ICS 621: Analysis of Algorithms Spring 2021

Lecture 11

Prof. Nodari Sitchinava *Scribe: In Woo Park, Darlene Agbayani, Michael Rogers*

1 Overview

In the last lecture, we introduced computational geometry and discussed the algorithm for finding convex hull of a set of points in $\mathbb{R}^2$. In this lecture, we introduce two new problems with applications to computational geometry: line segment intersection reporting and Voronoi diagrams. As we will show, both of these problems can be solved efficiently using the *plane sweep* technique.

Algorithm 1

```
1:  SLOW-CH(S)
2:      E ← ∅
3:      for every pair (p, q) in S × S such that p ≠ q
4:          valid ← true
5:          for each r in S such that r ≠ p & r ≠ q
6:              if r is to the left of pq→
7:                  valid ← false
8:          if valid == true
9:              E.add(pq→)
10:     Sort points in E in clockwise order
11:     return E
```

What is the runtime of this algorithm? The outer for loop iterates once for every pair of points. There are $O(n^2)$ such pairs. The inner loop then iterates once for each point, for $O(n)$ iterations. This gives us a total runtime of $O(n^3)$. Sorting the points clockwise can be performed in $O(n^2)$ time, but finding for each segment s_i, the other segment s_j, whose starting point is equal to the ending point of s_i. So the overall runtime of the algorithm is $O(n^3)$. Tractable, but we can do better.

Instead of checking every possible pair of points for inclusion in the convex hull, we can just construct the hull incrementally. Such an algorithm is known as an **incremental construction algorithm**.

Algorithm 2 Computes the upper hull of a set S of points in the plane

```
1:  Upper-CH(S)
2:     E ← ∅                                        ▷ will store a convex hull of points seen so far
3:     Sort points in S by their x-coordinates     ▷ giving us {p_1, p_2, ..., p_n}
4:     E.add(p_1)
5:     E.add(p_2)
6:     for i = 3 to n
7:         E.add(p_i)
8:         k ← E.size
9:         while k ≥ 3 & E[k] is to the left of →(E[k−2], E[k−1])   ▷ last three points make a left turn
10:            E[k−1] ← E[k]                        ▷ remove the next to last point from E
11:            E.size ← E.size − 1
12:            k ← k − 1
13:    return E
```

The above algorithm only returns the upper portion of the convex hull of S, but an algorithm to construct the lower half is symmetric.

So what is the runtime of this algorithm? We have a **for** loop that iterates $O(n)$ times, as well as a while loop that appears to iterate a possible $O(n)$ times as well. This would give a runtime of $O(n^2)$. However, if we realize that each point in S can only be added to E at most once and removed at most once, it becomes clear that the total work of the **for** loop is only $O(n)$. Since the initial sorting of the input points by their x-coordinates takes $O(n \log n)$ time, our total runtime is $O(n \log n)$.

Now that we have an algorithm that runs in $O(n \log n)$ time, can we do even better?

Lemma 1. *The convex hull problem has a lower bound of* $\Omega(n \log n)$*.*

Proof. To show that convex hull takes at least $O(n \log n)$ time, we can reduce the problem of sorting to the convex hull problem using the following linear time algorithm.

Algorithm 3 Sorts A in increasing order via Convex Hull computation

```
1:  CH-Sort(A)
2:     P ← ∅
3:     for i = 1 to n
4:         P.add((A[i], A[i]^2))
5:     C ← Lower-CH(P)
6:     for i = n downto 1
7:         output C[i].x_coord
```

The set of points P forms a parabola, and since a parabola is convex and our convex hull algorithm outputs the clockwise sorted points forming the hull, our inputs are now sorted in ascending order. We know that sorting has a lower bound of $\Omega(n \log n)$. Therefore, the convex hull problem also has lower bound of $\Omega(n \log n)$. ☐

Actually, the fastest known convex hull algorithm takes $O(n \log h)$ time, where h is the number of points in the convex hull (an example of an **output-sensitive algorithm**). Doesn't this contradict our previous proof that a convex hull takes $O(n \log n)$ time to compute? In our reduction above, every point passed into the convex hull algorithm was on the (convex) parabola, and therefore on the convex hull. If we used such a $O(n \log h)$ algorithm to solve sorting, h would be equal to n, and we would still need $\Omega(n \log n)$ time to sort.

