задані значення можна вводити через файл, ім’я якого передається в командному рядку, та який в кожному рядку містить пару координат x та y однієї точки.

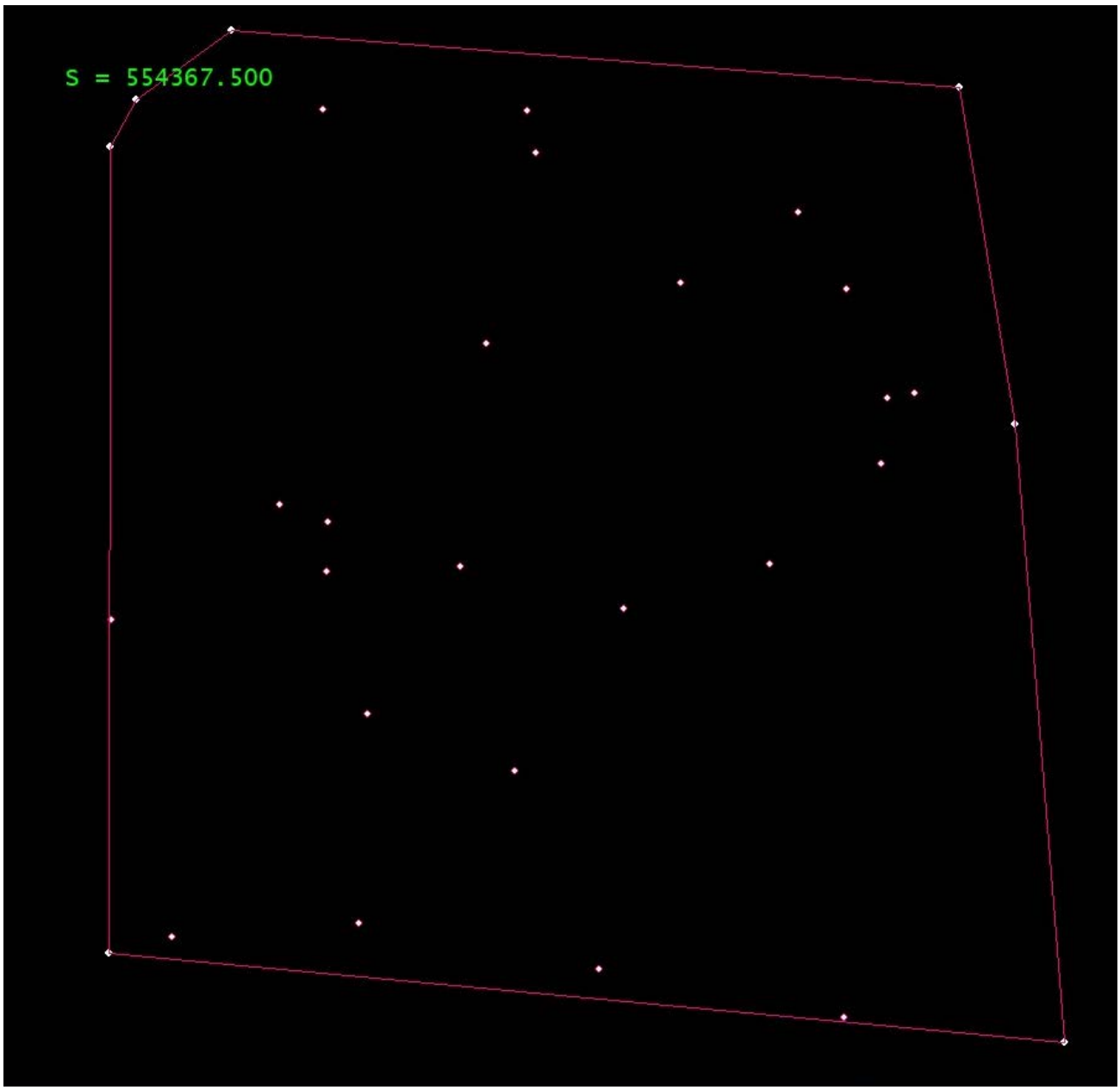

Рисунок 3. Приклад множини точок *S*, її опукла оболонка та площа

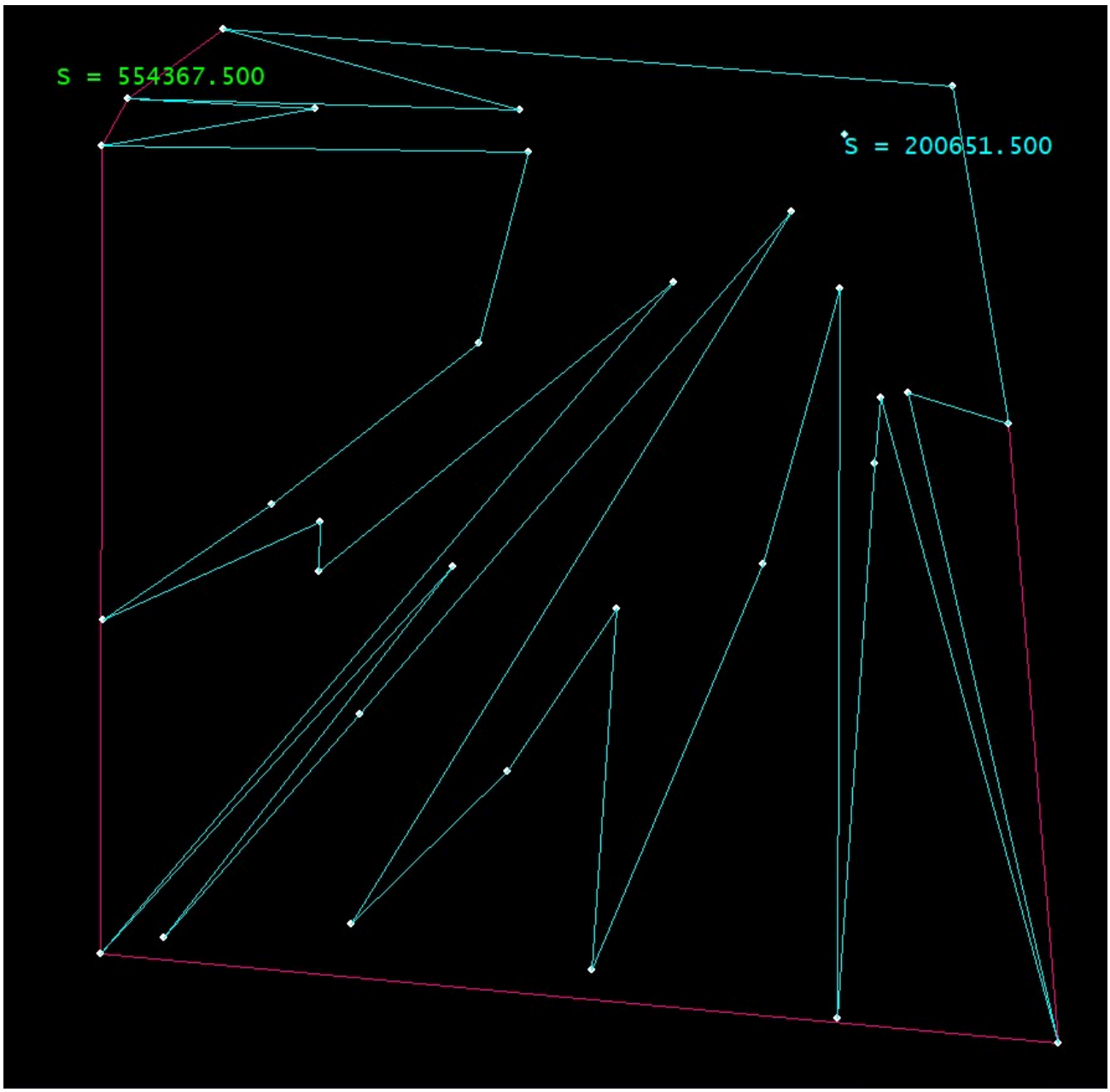

Рисунок 4. Зірковий многокутник мінімальної площі, отриманий рандомізованим алгоритмом

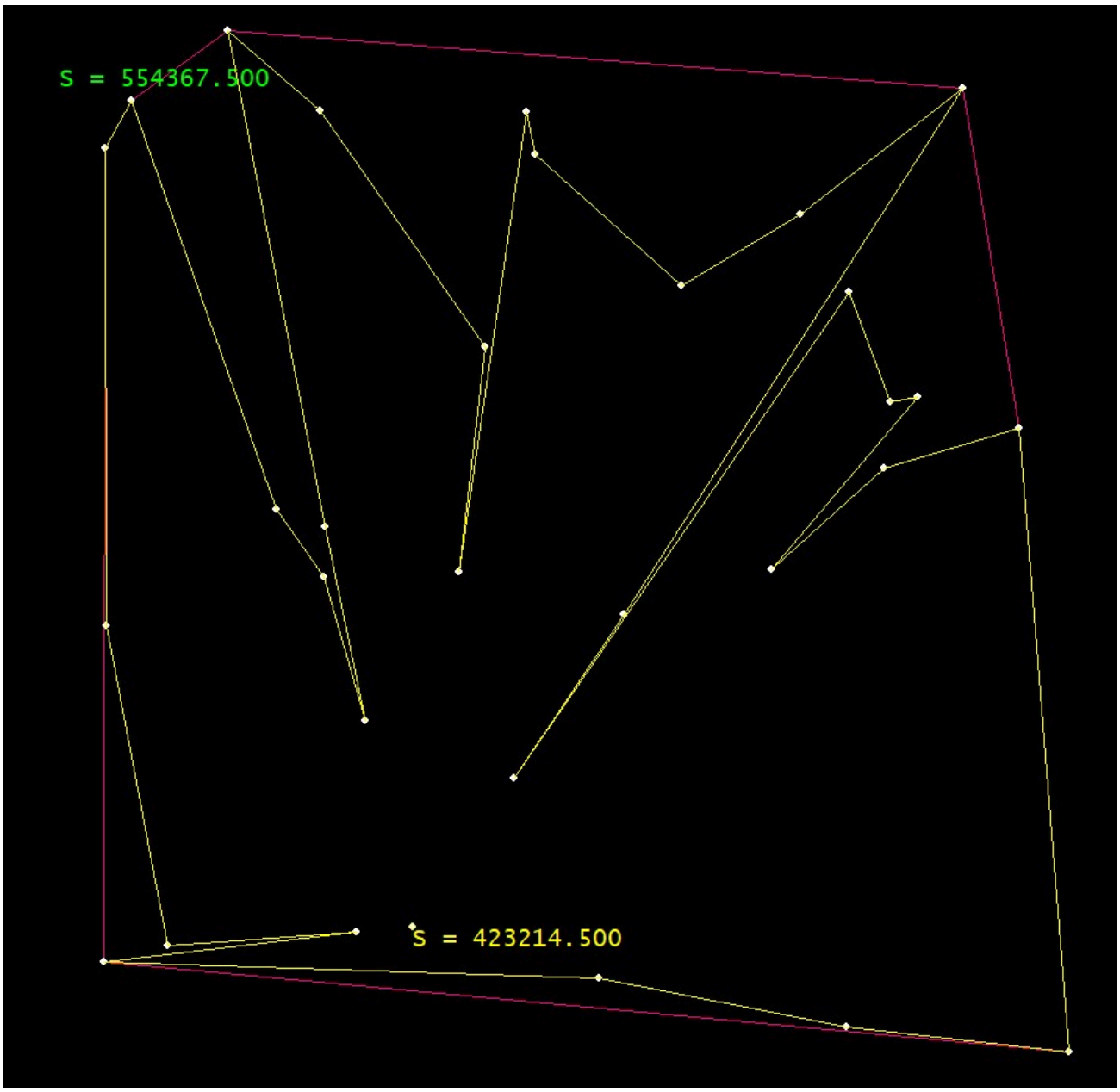

Рисунок 5. Зірковий многокутник максимальної площі, отриманий рандомізованим алгоритмом

Після цього програма обчислює «центр мас» точок, і використовує його для побудови зіркового многокутника і першого ядра. Потім, використовуючи метод пошуку в ширину, виконується побудова та обхід усіх ядер, з побудовою для кожного ядра відповідного зіркового многокутника та розрахунку його площі. На наведеному далі малюнку показано ядра, які мають різні кольори: чим площа многокутника для цього ядра менша, тим ядро темніше, і навпаки:

