Computational Geometry: Principles and Practices

Chen Shaoyuan

Nanjing University ICPC Training Team

August 8, 2019

Guidelines in Solving Geometry Problems

Rule 1: Prefer vectors to parameters in equations when representing geometric objects

For example, use a point (point can be viewed as a vector from the origin) and a directional vector to represent a straight line, instead of using the slope k and intercept b.

Rule 1: Prefer vectors to parameters in equations when representing geometric objects

For example, use a point (point can be viewed as a vector from the origin) and a directional vector to represent a straight line, instead of using the slope k and intercept b.

- Vectors have clearer geometric meanings than parameters;

Rule 1: Prefer vectors to parameters in equations when representing geometric objects

For example, use a point (point can be viewed as a vector from the origin) and a directional vector to represent a straight line, instead of using the slope k and intercept b.

- Vectors have clearer geometric meanings than parameters;
- Vectors have predefined aggregate operations (vector addition/subtraction, scalar multiplication, inner/outer product); when processing parameters we can only use atomic operations (scalar arithmetics).

Rule 1: Prefer vectors to parameters in equations when representing geometric objects

For example, use a point (point can be viewed as a vector from the origin) and a directional vector to represent a straight line, instead of using the slope k and intercept b.

- Vectors have clearer geometric meanings than parameters;
- Vectors have predefined aggregate operations (vector addition/subtraction, scalar multiplication, inner/outer product); when processing parameters we can only use atomic operations (scalar arithmetics).
- There is usually no degenerate case in vector-based representations.

Guidelines in Solving Geometry Problems

Implementation trick: a short yet powerful vector class

```
typedef double T;
typedef complex<T> pt, vec;
inline T operator , (pt a, pt b) // inner product
  { return real(a) * real(b) + imag(a) * imag(b); }
inline T operator * (pt a, pt b) // outer product
  { return real(a) * imag(b) - imag(a) * real(b); }
```

Guidelines in Solving Geometry Problems

Implementation trick: a short yet powerful vector class

```
typedef double T;
typedef complex<T> pt, vec;
inline T operator , (pt a, pt b) // inner product
  { return real(a) * real(b) + imag(a) * imag(b); }
inline T operator * (pt a, pt b) // outer product
  { return real(a) * imag(b) - imag(a) * real(b); }
```

Pros: vector addition/subtraction and scalar multiplication are provided by `std::complex`. Also, we may use functions applicable to `std::complex`, e.g., `std::abs` to get the length of the vector.

Cons: accessing individual component is a bit tedious. You may use `real` and `imag` functions.

Rule 2: Use integer arithmetics whenever possible

Rule 2: Use integer arithmetics whenever possible

- Integer arithmetics have no precision issue, which prevents you from falling into epsilon-tuning trap.

Guidelines in Solving Geometry Problems

Rule 2: Use integer arithmetics whenever possible

- Integer arithmetics have no precision issue, which prevents you from falling into epsilon-tuning trap.

Rule 3: Think twice before tuning epsilon

Guidelines in Solving Geometry Problems

Rule 2: Use integer arithmetics whenever possible

- Integer arithmetics have no precision issue, which prevents you from falling into epsilon-tuning trap.

Rule 3: Think twice before tuning epsilon

- Most computational geometry problems, especially those that the output can be written as a continuous function of its input, do not need an epsilon.

Rule 2: Use integer arithmetics whenever possible

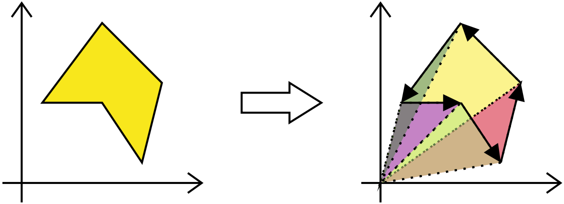
- Integer arithmetics have no precision issue, which prevents you from falling into epsilon-tuning trap.