2 Line Segment Intersection Reporting

Geographical maps are often made up of multiple layers which are overlapped depending on what type of information should be displayed. For example, a map may store roads in one layer and rivers in another. If the layers are overlapped, intersections of roads and rivers may serve as points of interest. This concept of finding intersections can be further expanded into higher dimensions and even abstract values such as population and weather. However, by considering only one or two dimensions, this problem can be reduced to finding intersections of line segments.

2.1 Problem Statement

Given a set S of n closed segments in a 2D plane, report all intersection points among the segments in S.*

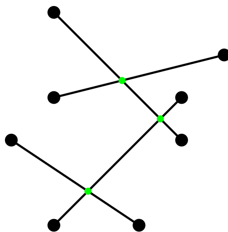


Figure 1: Set of line segments with 3 intersection points (highlighted in green)

A simple way to solve this problem is to take every pair of lines in S and check whether or not they intersect. If there are n segments in S, this algorithm would take $O(n^2)$ time. Furthermore, it could be the case that every pair of segments in S is overlapped in some way. In such a case, we must report $\Omega(n^2)$ intersections anyways. However, it is unlikely that this occurs often, especially in practice, so we would like to find a faster algorithm. The best we can do is to find segment intersections in a way that is dependent on the number of line segments in S as well the number of intersections that must be reported. This type of algorithm falls into a class of algorithms called *output-sensitive algorithms.* If the segments in S contain k intersections, the runtime of an optimal algorithm for finding intersections is given as $O(k + \log n)$. However, we will show a simpler algorithm that solves this problem in $O((k+n) \log n)$ and still demonstrates the plane-sweep technique.

*For simplicity, we assume no degenerate cases such as vertical and collinear segments

2.2 2D Plane Sweep

In order to avoid checking every pair of segments for an intersection, we make an observation: only segments that are neighbors of eachother are candidates for intersection. Let S be an arbitrary set of line segments. First, sort the segments by their y-coordinates. Once sorted, it is easy to check if the segment's y-intervals overlap. Only segments with overlapping y-intervals are considered neighbors and have possible intersections.

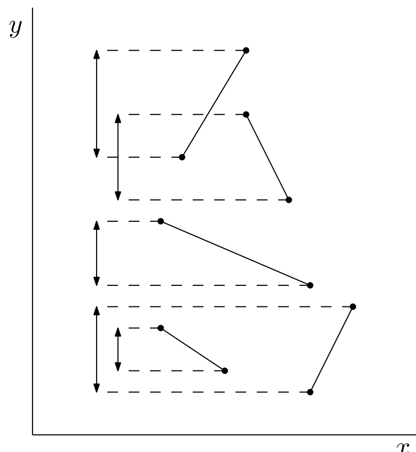


Figure 2: Segments sorted by y-coordinate

After finding the segments with overlapping y-coordinates, we must find which of those actually intersect. Starting above the highest segment, use an imaginary line, ℓ, to sweep downward and keep track of all segments intersecting ℓ at a given time. Segments exist on a continous set of coordinates, but we do not sweep ℓ down continuosly. ℓ only stops at "event" points. More specifically, once ℓ sweeps past and has processed any event point, nothing above ℓ's current position matters.

We define three types of **event points**:

nosep the top endpoint of a segment,

nosep the bottom endpoint of a segment,

nosep intersection points.

If a top endpoint is found, then a new line segment has begun to intersect ℓ, and we must check if it intersects any of the segments that were already intersecting ℓ. If a bottom endpoint is found, then ℓ is no longer intersecting the line segments below ℓ, and we no longer need to keep track of it.