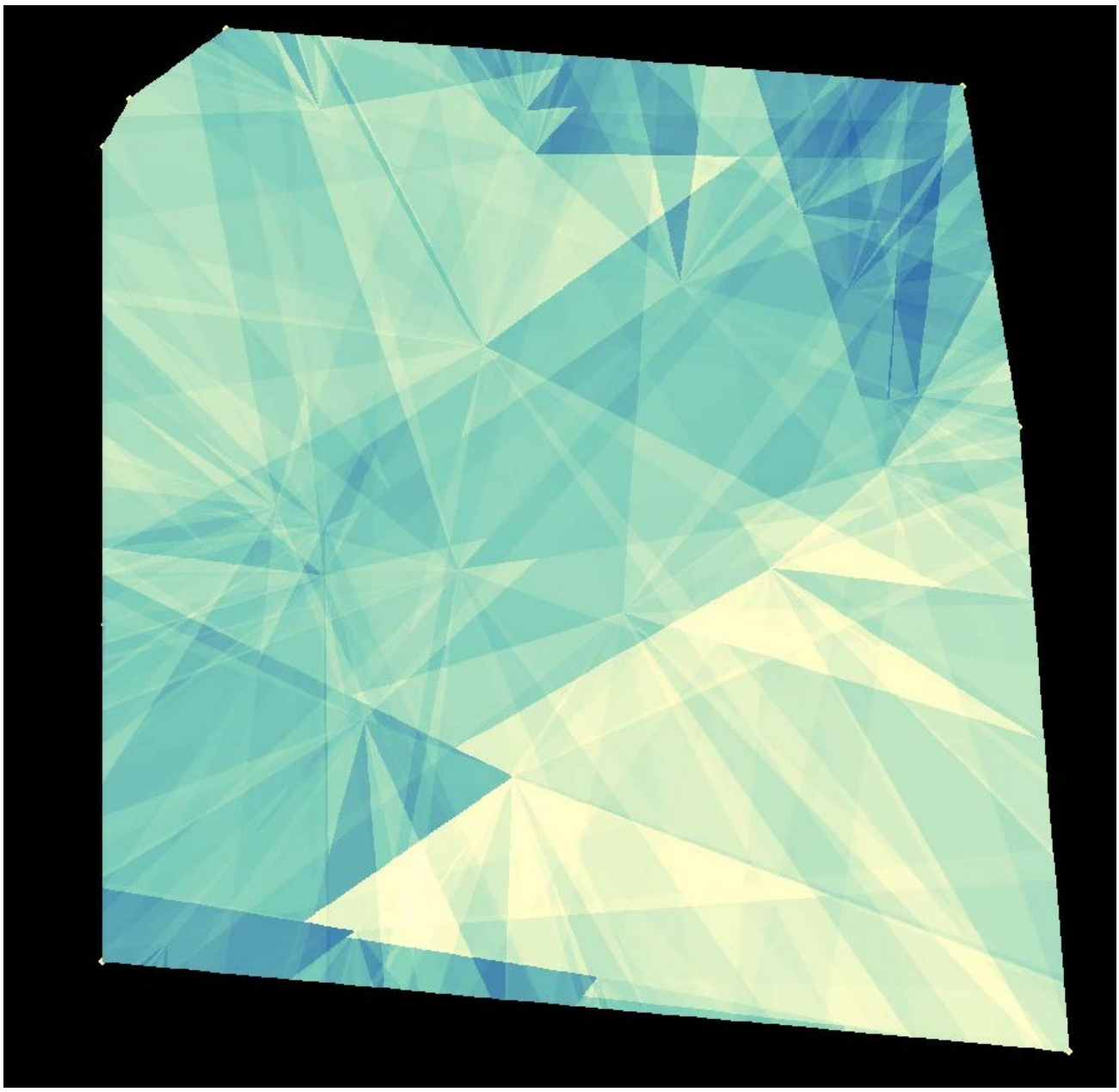

Рисунок 6. Побудова множини ядер всіх зіркових многокутників для даної множини точок

Потім просто виводяться зіркові многокутники максимальної та мінімальної площі:

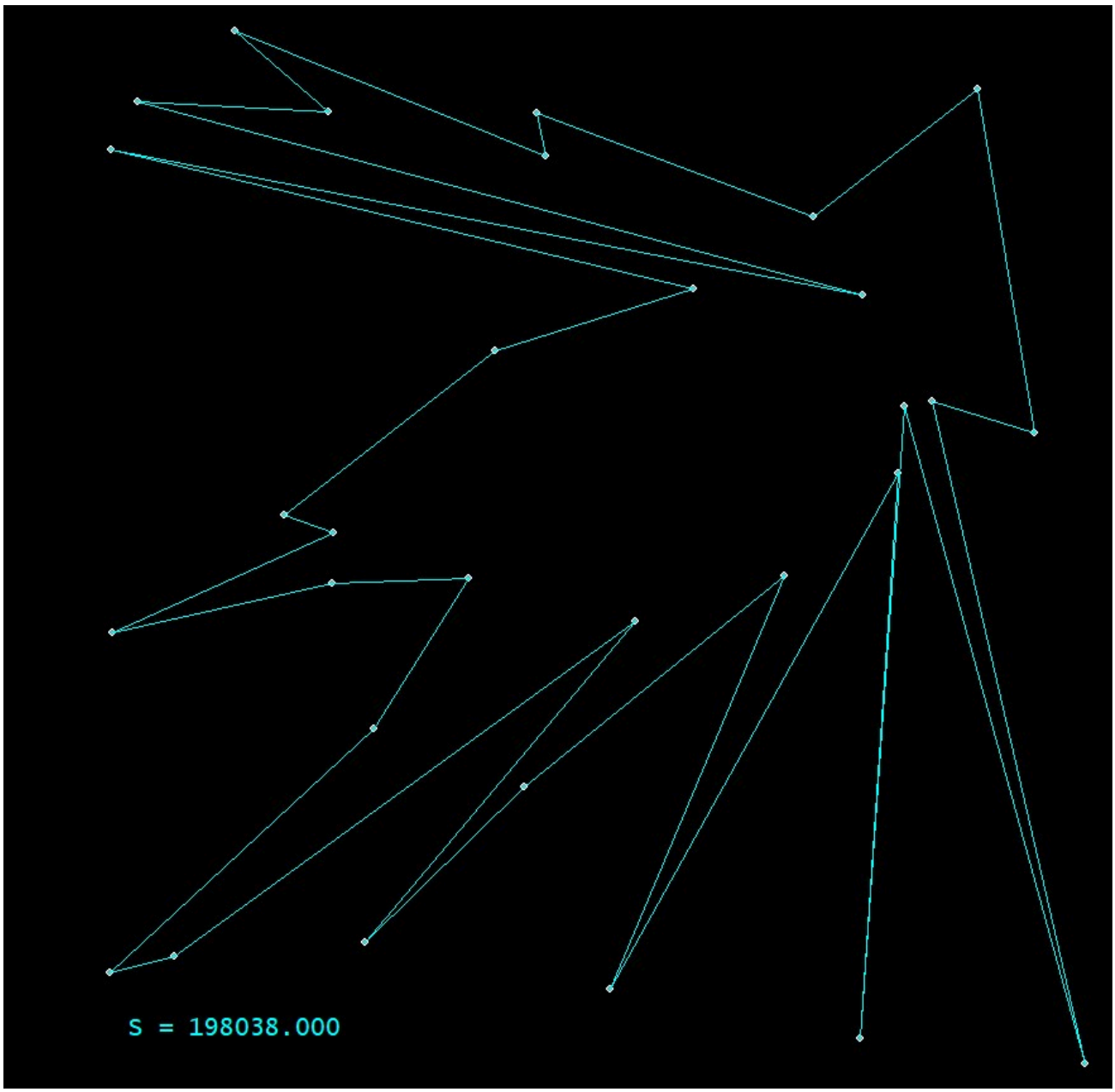

Рисунок 7. Зірковий многокутник мінімальної площі, отриманий алгоритмом вичерпного перебору

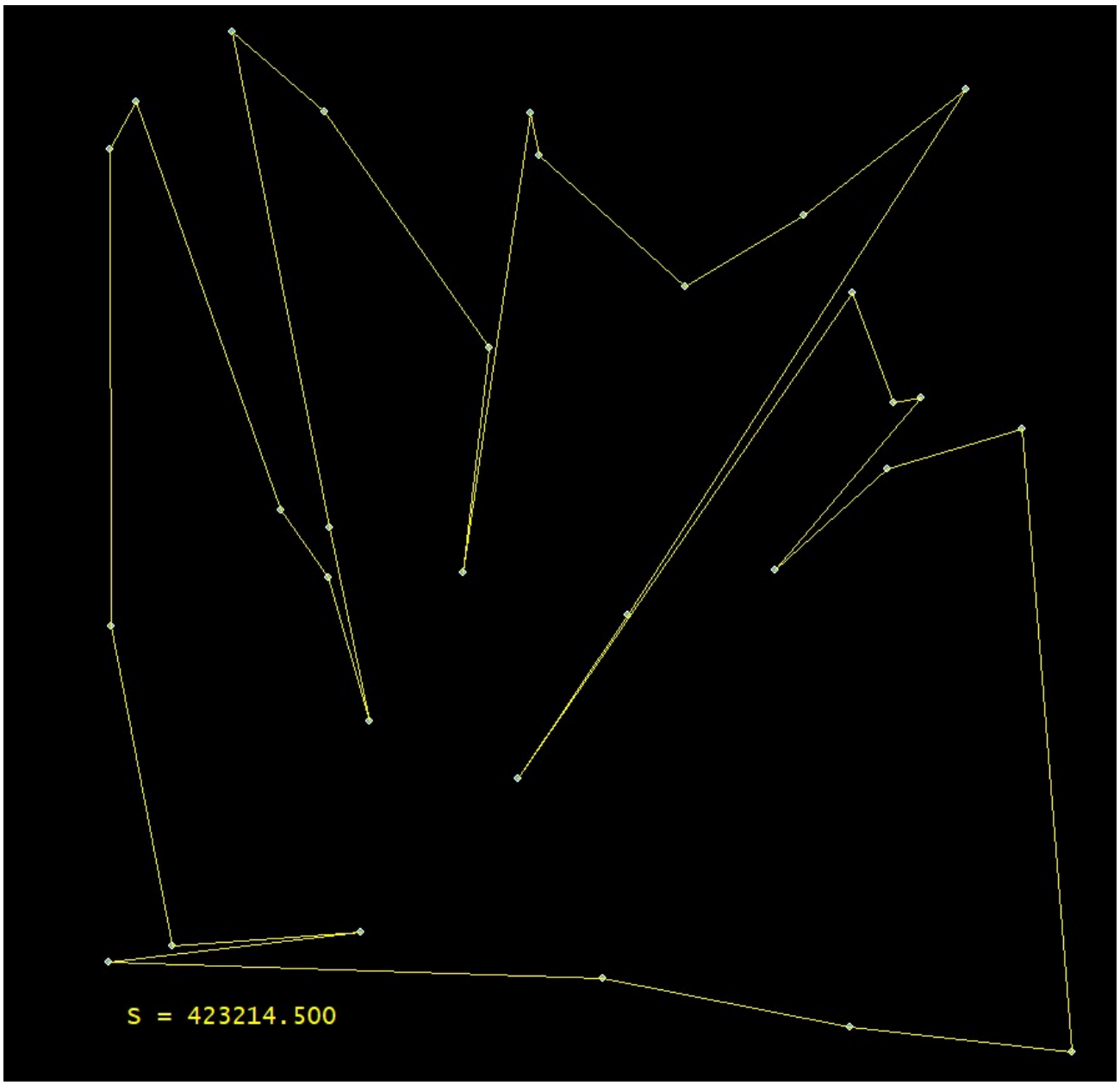

Рисунок 8. Зірковий многокутник максимальної площі, отриманий алгоритмом вичерпного перебору

Як бачимо, рандомізований метод, істотно перевершуючи метод вичерпного перебору ядер за швидкістю (для наведеної множини з 30 точок приблизно в 5 разів швидше), дає схожі результати: для многокутника максимальної площі 423214.5 в обох випадках, і 200651.5 проти 198038 для мінімальної площі (різниця близько 1%).

