Министерство образования и науки Российской Федерации федеральное государственное
автономное образовательное учреждение высшего образования
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет «Программной инженерии и компьютерной техники.»

Алгоритмы и структуры данных

Лабораторная работа №2
Timus

Выполнил
Григорьев Давид Владимирович
Группа: P3215
Преподаватели
Косяков Михаил Сергеевич
Тараканов Денис Сергеевич

Санкт-Петербург 2025

Содержание

1 1444

Пояснение к примененному алгоритму

Наша задача создать **планарный** граф на данных вершинах. Так как известно, что не может быть двух тыкв на одной координате, это означает, что максимальная степень вершины будет два, и граф будет без циклов.

по т. Куратовского, граф планарный тогда и только тогда, когда нет под разбиений $K_{3,3}$ и K_5. По скольку, $\Delta(G) = 2$, такого быть не может.

Значит, всегда существует путь, охватывающий все вершины.

Наша задача найти его.

Возможно, стоит воспользоваться стратегией, где мы идем сначала снаружи, и заканчиваем внутри графа.

По скольку все точки не могут быть на одной линии.

Предложение:

- в каждой вершины мы будем сканировать по часовой стрелке все вершины начиная с самого левого положения.

Вместо этого легче идти зигзагом

Код алгоритма

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>

namespace {
struct Point {
  int x, y;
  int index;
};

int SquaredLength(const Point& point) {
  return (point.x * point.x) + (point.y * point.y);
}

int DotProduct(const Point& point_a, const Point& point_b) {
  return (point_a.x * point_b.x) + (point_a.y * point_b.y);
}

int CrossProduct(const Point& point_a, const Point& point_b) {
  return (point_a.x * point_b.y) - (point_a.y * point_b.x);
}

bool AreCollinear(const Point& point_a, const Point& point_b) {
  return CrossProduct(point_a, point_b) == 0;
}

}  // namespace

int main() {
  int number_of_points = 0;
  std::cin >> number_of_points;

  std::vector<Point> points(number_of_points);
```

```
  for (int i = 0; i < number_of_points; i++) {
    std::cin >> points[i].x;
    std::cin >> points[i].y;
    points[i].index = i;
  }

  for (int i = 1; i <= number_of_points; i++) {
    // Translate everything towards the first point
    points[i].x -= points[0].x;
    points[i].y -= points[0].y;
  }

  // now first point is in 0.0
  points[0].x = 0;
  points[0].y = 0;

  std::sort(points.begin() + 1, points.end(), [](const Point& point_a, const Point& poin
    // Points with the same angle are sorted outwards
    if (AreCollinear(point_a, point_b) && DotProduct(point_a, point_b) >= 0) {
      return SquaredLength(point_a) < SquaredLength(point_b);
    }

    // atan2 is a function of how "left" the vectors are pointing
    // when we sort it like this we scan it from left to right
    return atan2(point_a.y, point_a.x) < atan2(point_b.y, point_b.x);
  });

  int start_index = 0;
  for (int i = 0; i < number_of_points - 1; i++) {
    Point current_point = points[i];
    Point next_point = points[i + 1];

    bool is_clockwise = CrossProduct(current_point, next_point) < 0;
    bool is_pointing_back = DotProduct(current_point, next_point) < 0;
    // Opposite turn, start at that point instead
    if (is_clockwise || AreCollinear(current_point, next_point) && is_pointing_back) {
      start_index = i;
      break;
    }
  }

  std::cout << number_of_points << '\n';

  std::cout << (points[0].index + 1) << '\n';
  for (int i = start_index; i < start_index + number_of_points - 1; i++) {
    std::cout << (points[(i % (number_of_points - 1)) + 1].index + 1) << '\n';
  }

  return 0;
}
```

2 1726

Пояснения к примененному алгоритму

Итоговое среднее

Суммарные горизонтальные и вертикальные расстояния складываются, после чего результат делится на количество пар ($n \cdot (n-1)$) для получения среднего значения.

Эффективность

- Сортировка обеспечивает сложность $O(n \log n)$, а префиксные суммы работают за $O(n)$.

Код алгоритма

```cpp
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <vector>

namespace {

size_t ComputeContribution(const std::vector<size_t>& coords) {
  size_t number_of_houses = coords.size();

  std::vector<size_t> prefix(number_of_houses + 1, 0);

  for (size_t i = 0; i < number_of_houses; ++i) {
    prefix[i + 1] = prefix[i] + coords[i];
  }

  size_t total = 0;

  for (size_t i = 0; i < number_of_houses; ++i) {
    size_t sum_before = prefix[i];
    size_t sum_after = prefix[number_of_houses] - prefix[i + 1];
    size_t contribution_before = ((coords[i] * i) - sum_before);
    size_t contribution_after = (sum_after - (coords[i] * (number_of_houses - 1 - i)));
    total += contribution_before + contribution_after;
  }
  return total;
}
}  // namespace

int main() {
  size_t number_of_houses = 0;
  std::cin >> number_of_houses;

  std::vector<size_t> x_coords(number_of_houses);
  std::vector<size_t> y_coords(number_of_houses);

  for (size_t i = 0; i < number_of_houses; ++i) {
    std::cin >> x_coords[i];
    std::cin >> y_coords[i];
```

```
    }

    std::sort(x_coords.begin(), x_coords.end());
    std::sort(y_coords.begin(), y_coords.end());

    size_t sum_x = ComputeContribution(x_coords);
    size_t sum_y = ComputeContribution(y_coords);

    size_t total = sum_x + sum_y;
    std::cout << total / (number_of_houses * (number_of_houses - 1)) << '\n';
    return 0;
}
```