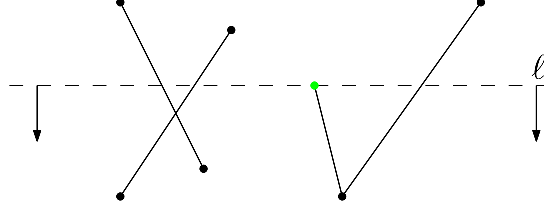


Figure 3: ℓ finding and stopping at an event point (highlighted in green)

Algorithm 4

1: **function** FindIntersection(S)
2: $Q \leftarrow$ MaxPriorityQueue()
3: Insert all segment endpoints e into Q with $key = y$-coordinate of e
4: **while** Q is not empty
5: $e \leftarrow$ Remove(Q)
6: HandleEventPoint(e)

The pseudocode (Algorithm 4) for finding intersections is fairly simple. We maintain a max-priority queue to keep track of segment endpoints in order of y-coordinate. We begin extracting the endpoints (by "sweeping" ℓ) one at a time and updating the set of segments intersecting ℓ. As we mentioned earlier, we must keep track of all segments currently intersecting ℓ. This is done with a binary search tree, which we will call $\mathcal{T}$. $\mathcal{T}$ will be used to keep track of only the segments currently intersecting ℓ and is ordered by the x-coordinate of the segment.

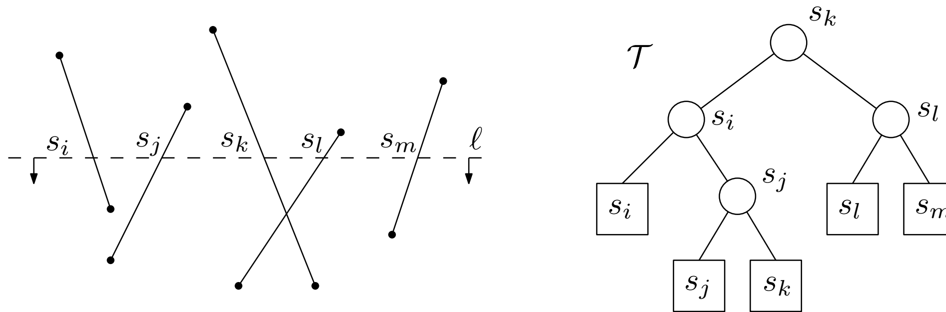


Figure 4: Line sweep with corresponding BST

Notice that if ℓ intersects every segment in S, then we may still need to check all $O(n^2)$ pairs of segments for intersection. However, by ordering all the endpoint events in $\mathcal{T}$ by x-coordinate, only neighboring segments need to be checked for an intersection. A segment's earliest intersection can only occur with its left neighbor or right neighbor. Therefore, processing any event requires checking ≤ 2 neighboring segments in $\mathcal{T}$.

HandleEventPoint does the main work of processing the three types of event points mentioned above. When finding endpoints of segments, we must insert or delete from $\mathcal{T}$. We must also check for intersections when finding a new top endpoint and insert the intersection point into Q to be processed when ℓ passes through it. When an intersection is processed, one segment has crossed with another, and, therefore, their order by x-coordinate must have swapped. This change must

also be reflected in the ordering of $\mathcal{T}$. For inserting and deleting segments as well as segments swapping order in $\mathcal{T}$, we must then check again for intersections with neighbors, since the segments will have new neighbors.

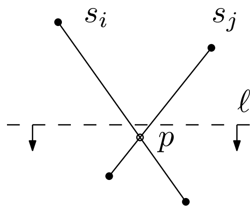


Figure 5: At intersection event p, the ordering of s_i and s_j will switch in regards to x-coordinate

Invariant: All intersections above the sweep line ℓ have been reported.

2.3 Runtime Analysis

In total, there are at most $2n + k$ events in the max-priority queue. If S contains n segments, there are $2n$ endpoints, and k is the number of intersections formed by S. Extracting and processing $2n + k$ events, we incur a cost of $O(\log(2n + k))$ per event. This is logarithmic, since we only need to check a constant number (≤ 2) of segments for intersection, so the overhead is from extraction and insertion from the priority queue and BST. The number of intersections is at most $k \leq n^2$, therefore:

$$\begin{aligned} \log(2n + k) &\leq \log(2n + n^2), \qquad \text{if } k \leq n^2 \\ &= O(\log n). \end{aligned}$$

Therefore, processing a set of n segments with k intersections, takes $O((n + k)\log(n))$ time.