Результати обчислень варіюються. Так, для випадкового набору з 40 точок метод Монте-Карло працював приблизно у 25 разів швидше вичерпного перебору,

видавши однакову мінімальну площу, а в разі максимальної помилився приблизно на 2%.

ВИСНОВКИ

Таким чином, в роботі розроблено та реалізовано

1. Алгоритм побудови зіркового многокутника для заданої множини точок.
2. Швидкий наближений рандомізований алгоритм пошуку зіркового многокутника для заданої множини точок мінімальної (та максимальної) площини.
3. Точний алгоритм пошуку зіркового многокутника для заданої множини точок мінімальної (та максимальної) площини методом вичерпного перебору ядер зіркових многокутників.

Показано, що рандомізований алгоритм, суттєво більш швидкий за метод вичерпного перебору ядер, дає досить близькі до точних результати.

Вибір конкретного алгоритму залежить від конкретних вимог до завдання.

СПИСОК ЛІТЕРАТУРИ

1. Терещенко В.М. *Аналіз методів розв'язання оптимізаційних задач обчислювальної геометрії. Навчальний посібник*. — К.: ВПЦ "Київський університет", 2022. — 112 с.
2. E.M.Arkin, Y.-J.Chiang, M.Held, J.S.B.Mitchell, V.Sacristan, S.S.Skiena, T.-C.Yang. *On Minimum-Area Hulls*. Algorithmica, 1998, 21, 119–136. DOI: 10.1007/PL00009204. URL: https://sci-hub.ru/10.1007/PL00009204.
3. László Kozma. *Minimum Average Distance Triangulations*. European Symposium on Algorithms 10th September 2012. PDF: https://arxiv.org/pdf/1112.1828.pdf
4. Thomas Auer, Martin Held. *Heuristics for the generations of Random Polygons*. In: Frank Fiala, Evangelos Kranakis, Jörg-Rüdiger Sack: Proceedings of the 8th Canadian Conference on Computational Geometry, Carleton University, Ottawa, Canada, August 12-15, 1996. Carleton University

Press, 1996, p.38-44. ISBN 0-88629-307-3, DOI: 10.1515/9780773591134-009.

5. Christian Sohler. *Generating random star-shaped polygons*. In: Proc. 11th Canadian Conference on Computational Geometry (CCCG99), 174-177, 1999. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=3fb70460a8e5fd7c8181fbcc0ef3d869eb3263ae.
6. Pratik Shankar Hada. *Approaches for Generating 2D Shapes*. UNLV Theses, Dissertations, Professional Papers, and Capstones. 2182. – 2014. DOI: 10.34917/6456412. URL: https://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=3183&context=thesesdissertations.
7. Wikipedia: *Star-shaped polygon*. URL: https://en.m.wikipedia.org/wiki/Star-shaped_polygon.
8. Michael Ian Shamos. *Geometric complexity, Proceedings of the Seventh A C M Annual Symposium on Theory of Computing*, 224-233, 1975. URL: https://dl.acm.org/doi/pdf/10.1145/800116.803772.
9. Кормен, Томас Х. и др. *Алгоритмы. Построение и анализ*, 3-е изд. — М., ООО «И.Д. Вильямс», 2013. — 1328 с.
10. Препарата Ф., Шеймос М. *Вычислительная геометрия. Введение*. — М.:Мир, 1989. — 478 с.
11. Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. Computational Geometry (2nd revised ed.), Springer-Verlag, 2000. ISBN 3-540-65620-0
12. Анісімов А.В., Терещенко В.М., Кравченко І.В. *Основні алгоритми обчислювальної геометрії*. — К.: ВПЦ "Київський університет", 2002. — 81 с. URL: http://cg.unicyb.kiev.ua/
13. Карташов М. В. *Імовірність, процеси, статистика*. — К.: ВПЦ "Київський університет", 2007. — 504 с.

14. Amy E. Hodler and Mark Needham. Graph Algorithms. O’Reilly Media, Inc., 2019. — 217 p.
15. BGI library implementation for Microsoft(R) Windows(TM). Copyright (C) 2006 Daniil Guitelson. URL: https://github.com/daniilguit/openbgi

