DD2440 Advanced Algorithms
Lecture 13: Computational Geometry

Lecturer: Johan Håstad
Scribe: Petter Lundahl

3 December 2014

1 The setup

We have points, lines and polygons in the plane. A polygon is given as a list of n points giving the position of its vertices. We have the following typical problems:

- n points, find convex hull. Can be solved in $\mathcal{O}(n \log n)$ time.
- One polygon, one point, is it inside? Can be solved in $\mathcal{O}(n)$ time.
- Given n segments, does any pair intersect? Naively this can be in time $\mathcal{O}(n^2)$ (you need to check that we can check whether two segments intersect in time $O(1)$ but we do that below). We want a better complexity of $\mathcal{O}(n \log n)$.
- Given n points, find the closest pair. This is again easy to solve in time $\mathcal{O}(n^2)$ and we want an algorithm running in time $\mathcal{O}(n \log n)$.

To warm up, let us start with a simple problem.

1.1 Do two segments intersect?

We are given two segments (p_0, q_0) and (p_1, q_1) and want to determine if they intersect. The segmenst are parametrized by

$$\begin{aligned} S_0 &= p_0 + t(q_0 - p_0) \quad t \in [0,1] \quad p_0, q_0 \in \mathbb{R}^2 \\ S_1 &= p_1 + s(q_1 - p_1) \quad s \in [0,1] \quad p_1, q_1 \in \mathbb{R}^2 \end{aligned}$$

and consider the following algorithm:

1. Extend the segments to lines allowing $t, s \in \mathbb{R}$.

2. Find intersection point given by t_0 and s_0, respetively.

3. Check if $s_o, t_0 \in [0, 1]$

This seems straightforward but let us point out some subtle points. Not all pairs of lines intersect at a single point, we might have no intersection point if the lines are parallel or an infinite number of such points if the lines are identical. To find out whether we are in one of these special cases requires us to check the equality of two numbers. If we use some kind of approximate real numbers (floats, double precision reals) in our algorithm, equality tests are always tricky due to rounding. For this reason, in many computational geometry algorithms it might be good, if possible, to use exact arithmetic with rational numbers.

If we use floats to represent numbers also checking the condition $s \in [0, 1]$ might be tricky when one end point of one segment is very close to the other segment. The reason is that a floating point approximation might end up on the wrong side of 0 or 1. Again, using exact rational numbers would take care of this problem.

In any case, all these problems can be dealt with and we conclude that determining whether two segments intersect can be done in time $\mathcal{O}(1)$. Let us turn to a different problem.

1.2 One polygon, one point, is it inside?

Given a polygon p $(p_0, p_1, \ldots, p_{n-1})$ and a point q. Is the point inside or outside of the polygon?

1. Draw a line from q to infinity and check for intersections with p.

2. If odd number of intersections $\Rightarrow q$ is inside of p.
   If even number of intersections $\Rightarrow q$ os outside of p.

Here we need to be careful how we count the intersection with the line from q with p when the line goes through one of the vertices of p. Careless programming might make this count as two intersections (as the line intersects two segments of p) and this is usually incorrect (unless the line is tangent to p at this point). We turn to the main algorithm of the lecture.

1.3 Find closest pair.

Given n points $(p_0, p_1, \ldots, p_{n-1})$, we want to find closest pair. To achieve this we use two standard ideas in computational geometry; a sweep line and a nice data structure and to be more exact we use a balanced binary search tree. We recall that for such a tree we can do the operations

- Insert
- Membership
- Delete
- Next/Previous

in time $\mathcal{O}(\log n)$, the as the height of a balanced search tree is $O(\log n)$.