3 Voronoi Diagrams (The Post Office Problem)

Given a set with n distinct points (sites or post offices) $P = \{p_1, \ldots, p_n\}$ in the xy-plane, partition the plane into n cells, one for each $p \in P$, such that a point q lies in cell $\mathcal{V}(p_i)$ if and only p_i is the closest site to q, i.e., $dist(q, p_i) < dist(q, p_j)$ for all $i \neq j$. The set of all cells is called Voronoi diagram. Voronoi diagrams have many applications in social geography, physics, astronomy, and robotics.

Example

Let $P = \{p_1, p_2\}$ be a set of points. The Voronoi diagram of P or Vor(P) is given by dividing the plane into two halves with the *bisector*† line of p_1 and p_2.

†A bisector of a line segment $\overline{p_1 p_2}$ is the perpendicular line that bisects $\overline{p_1 p_2}$.

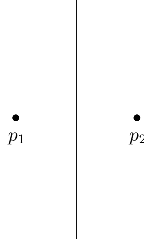


Figure 6: $\text{Vor}(\{p_1, p_2\}) = \{h(p_1, p_2), h(p_2, p_1)\}$.

If we denote the (open) left half plane with $h(p_1, p_2)$ and the right half plane with $h(p_2, p_1)$, then the Voronoi diagram can be written as $\text{Vor}(P) = \{h(p_1, p_2), h(p_2, p_1)\}$. Adding an additional point p_3 into P will change Voronoi diagram of Figure 6 to:

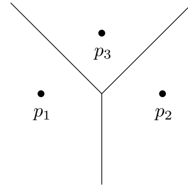


Figure 7: $\text{Vor}(P) = \{h(p_1, p_2) \cap h(p_1, p_3), h(p_2, p_1) \cap h(p_2, p_3), h(p_3, p_1) \cap h(p_3, p_2)\}$.

Note that any point in $\mathcal{V}(p_i)$ has indeed p_i as the closest site, and hence the following fact.

Fact 2. *For $P = \{p_1, \ldots, p_n\}$ we have the Voronoi diagram given by $\mathcal{V}(p_i) = \bigcap_{j \neq i} h(p_i, p_j)$ for $i = 1, \ldots, n$.*

Observe that we used bisectors to define boundaries between half planes, and therefore, the $\mathcal{V}(p_i)$ can be represented as collection of lines, half lines, or line segments. Notice that we only have full lines when all sites in P are collinear, i.e., all points lie in the same line as shown in Figure 8.

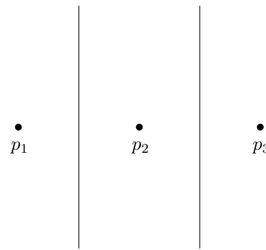


Figure 8: $\mathcal{V}(p_1) = h(p_1, p_2), \quad \mathcal{V}(p_2) = h(p_2, p_1) \cap h(p_2, p_3), \quad \mathcal{V}(p_3) = h(p_3, p_2)$..

If all points are not collinear then there will be no boundaries that are full lines.

Theorem 3. *Let $P = \{p_1, \ldots, p_n\}$ a set of distinct points. If all sites are collinear, then Vor(P) consists of $n - 1$ parallel lines and n cells. Otherwise, Vor(P) is a connected planar graph and its edges are either line segments or half-lines.*

Proof. If all sites are collinear, then we can order the n points based on the line that connects all of them. Hence, there are $n - 1$ parallel bisectors and n cells.

If not all lines are collinear, then for the sake of contradiction, assume e is a boundary that is a full line, i.e., a bisector of p_i and p_j. Since not all points are collinear, there exists a site p_k that is not collinear with p_i and p_j. Both bisectors of $\overline{p_k p_i}$ and $\overline{p_k p_j}$ will intersect e, and therefore, some parts of e will be closer to p_k than p_i or p_j. Hence, e is not a boundary of either $\mathcal{V}(p_i)$ or $\mathcal{V}(p_j)$, i.e., isn't a full line – a contradiction. The connectedness follows from the fact that there is no edge e that is a full line. See Figure 9 below:

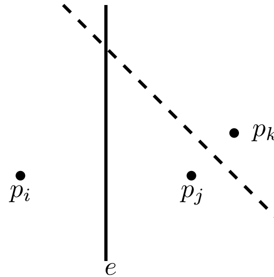


Figure 9: Non-collinear points.