ДОДАТОК. ВИХІДНІ ТЕКСТИ ПРОГРАМИ

Вихідний файл point.h

```
#pragma once
#include <vector>
#include <algorithm>
#include <cmath>
#include <tuple>
#include <bit>
namespace poly
{
    double atan3(double y, double x)
    {
        double f = atan2(y,x);
        if (f >= 0) return f;
        return 2*3.141592653589793 + f;
    }
    struct Point
    {
        Point(double x = 0, double y = 0):x(x),y(y),n(N++){}
        double x = 0, y = 0;
        size_t n = 0;
        inline static int N = 0;
        static inline bool less(const Point& a, const Point& b)
        {
            return a.x < b.x || a.x == b.x && a.y < b.y;
        }
        static inline bool cw(const Point& a, const Point& b, const Point& c)
        {
            return a.x*(b.y - c.y) + b.x*(c.y - a.y) + c.x*(a.y - b.y) < 0;
        }
        static inline bool ccw(const Point& a, const Point& b, const Point& c)
        {
            return a.x*(b.y - c.y) + b.x*(c.y - a.y) + c.x*(a.y - b.y) > 0;
        }
        static inline double S0(const Point& a, const Point& b)
        {
            return (a.x*b.y - a.y*b.x)/2.;
        }
    };
    inline bool operator == (const Point& a, const Point& b)
    {
        return a.x == b.x && a.y == b.y;
    }
    inline std::vector<Point> convexHull(const std::vector<Point>& a)
    {
        std::vector<Point> r;
        if (a.size() <= 1) return r;
```

```
    r = a;
    // Sort all points left to right
    sort(r.begin(),r.end(), Point::less);
    Point p1 = r.front(), p2 = r.back();
    std::vector<Point> u{p1} /* Up side */, d{p1} /* Down side */;
    for(size_t i = 1; i < r.size(); ++i)
    {
        if (i == r.size() - 1 || Point::cw(p1, r[i], p2))
        {
            while(u.size()>=2 &&
                !Point::cw(u[u.size()-2],u[u.size()-1],r[i])) u.pop_back();
            u.push_back(r[i]);
        }
        if (i == r.size() - 1 || Point::ccw(p1, r[i], p2))
        {
            while(d.size()>=2 &&
                !Point::ccw(d[d.size()-2],d[d.size()-1],r[i])) d.pop_back();
            d.push_back(r[i]);
        }
    }
    r.clear();
    for(size_t i = 0; i < u.size(); ++i)   r.push_back(u[i]);
    for(size_t i = d.size()-2; i > 0; --i) r.push_back(d[i]);
    return r;
}
// All points on the hull
inline std::vector<Point> convexHullFull(const std::vector<Point>& a)
{
    std::vector<Point> r;
    if (a.size() <= 1) return r;
    r = a;
    // Sort all points left to right
    sort(r.begin(),r.end(), Point::less);
    Point p1 = r.front(), p2 = r.back();
    std::vector<Point> u{p1} /* Up side */, d{p1} /* Down side */;
    for(size_t i = 1; i < r.size(); ++i)
    {
        if (i == r.size() - 1 || !Point::ccw(p1, r[i], p2))
        {
            while(u.size()>=2 &&
                Point::ccw(u[u.size()-2],u[u.size()-1],r[i])) u.pop_back();
            u.push_back(r[i]);
        }
        if (i == r.size() - 1 || !Point::cw(p1, r[i], p2))
        {
            while(d.size()>=2 &&
                Point::cw(d[d.size()-2],d[d.size()-1],r[i])) d.pop_back();
            d.push_back(r[i]);
        }
    }
    r.clear();
    for(size_t i = 0; i < u.size(); ++i)   r.push_back(u[i]);
    for(size_t i = d.size()-2; i > 0; --i) r.push_back(d[i]);
    return r;
}
inline double Area(const std::vector<poly::Point>& a)
{
    double S = 0;
    for(size_t i = 0; i < a.size(); ++i)
    {
        poly::Point
            p = i ? a[i-1] : a[a.size()-1],
            q = a[i];
        S += (p.x-q.x)*(p.y+q.y);
```

```
    }
    return abs(S/2);
}
// Build star poligone
inline std::vector<poly::Point> starPoly(const poly::Point& pt,
                                         const std::vector<poly::Point>& a)
{
    poly::Point p = pt;
    std::vector<poly::Point> r;
    std::vector<std::tuple<size_t,double,double>> angls;
    // Is this point hull point?
    bool onHull = false;
    auto ch = convexHullFull(a);
    size_t id = 0;
    for(; id < ch.size(); ++id) if (pt == ch[id]) { onHull = true; break; }
    if (onHull)
    {
        // Small shift :)
        id = (id + ch.size()/2)%ch.size();
        p.x += (a[id].x - p.x)*0.001;
        p.y += (a[id].y - p.x)*0.001;
        onHull = false;
    }
    for(size_t i = 0; i < a.size(); ++i)
    {
        angls.emplace_back(i,
                (a[i].y == p.y && a[i].x == p.x) ? 0 :
                poly::atan3(a[i].y-p.y,a[i].x-p.x),
                (a[i].x-p.x)*(a[i].x-p.x)+(a[i].y-p.y)*(a[i].y-p.y));
    }
    sort(angls.begin(),angls.end(),
         [](const std::tuple<size_t,double,double>& a,
            const std::tuple<size_t,double,double>& b)
         {
             return
                 std::get<1>(a) < std::get<1>(b) ||
                 std::get<1>(a) == std::get<1>(b) &&
                 std::get<2>(a) < std::get<2>(b);
         });
    for(size_t i = 0; i < angls.size(); ++i)
        r.emplace_back(a[std::get<0>(angls[i])]);
    return r;
}
// Is point in hull
// Is line from h[0] crossing some segment?
inline bool inHull(const poly::Point& p, const std::vector<poly::Point>& h)
{
    if (p.x < h[0].x) return false;
    std::vector<double> angls;
    for(int i = 1; i < h.size(); ++i)
        angls.push_back(atan2(h[i].y-h[0].y,h[i].x-h[0].x));
    double f = atan2(p.y-h[0].y,p.x-h[0].x);