Rule 3: Think twice before tuning epsilon

- Most computational geometry problems, especially those that the output can be written as a continuous function of its input, do not need an epsilon.
- Even though a problem indeed requires an epsilon, it is more often that other part of your code causes the Wrong Answer.

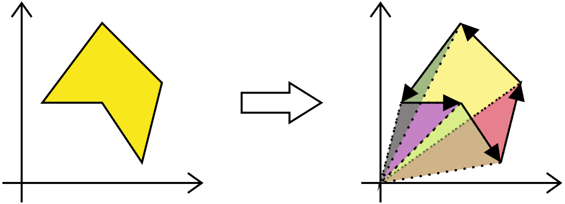
Triangle Partition

The triangular partition method arises from computing the area of a polygon: partition the polygon into several directed triangles, and compute the sum of the signed areas of the triangles.



Triangle Partition

The triangular partition method arises from computing the area of a polygon: partition the polygon into several directed triangles, and compute the sum of the signed areas of the triangles.



This yields the shoelace formula (aka Gauss's area formula or surveyor's formula):

$$S_P = \frac{1}{2}\left|\sum_{i=0}^{n-1} \vec{P}_i \times \vec{P}_j\right| \quad (P_n = P_0)$$

Triangle Partition

From the view of calculus, this method is derived from the Green's formula:

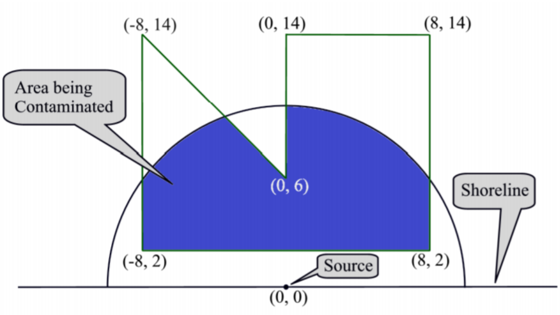
$$\iint_P \left(\frac{\partial M}{\partial x} - \frac{\partial L}{\partial y} \right) dxdy = \oint_{\partial P} (Ldx + Mdy)$$

and thus it can be used to compute double integral over a polygon.

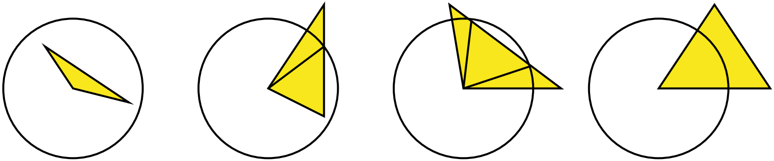
Triangle Partition Method

ICPC WF'13 J: Pollution Solution

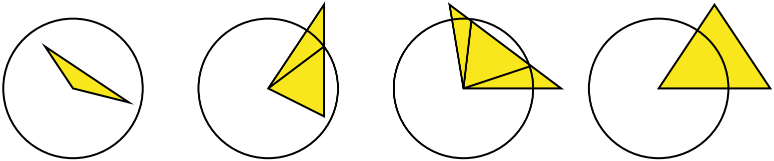
Find the area of the intersection of a polygon and a circle.



Still partition the polygon into triangles. Compute the intersection of each triangle and the circle, and sum up their signed areas. (If the center of the circle is not origin, translate the coordinate system such that the center becomes the origin.)



Still partition the polygon into triangles. Compute the intersection of each triangle and the circle, and sum up their signed areas. (If the center of the circle is not origin, translate the coordinate system such that the center becomes the origin.)



The first and the fourth can be computed directly. The second and third can be further partitioned to several triangles and compute separately.

Define a point set S:

$$S = \{(x,y) : 2k \leq x \leq 2k+1, k \in \mathbb{Z}\}.$$

Given a polygon P. Compute the area of the intersection of P and S.

Define a point set S:

$$S = \{(x, y) : 2k \leq x \leq 2k+1, k \in \mathbb{Z}\}.$$

Given a polygon P. Compute the area of the intersection of P and S.

