Computational Geometry-Project 3

Group 3
Bingsen Wang - au641000
Zhile Jiang - au641092
Fatemeh Zardbani - au640364

November 28, 2019

1 unbounded cells

First, given any arrangement A, we create a big enough triangle which include all the vertex of A in it like below.

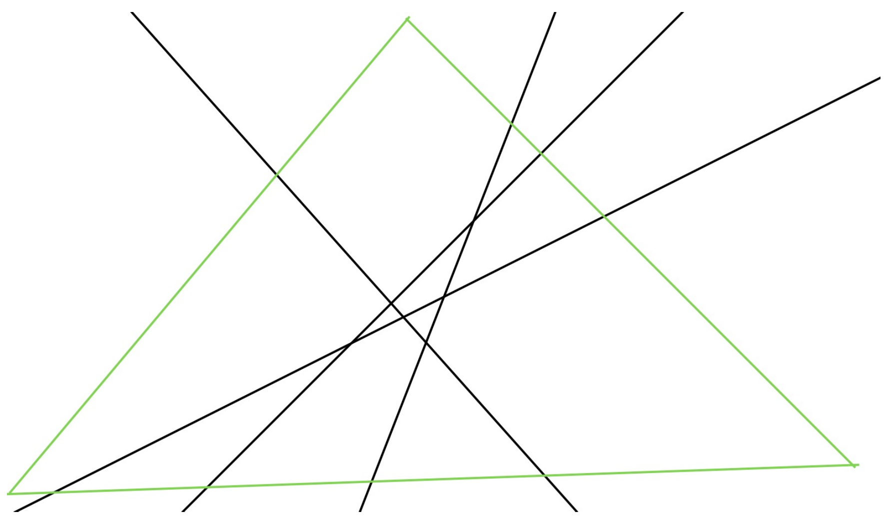


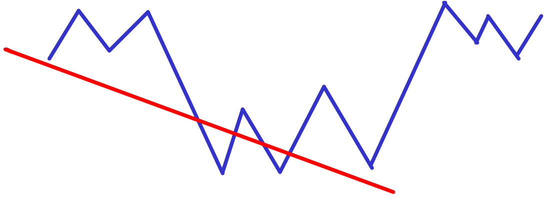
Figure 1: all the vertex of the arrangement are in the triangle

Since all the unbounded cells will extend into infinite space, we can make sure that the three line of triangle will cross all the unbounded cells.
Therefore, all the vertex, edges, faces in any unbounded cell will be in the zone of three lines of the big triangle.
Now, we can use the *Zone Theorem* to give a bound to the complexity of it. The number of line in the arrangement is n, then the complexity of the zone of one line of triangle will be $O(n)$. 3 times of $O(n)$ is still $O(n)$.
Therefore, we've prove that the total complexity of all the unbounded cells is linear to the number of lines in the arrangement.

A more efficient way is to use the zone of the line at infinity. All the unbounded cells are in the zone of the line at infinity so you just need to use the zone theorem once!

2 dividing lines

First, we need to sort all the points by X-coordinate. And we name all the points by their order, the i-the point in X-coordinate is point i, where $i = 1, 2, \ldots, n-1$. For any point i we could know a hull with all the points in its left, we call this left-hull i, and another hull with all the points in its right called right-hull i.



It is actually more complicated.
What if the first and the last segments are both above the line?

If there is a dividing line between point k and point $k+1$ where $k = 1, 2, \ldots, n-1$, this line must have an intersection with line segment determined by point k and point $k+1$. To decide whether a line is a dividing line, we need to find out the k makes this kind of line segment at first. We could using binary search to find k, it is easy and need $O(\log n)$ time to do search and $O(n)$ space to store the points in order. If we cannot find k, all the points lies on the same side of the given line. So it must be a dividing line. And note there can be more than one k for a non-dividing line, but we only need to find any one of them. The reason is the non-dividing line cannot pass the following test using any k determining the line segment we previously mentioned.

We said it is easy because we assume the given line is a dividing line line. And then try to check if it is really a dividing line.
If it is really a dividing line, there must be only one segment that two endpoints are on different sides. And using binary search to find this segment is easy.
But we will always meet such a segment even if the given line is not a dividing line.
And the answer to your question is it doesn't matter. Because we start at middle. If we find both endpoints is below, we go right. above, go right. If they are on the different side, that is the segment we want.

A line is dividing line if and only if it has no intersection with left-hull k and right-hull k. This could be done in $O(\log n)$ time using binary search on hull. In more detail, we store a hull by storing all the points on hull clockwise or counterclockwise, so it is automatically sort by slope. And to decide whether a line has intersection with a hull, we only need to find two points (there must be two and only two) which makes slope of line segment on hull minus slope of the given line changes. And if two points lie to the same side of given line, the hull has no intersection with the line. Otherwise, there are intersections.