    if (f > angls[0] || f < angls[angls.size()-1]) return false;
    auto q = std::upper_bound(angls.begin(), angls.end(), f,
                              std::greater<>());
    if (q == angls.begin() || q == angls.end()) return false;
    int b = int(q - angls.begin()) + 1, c = b - 1;
    double s = p.x*(h[b].y - h[c].y) +
               h[b].x*(h[c].y - p.y) + h[c].x*(p.y - h[b].y);
    return s > 0;
}
inline std::vector<poly::Point>
        BoundRect(const std::vector<poly::Point>& poly)
{
```

```
    auto min_x = poly[0].x;
    auto max_x = poly[0].x;
    auto min_y = poly[0].y;
    auto max_y = poly[0].y;
    for(size_t i = 1; i < poly.size(); ++i)
    {
        min_x = std::min(min_x, poly[i].x);
        max_x = std::max(max_x, poly[i].x);
        min_y = std::min(min_y, poly[i].y);
        max_y = std::max(max_y, poly[i].y);
    }
    return std::vector<poly::Point>{
        {min_x, min_y},
        {min_x, max_y},
        {max_x, max_y},
        {max_x, min_y}
    };
}
// Vector product
inline double leftOf(const poly::Point& p, const poly::Point& begin,
                     const poly::Point& end)
{
    return (end.x-begin.x)*(p.y-begin.y)-(p.x-begin.x)*(end.y-begin.y);
}
inline poly::Point cross(double Lp, const poly::Point& p,
                         double Lq, const poly::Point& q)
{
    return { (Lp*q.x - Lq*p.x)/(Lp-Lq),(Lp*q.y - Lq*p.y)/(Lp-Lq) };
}
struct KernelVertex
{
    poly::Point p;
    size_t b,e; // begin-end
};
inline std::vector<poly::KernelVertex>
        buildKernel(const std::vector<poly::Point>& starPoly);
struct StarKernel // Kernel of star polygone
{
    std::vector<poly::Point>  s;        // Polygone
    std::vector<poly::KernelVertex> k;  // Kernel for s
    std::vector<std::pair<size_t,size_t>> ixs;
    std::size_t hashvalue;
    StarKernel(const std::vector<poly::Point>& star)
            :s(star),k(buildKernel(star))
    {
        ixs = StarKernelVal();
        hashvalue = StarKernelHash();
    }
    StarKernel(const StarKernel& kernel, size_t edge)
        // Unfolding of the star edge
        :s(kernel.s)
        // recorded at the vertex of the kernel
    {
        std::swap(s[kernel.k[edge].b],s[kernel.k[edge].e]);
        k = buildKernel(s);
        ixs = StarKernelVal();
        hashvalue = StarKernelHash();
    }
    std::vector<std::pair<size_t,size_t>> StarKernelVal()
    {
        ixs.clear();
        for(size_t e = 0; e < k.size(); ++e)
        {
            poly::KernelVertex v = k[e];
```

```
            ixs.emplace_back(s[v.b].n,s[v.e].n);
        }
        auto it = std::min_element(ixs.begin(),ixs.end());
        std::rotate(ixs.begin(),it,ixs.end());
        return ixs;
    }
    size_t StarKernelHash()
    {
        size_t z = 0;
        for(int i = 0; i < ixs.size(); ++i)
        {
            // There are many various hashes...
            z ^= std::rotr(ixs[i].first,2*i);
            z ^= std::rotr(ixs[i].second,2*i+1);
            // z ^= ixs[i].first  + 0x9e3779b9 + (z<<6) + (z>>2);
            // z ^= ixs[i].second + 0x9e3779b9 + (z<<6) + (z>>2);
        }
        return z;
    }
};
struct StarKernelHash
{
    size_t operator()(const StarKernel& k) const { return k.hashvalue; }
};
struct StarKernelComp
{
    size_t operator()(const StarKernel& k, const StarKernel& m) const
    {
        return k.ixs == m.ixs;
    }
};
inline std::vector<poly::KernelVertex>
        clipBound(const std::vector<poly::KernelVertex>& bound,
                  const std::vector<poly::Point>& star,
                  size_t b, size_t e)
{
    std::vector<poly::KernelVertex> result;
    // Go through all the cutoffs counter clockwise
    // p, q - ponts of star poligone
    for(size_t j = bound.size()-1, i = 0; i < bound.size(); j = i++)
    {
        double Lp = leftOf(bound[j].p,star[b],star[e]);
        double Lq = leftOf(bound[i].p,star[b],star[e]);
        // TODO Optimization: Lp = Lq for the next step
        // If the points are on different sides, add the intersection point
        if (Lp < 0 && Lq > 0)
        {
            result.push_back({cross(Lp,bound[j].p,Lq,bound[i].p), b,e});
        }
        if (Lp > 0 && Lq < 0)
        {
            result.push_back({cross(Lp,bound[j].p,Lq,bound[i].p),
                              bound[i].b, bound[i].e});
        }
        if (Lq > 0) result.push_back(bound[i]);
        if (Lq == 0)
            if (Lp > 0)
                result.push_back(bound[i]);
            else if (Lp < 0)
                result.push_back({bound[i].p,b,e});
            else
                if (bound[i].b == (size_t)-1)
                    result.push_back({bound[i].p,b,e});
                else
```

```
                        result.push_back(bound[i]);
        }
        return result;
    }
    inline std::vector<poly::KernelVertex>
            buildKernel(const std::vector<poly::Point>& starPoly)
    {
        std::vector<poly::Point> rect = BoundRect(starPoly);
        std::vector<poly::KernelVertex> kernel;
        for (const auto& p : rect)
            kernel.push_back({p, (size_t)-1, (size_t)-1});
        for (size_t j = starPoly.size() - 1, i = 0; i < starPoly.size(); j = i++)
        {
            kernel = clipBound(kernel, starPoly, j, i);  // Points numbers!
        }
        return kernel;
    }
}
```