□

3.1 Complexity of Voronoi diagram

Each site in a Voronoi diagram can have at most $n-1$ vertices and edges. Hence, at most, the number of vertices and edges is quadratic. The next theorem shows that the complexity in terms of edges and vertices of Voronoi diagram is actually linear.

Theorem 4. *Vor(P) has at most $2n-5$ vertices and at most $3n-6$ edges.*

Proof. We use Euler's formula of a connected planar graph which states the following:

$$\#\text{ of vertices } - \#\text{ of edges } + \#\text{ of faces} = 2.$$

Since a planar graph cannot contain half-lines and Vor(P) can have half-lines, we add one additional vertex v_∞ that terminates ends of all loose half-lines. Observing Figure 7, we know that each vertex has degree is at least 3, and therefore, summing up the degrees of all vertices should be less than twice the number of edges i.e. $2n_e \geq 3(n_v+1)$.

$$\begin{aligned}
(n_v+1) - n_e + n_f &= 2 \\
(n_v+1) - n_e + n &= 2 \\
(n_v+1) - \frac{3}{2}(n_v+1) + n &\geq 2 \\
2(n_v+1) - 3(n_v+1) + 2n &\geq 4 \\
2n-5 &\geq n_v && (\text{ inequality for vertices}) \\
(2n-5) + 1 - n_e + n &\geq 2 \\
3n-6 &\geq n_e. && (\text{ inequality for edges})
\end{aligned}$$

□

Though we have estimated quadratic number of bisectors, we are restricted to only a linear number of edges and vertices in a Voronoi diagram. Hence not all bisectors are in $\mathcal{V}(P)$ diagram, and not all intersections of bisectors appear in $\mathcal{V}(P)$.

One way to compute the Voronoi diagram is to use Fact 2, and obtain all intersections of bisectors which takes $O(n \log n)$ (see previous section). Since we have n points, the total cost is $O(n^2 \log n)$. The optimal solution for Voronoi diagram utilizes a plane sweep technique effectively to compute Voronoi diagram in $O(n \log n)$ time by avoiding unnecessary computations.

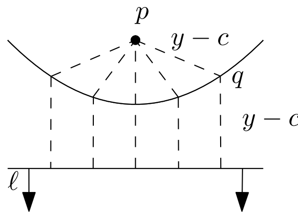
3.2 Fortune's algorithm to compute Voronoi diagram

The intuition of this algorithm lies on the the idea of a sweep line coming from $y = +\infty$ downwards. Whenever the sweep line ℓ intersects an event point, then some information about the structure is maintained and small amount of information needs to be processed. For the Voronoi diagram, this information is partial about the cells, because a cell might still depend on an event point below the sweeping line. However, a cell, that is above the line and at a farther distance from sites below ℓ than ℓ itself, would no longer be affected by the sweeping process. Hence, we have an invariant that can be used to justify that the sweeping process does compute the Voronoi diagram.

Invariant A cell $\mathcal{V}_\ell(p)$ for a point p above ℓ (or $p \in \ell^+$) does not change if for any point $q \in \mathcal{V}_\ell(p_i)$ we have $dist(q, p_i) < dist(q, p')$ for all p' below ℓ (or $p' \in \ell^-$). Hence, $\mathcal{V}(p) = \mathcal{V}_\ell(p)$.

The distance from any point $q \in \ell^+$ to the horizontal line ℓ at $y = c$ depends on the y-coordinate of q i.e. $dist(q, \ell) = y_q - c$. In order for q to lie at the boundary distance between p and ℓ, we should have $dist(p, q) = y_q - c$. The following analysis shows that the boundary between ℓ and p constructs a parabola.

$$dist(q,p)^2 = (x_q - x_p)^2 + ((y_q - c) - y_p)^2 = (y_q - c)^2$$
$$(x_q - x_p)^2 + (y_q - c)^2 - 2(y_q - c)y_p + y_p^2 = (y_q - c)^2$$
$$\frac{(x_q - x_p)^2 + 2cy_p + y_p^2}{2y_p} = y_q \qquad \text{(equation of a parabola)}$$



Combining these parabolas of multiple sites above ℓ constructs a sequence of parabolic arcs which we call a *beach line* as shown in Figure 10.