In this way, we could determine whether a line is dividing line using $O(\log n)$ time. But if we store all the hulls directly, the space is $O(n^2)$. So we need two partially persistent black-red tree to store all the hulls, one for all left-hulls and the other for all right-hulls. The two trees is very same. So we only show how to build up the tree for left-hulls The tree and the hull start at empty. From left to right, we add a point to hull each time. It will add one point and may remove some points on hull. And we need to add and remove the same points for the black-red trees. All the points will be added once and be removed at most once. So there is at most $2n$ changes. And since A red-black tree can be rebalanced using $O(1)$ rotations for a change. We only need $O(n)$ space for each tree. The total space is also $O(n)$.

True!

Nice idea!

Very nice!

I think this works and it's a very nice solution that I had not seen before! There are some details that need to be taken care of but I take it!

Therefore, we got the data structure of linear size space could decide whether a line is dividing line in $O(\log n)$ time.

3 halving lines

Figure 2: Both points are above the pivot.

Figure 3: Both points are below the pivot.

Figure 4: Points are on different side.

Before starting the algorithm, we want to explain a conclusion: Given $2n$ points on a plain and there are no three points lie on a line, a line with direction determined by 2 points and assume that there are k points on the left side of this line and $2n-k-2$ point on the right side. Choose a point on the given line as *pivot* and call the other point as *start point*. Then rotate the line clockwise until it touch next point. There is, if the next point and start point are both above pivot, the number of point on the left side will become $k+1$ and the number of point on the left side will become $2n-k-3$ (Figure 2); if the next point and start point are both below pivot, the number of point on the left side will become $k-1$ and the number of point on the left side will become $2n-k-1$ (Figure 3); if the next point and start point are on different side on the line, the number of points on left and right will not change (Figure 4).

You can make ideas like this work. In fact observations like these have been used in getting bounds on the number of halving lines so I'm happy that you have discovered some of them.

However, as you say, this solution is not complete yet, it gets close but not there.

As a hint you should look at the dual space.

And if we using the new point as pivot for each rotation, the number of points on left cannot increase continuously. The reason is if an increase happens, that means the new point is above the pivot. While we use the new point as pivot, the old pivot will be new start point. And the start point must be below the pivot. The only change may happen on the number of points lies on left is decrease, or there is no change. The same thing also happens on decrease situation.

So the algorithm is: First, using the most left point as center, sort all the points by the polar angles; Second, using center as pivot, rotate a line to find a halving line; Last, rotate the halving line and each time let the new point we got as pivot until we meet a line we have already met. The lines with $n-1$ points in the left is what we want.

This problem is much easier in dual space. The points become lines in dual space. And the intersections is the line in original problem. The problem is to count the intersections with both n-1 lines below and above it. This could be solved be maintaining the level of point which is on page 185-186, chapter 8 of textbook.

The time-consuming is $O(n \log n) + O(n) + \textit{the number of halving lines} * O(n)$. And since we know the number of halving lines is super linear. So the time is very near $O(n^2)$. (There is a conjecture invented by Erds, Strauss, Lovasz, and Simonovits says the upper bound is always upper bounded by $n^{(1+o(1))}$.)

4 Voronoi Diagrams in 3D

I think this makese sense and it is a correct construction.

Should $\frac{n}{2}$ of points be set on line segment $L_1 = \{(x, 0, 0) : x \in [0, \frac{1}{2}]\}$ and the other half on $L_2 = \{(1, y, 1) : y \in [\frac{1}{2}, 1]\}$, then we will have that the $|V|$ is $\Theta(n^2)$. As the complexity of the Voronoi Diagram is $|V|+|E|+|F|$ in 3 dimensions, then it will also be $\Theta(n^2)$. A simple explanation would be that to create the cell for a point p on the L_2, we will have to create the intersection of the half-planes generated by the bisectors. The intersection of the bisector half-planes for all the points below p on L_2, will be above the bisector half-plane of the point exactly below it, same goes for the points above it on L_2. SO far, we have the space between these two parallel half-planes; we will call them $l1, l2$. For all the points on L_1, we will have a half-plane that intersects with $l1$ and $l2$, creating 2 new vertices. As there are $\frac{n}{2}$ points on L_1, the total number of vertices for this cell will be of $\Theta(n)$, making the total number of vertices, of $\Theta(n^2)$.