1.3.1 Algorithm

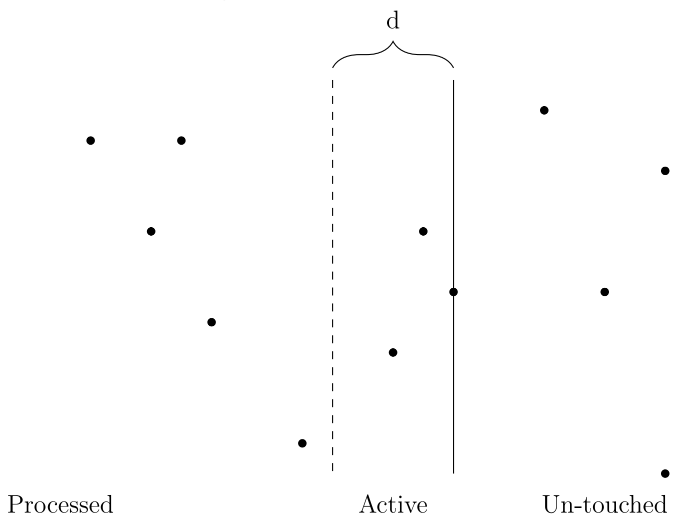
We have a snapshot of a state of the algorithm in Figure 1. The solid line is the sweep line moving to the right. We start by sorting the points according to the x-coordinate and points at any point of the algorithm are divided into three categories, processed, active and un-touched. We also have a number d which is the distance of the two closest points found so far. It is the smallest found distance of any two points that are either active and processed. We need to see if there is a pair of points that are at a closer distance. This must in such case be a distance between an un-touched point and a point that is either un-touched or active. This follows as the distance from the dotted line to the sweep line is d and hence the distance from any processed point to any un-touched point is at least d.

The active points are stored in a balanced binary search tree ordered by y-coordinate. We now sweep the line and update the situation as follows.

1. Move sweep line right to touch next point (x_i, y_i).
2. Let the dotted line follow at distance d (the distance between the closest pairs we've seen so far). Remove points no longer active from the search tree.
3. Insert the new point into the search tree sorted by y-coordinate, and check if any active points is closer then d to (x_i, y_i).

Let us comment on how to do the third item. We need to search within a half-circle from the new point, see figure 2.

Figur 1: Closest pair.



Any point at distance $< d$ of the new point must be within this half-circle. Instead of searching the half-circle heuristically check the points before and after in the search tree sorted by y. This results in searching the rectangle in figure 2.
The speed of this algorithm depends on how many points we need to check at every step, which is the same as how many points can fit inside the half-circle or in our heuristic the rectangle in figure 2. In fact we only need to check ≤ 8, probably fewer.
Proof:
Make 8 squares of size $\frac{d}{2}$ by $\frac{d}{2}$, each square can contain ≤ 1 point, since any two active points are at distance $\geq d$, see figure 2.

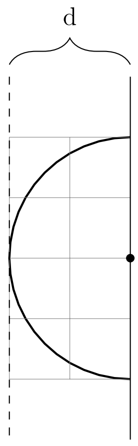
1.3.2 Analysis

The initial sorting of the points in the x-direction requires time $\mathcal{O}(n \log n)$. During the sweep we have n inserts, at most $8n$ previous/next, and n deletes, each operation costing $\mathcal{O}(\log n)$. This implies that the total cost of the algorithm is $\mathcal{O}(n \log n)$.

1.3.3 Is linear time possible?

Probably not and let us give a heuristic motivation. Suppose we want to solve the element distinctness problem, i.e. given n numbers, are they all distinct and suppose we are only allowed comparisons.

Figur 2: Shows that there can be at most 8 points within d from the current point.



After a number of comparisons I know that $x_i \neq x_j \forall i \neq j$. Must I also know which of the x_i and x_j is the largest?
Intuitively obvious (and true) that I must know which is larger, if all information is based on comparisons. This leads to the implication:
Element distinctness $\Rightarrow$ sorting (in compare mode) $\Rightarrow \Omega(n \log n)_{operations}$

Closest pair is harder then element distinctness, as finding the the closest pair of a set of points of the form $(x_i, 0)$ requires to check whether all x_i are distinct. The moral consequence is that, unless we do strange things (hashing etc.) that allows us to check element distinctness in time $O(n)$ we should be happy with $\mathcal{O}(n \log n)$ time complexity for closest pair.