The beach line is the structure that we will maintain while sweeping down. The question remains at what event points shall we process the beach line structure? The answer is when we construct a new arc after the sweeping line reaches a new site below ℓ, and when an arc disappears. We will call such an event a *site event.*

Lemma 5. *The only way in which a new arc can appear on the beach line is through a site event.*

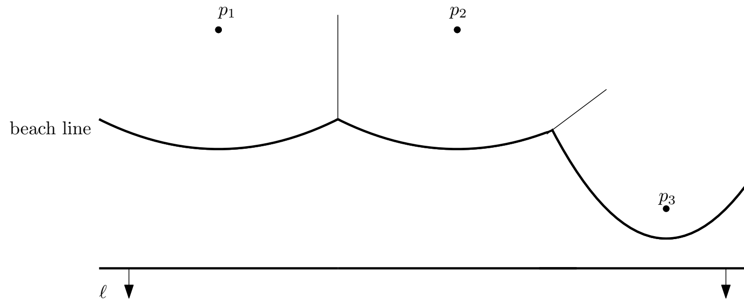
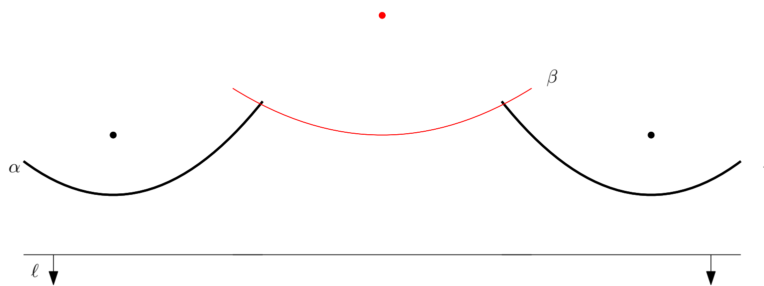


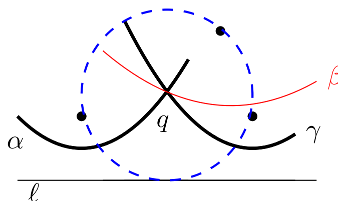
Figure 10: An example of a beach line.

Proof. The intuition behind the proof is that there is no parabola arc that breaks through the beach line. If there exists such arc β, then either β breaks through a bigger arc β', essentially touching it at one point before going deeper down. However that can never happen as β can only intersect β' at two locations which means β never breaks through β.[‡]

The other case β is breaking through two arcs α and γ with all sites above line ℓ as shown in the Figure below:



For this to happen, β arc needs to touch the intersection point q of α and γ before going deeper down. However, this creates a scenario where q is at the centre of a circle that touches the sites of α, β and γ, and also the line ℓ (equidistant). When such alignment of three arcs happen at a circle, we call such event a *circle event*.



When ℓ takes a slight step down, a bigger circle is needed to accommodate all previous sites, but that can never happen, as one site will be an interior point to the big circle, which means it has a closer distance to ℓ than β site which means β never break through the beach line. □

Circle event in the proof above, is when one arc β disappears from the beach line. Such an event is important while processing the structure of beach line during sweeping. The next Lemma shows the only way an arc in the beach line disappears.

[‡]Details of the algebraic proof are in chapter 7 page 151 in the Computational Geometry book [].

Lemma 6. *The only way in which an existing arc can disappear from a beach line is through a circle event.*

Since we know the event points to either a new site, or a circle event, we are left with the task of determining the data structure that maintains the information about the beach line and only small part of that structure is accessed at each event point. As we did with line segment intersections in the previous section, we maintain a priority queue for event points along with a BST to hold the structure. Since both of these structures are balanced trees, we can predict the processing time to be $O(n \log n)$.