5 Halfspace range max queries

We could translate the points and lines in x-y coordinate system to lines and points in k-b coordinate system. For a point which is (x_0, y_0) in x-y coordinate system, it is equal to a line $b = -x_0k + y_0$ in k-b coordinate system. And for a line which is $y = k_0x + b_0$ in x-y coordinate system, it is equal to a point (k_0, b_0) in k-b system.

The problem is we need to find the biggest point under the given line. A point (x_0, y_0) is below a line $y = k_0x + b_0$ can be represented as $y_0 < k_0x_0 + b_0$. This inequality is equivalent to $b_0 > -x_0k_0 + y_0$. So the problem is the same as to find the biggest line under the given point in k-b coordinate system.

And this problem can be solved by solving a point location problem. We just need one more variable to store the largest weight for each cell. The cost of a point location problem depends on the number of segments. But we only have lines, so we need to compute the number of segments we have. Considering we already have $n-1$ lines and we add another line, up to $n + (n-1)$ new segments will be generated. So number of segments for n lines is up to $\sum_{i=1}^{n} 2 * i - 1 = n^2$.

If we use a partially persistent red-black tree, to solve this problem, the time-consuming is $O(\log n^2) = O(2 \log n) = O(\log n)$. But the space we need is $O(n^2)$. However, we noticed that there are a lot of useless segments because we only care the line with largest weight under the given point. If a segment's weight is less than any below segment's weight, the segment is not necessary. So we could construct the segments by adding lines in descending order of weight. We will of course get one segment by adding the line with largest weight. Then only consider the situation we will get maximum new segment. For the second line, we will get 2. And we will get up to 3 segments for each line added after. So the total number of necessary segments is $1 + 2 + 3 * (n-2) = 3n - 3$. The using space become $O(n)$.

Why do we always get at most 3? You need to mention a key property of the "outside" region.

The rest is correct though.

The reason is that the only one side of the line we added will still be useful. So the problem is just like we are adding some half-planes and we geta convex hull. And only the segments in or on the hull retain. So it is up to 3 new segments.

In conclusion, we could solve this problem by solving its dual problem using a partially persistent red-black tree. The tree we use is the same as the tree used in point location problem. We store the largest weight of segments under each cell and use the tree to find the cell that given point locates in. The segments are constructed by by adding lines in descending order of weight. We only keep the segments without any larger segment above them.

6 area computation

6.1 Sweep line

We make the sweep line vertical to the y-axe, the event point would be all the sides of all the rectangles parallel to x-axe.

While sweeping, we use a queue Q to store all the event points , a balanced binary search tree T to maintain all the disjoint intervals, a real number $X = 0$ to store the current total length of all the interval and $Y = +inf$ to store the current y coordinate, of course real number $ANS = 0$ for the total area.

Before sweeping, we need to sort all the event point and store them into Q, this will take $O(nlogn)$ time.

For each event point, it would be a begin or end of a rectangle.

First, we should get the new value for Y from the y coordinate of this side, name it as Y_n, and update

$$ANS = ANS + (Y_n - Y) * X, Y = Y_n$$

More details are needed to complete this solution.
How do you do the insertion and deletion to maintain the information you need?

Second, we should add or delete this side as a interval$[a, b]$ to T. We find all the interval in T which include a or b named $intervals$, for add, we delete $intervals$ and add the union of $[a, b]$ and $intervals$, for delete, we delete $intervals$ and add the joint of $intervals$, $[-inf, a]$,$[b, +inf]$. For each addition or deletion, we should update X.

Final, when Q is empty, output ANS.

Clearly, the number of event points is $O(n)$, for each event point we will do $O(1)$ times binary tree operation which will take $O(logn)$. Therefore, including the initiation of event point which is $O(nlogn)$, the total time complexity will be $O(nlogn)$.

6.2 Divide-and-Conquer

6.2.1 Strip form of rectangles

We create a big enough which will enclose all the rectangles in it and extend all the side vertical to y-axe. Like below.

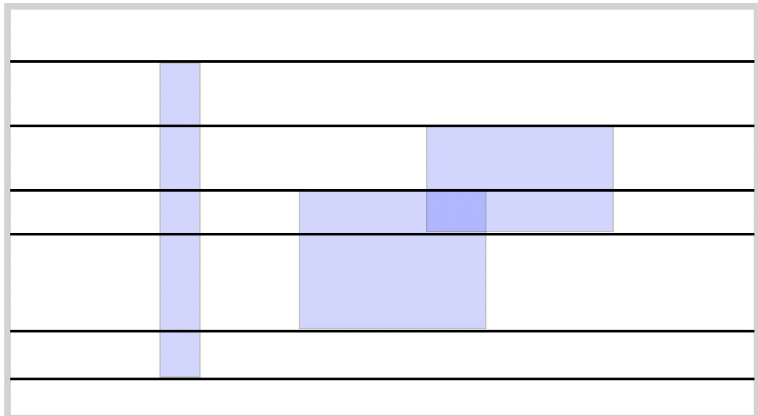


Figure 5: Strip form of several rectangles

Now, if for each strip, we know the total length of covered area, we can easily compute the total area since we can easily get the height of each strip.