In this problem, we may partition the polygon into several trapezoids (instead of triangles), and compute their contributions separately.

Exercise:

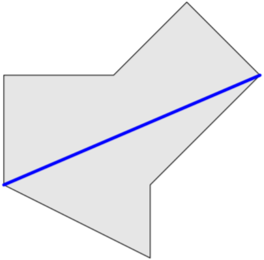
2019 MW-Bytedance Camp, Day 2, Divison A: D. Cross-section

Enumerating Local Optima

There are many optimization problems in computational geometry. They usually requires to find a geometric object, possibly under some conditions, such that some value is minimized/maximized.

In most cases the set of all feasible solutions is infinite. However, for these problems, the set of all local optima is often finite! This enables us to enumerate all local optima and pick the most optimal one.

Given a polygon (not necessarily convex), find a line segment entirely lies in the polygon, such that the length is maximized. (number of vertices does not exceed 200)



Observation: the optimal line segment must pass through at least two vertices of the polygon.

Observation: the optimal line segment must pass through at least two vertices of the polygon.

1. the endpoints of the line segment must be on the border of the polygon;

Observation: the optimal line segment must pass through at least two vertices of the polygon.

1. the endpoints of the line segment must be on the border of the polygon;
2. if the line segment does not pass any vertex, translate the segment in some direction which will increase the length, until it touches a vertex of the polygon;

Observation: the optimal line segment must pass through at least two vertices of the polygon.

1. the endpoints of the line segment must be on the border of the polygon;
2. if the line segment does not pass any vertex, translate the segment in some direction which will increase the length, until it touches a vertex of the polygon;
3. if the line segment passes only one vertex, rotate the segment about the vertex clockwise or counterclockwise, depending on which one will increase the length of the segment.

The algorithm

1. for each vertex pair A, B:
    1. check if segment AB is entirely in the polygon;
    2. if so, extend the segment as far as possible.
2. among all possible extended segments, pick the longest one.

The algorithm

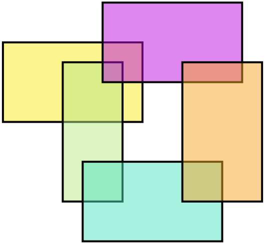
1. for each vertex pair A, B:
    1. check if segment AB is entirely in the polygon;
    2. if so, extend the segment as far as possible.
2. among all possible extended segments, pick the longest one.

Checking if a segment is entirely in the polygon, and extending the segment can both be done in $O(n)$ time. The total time complexity is $O(n^3)$.

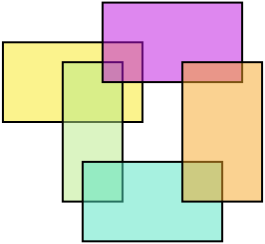
Exercise:

NJUPC'19 H. Road Construction

Given a set of orthogonal rectangles (sides parallel to axes). How to efficiently compute the area of their union?



Given a set of orthogonal rectangles (sides parallel to axes). How to efficiently compute the area of their union?



Consider a line scans from left to right. Let $L(x_0)$ denote the total lengths of line $x = x_0$ clipped by the union of the rectangles. The total area is simply $\int_{-\infty}^{\infty} L(x)dx$.

The algorithm:

- Imagine a line scans from left to right.
- When the line enters a rectangle, add the vertically clipped segment into the set of intervals.
- When the line leaves a rectangle, delete the segment from the set of intervals.
- Before processing any of the above events, add to the answer the total length of the union of the intervals, times the distance of the scan line traveled since last event.