Вихідний файл polys.cpp

```
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <random>
#include <cassert>
#include <unordered_set>
#include <queue>
#include <fstream>
#include <bgi/graphics.h>

#include "Point.h"
using namespace std;
constexpr int Gsize = 900;  // Canvas size

std::default_random_engine rgen(std::random_device{}());
std::uniform_int_distribution<int> unidist{1,Gsize - 2};

#pragma warning(disable:4244 4996)

unsigned int scale[] =
{
    0x303398,0x303398,0x303398,0x303398,0x303398,
    0x303398,0x303398,0x303398,0x303398,
    0x303398,0x4074b7,0x72aed3,0xaad9e9,0xe0f3fa,
    0xfefcbc,0xffe18c,0xffae5a,0xf56d3a,0xdb2d20,
    0xdb2d20,0xdb2d20,0xdb2d20,0xdb2d20,0xdb2d20,0xdb2d20,0xdb2d20
};

int colorScale(double value,
                        size_t count,
                        unsigned int * cols)
{
    double step = 1./(count-1);
    int no = floor(value/step);
    auto cl = [=](int C)
    {
        unsigned int mask[] = { 0xFF0000, 0xFF00, 0xFF };
        unsigned int shft[] = { 16, 8, 0 };
        int c1 = int((cols[no+1]&mask[C])>>shft[C]);
        int c0 = int((cols[no  ]&mask[C])>>shft[C]);
```

```
        return (unsigned int)((value - no*step)/step*(c1-c0) + c0)&0xFF;
    };
    return rgb(cl(0),cl(1),cl(2));
}
// To draw polygone
void draw(int color, const std::vector<poly::Point>& a, bool points = true)
{
    if (points)
    {
        setcolor(WHITE);
        for(size_t i = 0; i < a.size(); ++i)
            fillellipse(a[i].x,Gsize-a[i].y,3,3);
    }
    setcolor(color);
    for(size_t i = 0, j = a.size() - 1; i < a.size(); j = i++)
        line(a[j].x,Gsize-a[j].y,a[i].x,Gsize-a[i].y);
}
// To draw kernel
void draw(int color, const std::vector<poly::KernelVertex>& a, bool points = true)
{
    if (points)
    {
        setcolor(WHITE);
        for(size_t i = 0; i < a.size(); ++i)
            fillellipse(a[i].p.x,Gsize-a[i].p.y,3,3);
    }
    setcolor(color);
    for(size_t i = 0, j = a.size() - 1; i < a.size(); j = i++)
        line(a[j].p.x,Gsize-a[j].p.y,a[i].p.x,Gsize-a[i].p.y);
}
int main(int argc, char ** argv)
{
    vector<poly::Point> a;   // Initial set
    int PointCount = 25;
    if (argc > 1) PointCount = atoi(argv[1]);
    if (PointCount == 0) // datafile!!
    {
        // Datafile format:
        //  x_i  y_i           N points coordinates
        ifstream in(argv[1]);
        for(;;)
        {
            double x, y;
            if (in >> x >> y)
            {
                a.emplace_back(poly::Point(x,y));
                PointCount++;
            }
            else break;
        }
    }
    int gd = CUSTOM, gm = CUSTOM_MODE(Gsize,Gsize);;
    initgraph(&gd, &gm, "RGB");
    if (a.size()== 0)  // not filled yet
    for(size_t i = 0; i < PointCount; ++i)
    {
        int x = unidist(rgen), y = unidist(rgen);
        a.emplace_back(poly::Point(x,y));
    }
    if (true) // Randimize
    {
        cleardevice();
        for(auto p: a) fillellipse(p.x,Gsize-p.y,3,3);
        // Build convex hull
```

```
    auto hull = poly::convexHull(a);
    // Draw it
    draw(LIGHTMAGENTA,hull);
    setcolor(LIGHTGREEN);
    // Area
    char txt[40];
    sprintf(txt,"S = %.3lf",poly::Area(hull));
    settextstyle(SANS_SERIF_FONT,HORIZ_DIR,10);
    outtextxy(50,50,txt);
    readkey();
    // MonteCarlo
    std::vector<poly::Point> m, M;
    poly::Point pm, pM;
    double sm = poly::Area(hull), sM = 0;
    for(int i = 0; i < 100000; ++i)
    {
        poly::Point pt(unidist(rgen),unidist(rgen));
        if (!poly::inHull(pt,hull)) { --i; continue; }
        auto r = poly::starPoly(pt,a);
        double s = poly::Area(r);
        if (s < sm)
        {
            cout << (sm = s) << endl;
            m = r;
            pm = pt;
        }
        if (s > sM)
        {
            cout << (sM = s) << endl;
            M = r;
            pM = pt;
        }
    }
    // draw minimal polygone
    setcolor(LIGHTCYAN);
    fillellipse(pm.x,Gsize-pm.y,3,3);
    setcolor(LIGHTCYAN);
    draw(LIGHTCYAN,m);
    sprintf(txt,"S = %.3lf",sm);
    settextstyle(SANS_SERIF_FONT,HORIZ_DIR,10);
    outtextxy(pm.x,Gsize-pm.y,txt);
    readkey();
    // draw maximal polygone
    cleardevice();
    for(auto p: a) fillellipse(p.x,Gsize-p.y,3,3);
    // Build convex hull
    hull = poly::convexHull(a);
    // Draw it
    draw(LIGHTMAGENTA,hull);
    setcolor(LIGHTGREEN);
    sprintf(txt,"S = %.3lf",poly::Area(hull));
    settextstyle(SANS_SERIF_FONT,HORIZ_DIR,10);
    outtextxy(50,50,txt);
    setcolor(YELLOW);
    fillellipse(pM.x,Gsize-pM.y,3,3);
    setcolor(YELLOW);
    draw(YELLOW,M);
    sprintf(txt,"S = %.3lf",sM);
    settextstyle(SANS_SERIF_FONT,HORIZ_DIR,10);
    outtextxy(pM.x,Gsize-pM.y,txt);
    readkey();
}
if (true) // kernels exhausting search
{
```

```
cleardevice();
auto area = poly::Area(poly::convexHull(a));
std::vector<poly::Point> m, M;
double sm = area, sM = 0;
// First point/kernel
poly::Point pc;
for(auto p: a)
{
    fillellipse(p.x,Gsize-p.y,3,3);
    pc.x += p.x;
    pc.y += p.y;
}
pc.x /= a.size();
pc.y /= a.size();
auto star = poly::starPoly(pc,a);
poly::StarKernel krnl(star);
setfillstyle(SOLID_FILL,BLUE);
fillellipse(pc.x,Gsize-pc.y,5,5);
// BFS
queue<poly::StarKernel> Q;
unordered_set<poly::StarKernel,poly::StarKernelHash,
              poly::StarKernelComp> S;
S.insert(krnl);
Q.push(krnl);
while(!Q.empty())
{
    poly::StarKernel K = Q.front();
    Q.pop();
    // Kernel polygone
    vector<int> d;
    for(auto z: K.k)
    {
        d.push_back(z.p.x);
        d.push_back(Gsize - z.p.y);
    }
    // Save min, max
    double sq = poly::Area(K.s);
    double value = sq/area;
    if (sq < sm)
    {
        cout << (sm = sq) << endl;
        m = K.s;
    }
    if (sq > sM)
    {
        cout << (sM = sq) << endl;
        M = K.s;
    }
    // draw filled kernel
    setcolor(colorScale(value,size(scale),scale));
    setfillstyle(SOLID_FILL, colorScale(value,size(scale),scale));
    fillpoly(int(d.size()/2),d.data());
    for(size_t e = 0; e < K.k.size(); ++e)
    {
        poly::StarKernel K2(K,e);
        if (K2.k.size() == 0) continue;
        if (S.insert(K2).second)
        {
            //draw(YELLOW,K2.k,false); //Kernel borders?
            Q.push(K2);
        }
    }
}
readkey();
```

```
        cleardevice();
        char txt[20];
        setcolor(LIGHTCYAN);
        draw(LIGHTCYAN,m);
        sprintf(txt,"S = %.3lf",sm);
        settextstyle(SANS_SERIF_FONT,HORIZ_DIR,10);
        outtextxy(100,Gsize-100,txt);
        readkey();
        cleardevice();
        setcolor(YELLOW);
        draw(YELLOW,M);
        sprintf(txt,"S = %.3lf",sM);
        settextstyle(SANS_SERIF_FONT,HORIZ_DIR,10);
        outtextxy(100,Gsize-100,txt);
        readkey();
    }
    closegraph();
}
```