6.2.2 Compute the strip form with divide and conquer

In this section, we give a brief sketch of how to compute the strip form of n rectangles with maintaining the covered length of each strip.
First, with input of n rectangles, we represent all rectangle with it's two edges vertical to x-axe.
Now, we show that how to compute the strip form with divide and conquer.
When there is only one edge, we form a strip form like below.

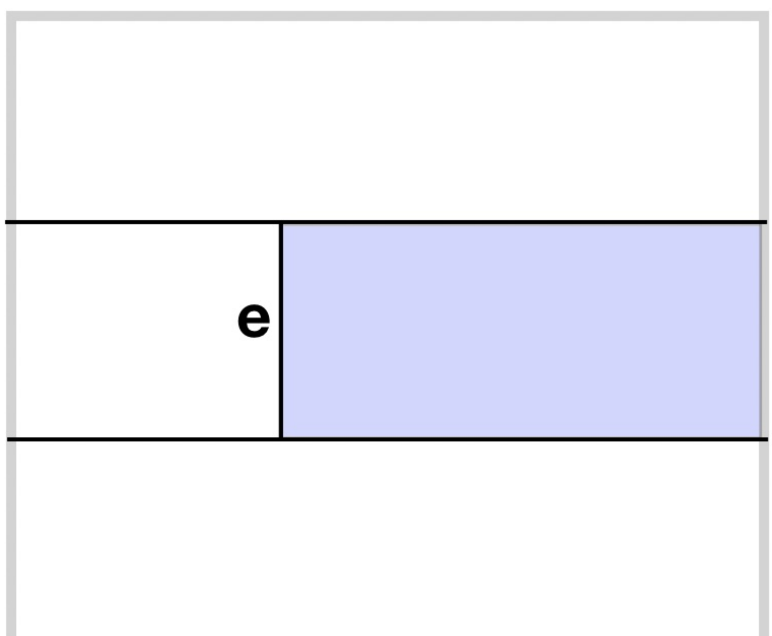


Figure 6: Strip form of one edge

When there are more than one edge, we create a line vertical to x-axe and split the whole space into two subspace S_l, S_r with nearly same number of edges, and compute strip form of S_l, S_r.
The difficult part is how to merge the strip form of S_l and S_r with maintaining the covered length of each strip.
The strip form of S_l and S_r have different partition line, so, we should first extend them and copy the corresponding intervals in them. Then, we should merge all the intervals in each strip and concat them.

6.2.3 Analysis

This is the idea, yes.

FYI, the divide and conquer solution does the following:

Given an input of n rectangles, spends O(n) time doing stuff, and also calculates some area A_0.
Then partitions the input into two sets of roughly half the size, and recursively computes some areas A_left and A_right for those.
Then outputs A_0 + A_left + A_right.
There is no merging step in this algorithm.

One hint regarding the D&C algorithm: it *modifies* the rectangles, when it passes them to the recusive call.

The detail of the divide and conquer algorithm is too complicated, We can only give a brief analysis and skip the details.
For the divide operation, it will cost O(n) time to find a good divide line.
For the merge operation, it is same to travel all the strips, so, it will take $O(n)$ time. Therefore, the time complexity is

$$T(n) = O(n) + 2T(n/2) + O(n)$$

. Using the master theorem, we can get

$$T(n) = O(nlogn)$$

7 Minkowski Sum

7.1 a

The idea is correct but I prefer a more mathematical proof with more details.