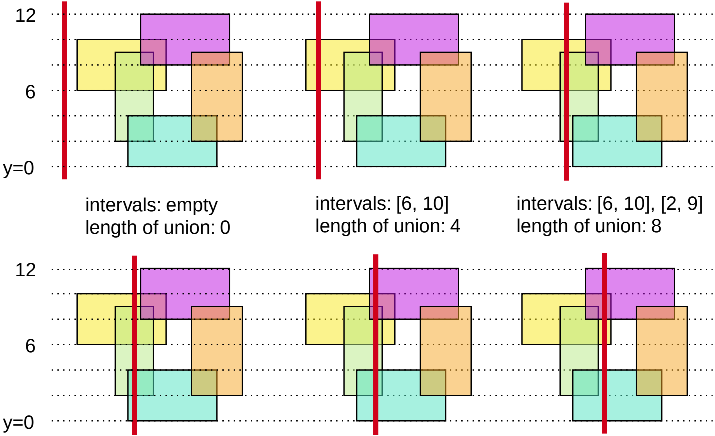
The algorithm:

- Imagine a line scans from left to right.
- When the line enters a rectangle, add the vertically clipped segment into the set of intervals.
- When the line leaves a rectangle, delete the segment from the set of intervals.
- Before processing any of the above events, add to the answer the total length of the union of the intervals, times the distance of the scan line traveled since last event.

We need to maintain a set of intervals and the total length of their union. The naive solution gives $O(n^2)$ time. If we use data structures like segment tree or binary search tree, the total time is $O(n \log n)$.

Sweep Line Algorithm

An Introductory Example



intervals: empty
length of union: 0

intervals: [6, 10]
length of union: 4

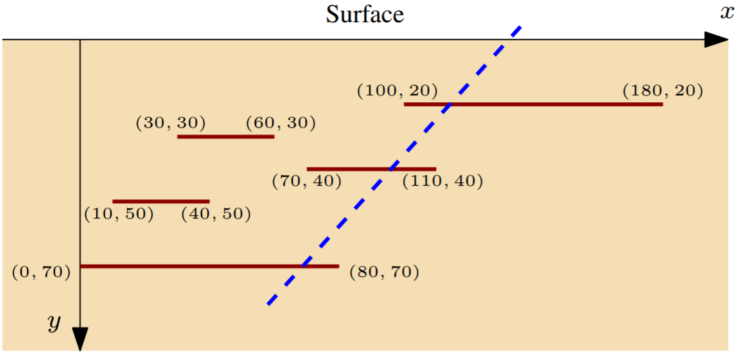
intervals: [6, 10], [2, 9]
length of union: 8

intervals: [6, 10],
[2, 9], [0, 4]
length of union: 10

intervals: [6, 10], [2, 9]
[0, 4], [8, 12]
length of union: 12

intervals: [6, 10], [0, 4]
[8, 12]
length of union: 10

Given a set of horizontal line segments, find a line that intersects maximum number of them. There are at most 2000 segments. No two segments intersect, not even at a point.



This is an optimization problem. We can use the **enumerating local optima** paradigm!

This is an optimization problem. We can use the **enumerating local optima** paradigm!

Not hard to prove that the optimal line passes through at least two end points of these line segments.

This is an optimization problem. We can use the **enumerating local optima** paradigm!

Not hard to prove that the optimal line passes through at least two end points of these line segments.

However, enumerating all such lines and counting the number of intersections for each of these lines take $O(n^3)$ time.

This is an optimization problem. We can use the **enumerating local optima** paradigm!

Not hard to prove that the optimal line passes through at least two end points of these line segments.

However, enumerating all such lines and counting the number of intersections for each of these lines take $O(n^3)$ time.

We may enumerate one end point. Consider a line passing this end point, and we rotate this line. During rotation, several *enter* and *leave* events occur, and we only have to maintain the number of intersections. Just sort all other points by their polar angles to the fixed end points. The total time complexity is thus reduced to $O(n^2 \log n)$.

Sweep Line Algorithm

Exercise:

Determining whether any two line segments in a set of segments intersect, in $O(n \log n)$ time.

Overview

Three rules on writing geometry problems:

1. Prefer vectors to parameters in equations when representing geometric objects;
2. Use integer arithmetics whenever possible;
3. Think twice before tuning epsilon.

Three algorithmic paradigms on solving geometry problems:

1. Triangle/trapezoid partition;
2. Enumerating local optima;
3. (Rotational) sweep line.