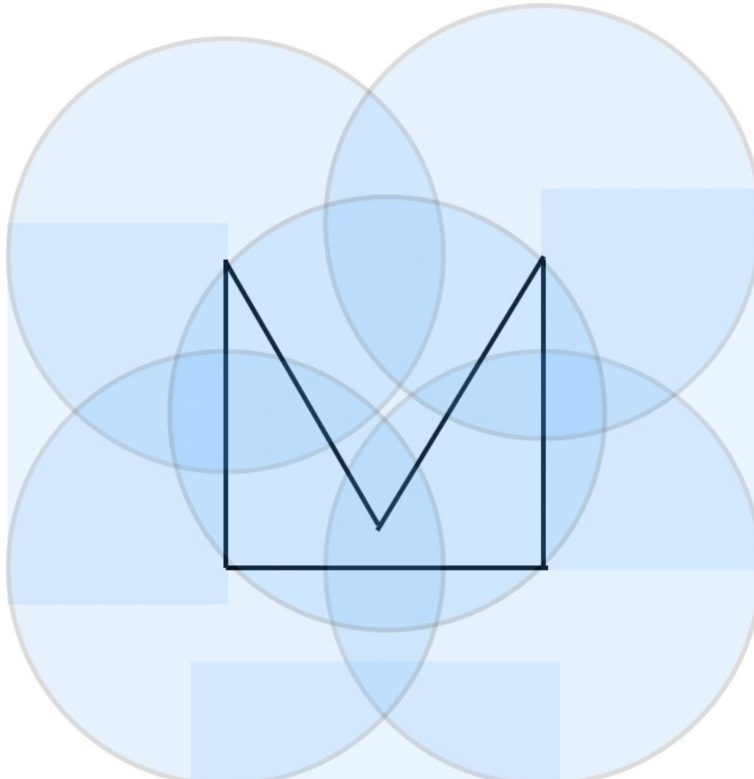
True.
Imaging we compute $P \oplus Conv(P)$ by sliding P along each edge of $Conv(P)$.
On any direction d perpendicular to an edge of $Conv(P)$, the extreme point of P on this direction would be a vertex on $Conv(P)$ and the outline of $P \oplus Conv(P)$ will be determine by this extreme point.
Therefore, we can say that

$$P \oplus Conv(P) = Conv(P) \oplus Conv(P)$$

which must be convex.

7.2 b

False. Counterexample as below.



Good!

Figure 7: Sliding a circle along the edges of a simple polygon

7.3 c

False.

Good!

The reason is same as b, no matter how big the circle is, the intersection of the two arcs will always dent.

7.4 d

True.

Imaging we compute $P \oplus Q$ by sliding P along the edge of Q, we should make sure the center of P is at the edge of Q. Like below.

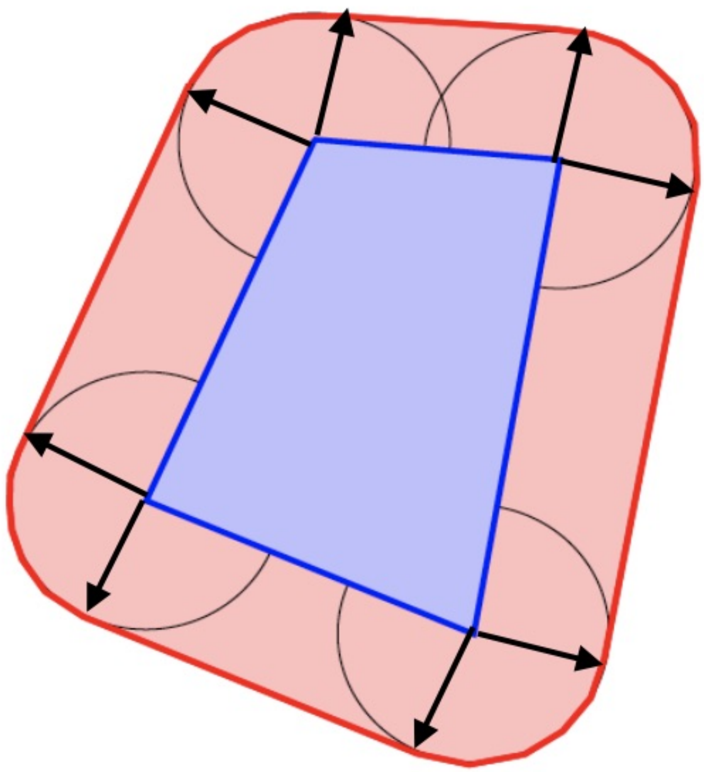


Figure 8: Sliding P along the edges of Q

Is this assuming one of them is a circle?

We can see that when the center of P is not on any vertex of Q, then the outline of $P \oplus Q$ is just shift one edge in a direction d perpendicular to it.

I think you have the main idea and this is the correct reason but again, I prefer a bit more precise formulation of the same idea.

Hint: Can you show it when P and Q are just one line segment each?

When the center of P is on a vertex of Q, the direction d should change between two edge adjacent with the vertex. For now, the outline of $P \oplus Q$ is determined by the part of the polygon P that was swept by d. During the whole process, the direction d will change total 2π, which means all edges on P will be swept exactly once. And since the edges of Q were shifted to $P \oplus Q$, the perimeter of $P \oplus Q$ is equal to the sum of the perimeters of P and Q.