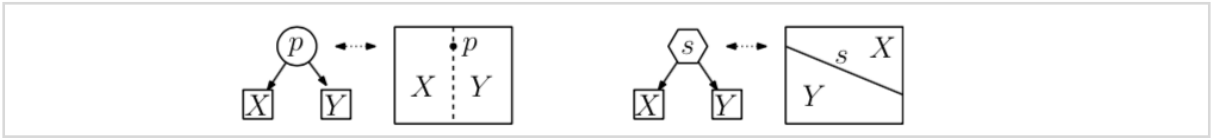
CS 6301.002.20S Lecture 10—February 13, 2020

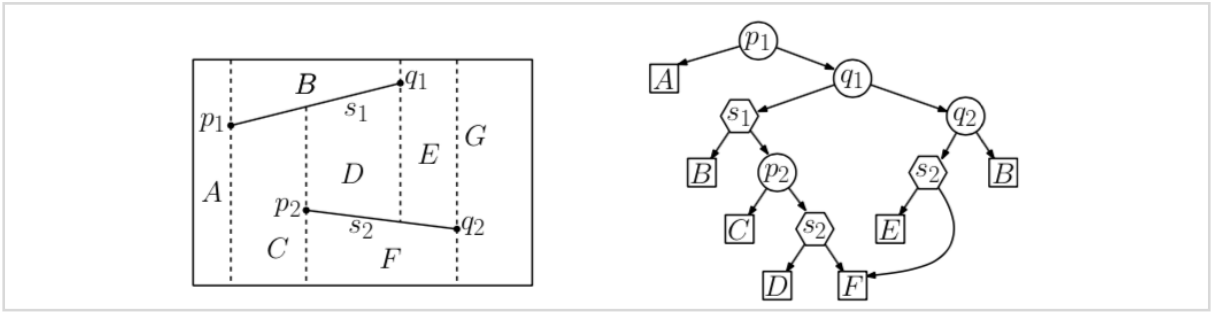
Main topics are #planar_point_location.

Point Location

- Let's finish discussing the data structure for planar point location.
- We'll still assume we're given segments S = {s_1, …, s_n}.
- We'll use a directed acyclic graph with a single root / source node where each leaf / sink represents a trapezoid of S's trapezoidal decomposition.
- There are two types of nodes:
  - *x-nodes* reference an endpoint p from one of the segments. They have two outgoing edges/children corresponding to points lying left or right of the vertical line through p.
  - *y-nodes* reference an input segment and their left and right outgoing edges/children correspond to points above or below the segment's line, respectively.



- To perform a query, you simply start at the unique source node / root and follow the correct child from each node until you hit a sink / leaf.
- We build the data structure so that each sink corresponds to a trapezoid in the trapezoidal map.



- Query time is proportional to the distance you must travel to reach a leaf.

Construction

- To build the trapezoidal map, we used randomized incremental construction.
- We added segments one by one in random order. Let S_i be the first i segments and T_i be their map.
- To add segment s to S_{i-1}, we had to do a point location query to find s's left endpoint. Then we added extensions for its endpoints and trimmed back the walls from trapezoids of T_{i-1}, creating a bunch of new trapezoids.

- We said these new trapezoids *depend* upon s.
- So here's the trick: since we need to do point location to insert each segment, we should incrementally build and query the point location data structure as well!
- Adding the segment s destroys some trapezoids and adds some new ones. What we did is replace each destroyed trapezoids' leaves with a constant number of nodes leading to newly created trapezoids' leaves.
- Each new trapezoid appears as a leaf exactly once to keep the space low.

Analysis

- The construction of the trapezoidal map and point location data structure are randomized. In the latter case, the structure itself is random, meaning both its size and the time for queries are random.
- So we'll try to figure out their expected values.
- For size, recall that we added a constant number of nodes every time we destroy a trapezoid. There are O(n) trapezoids created in expectation across the whole algorithm, so the expected size of the point location structure is O(n).
- Query time is more complicated. Say we're given some arbitrary query point q. This point is not random and may be chosen to make this analysis seem as bad as possible. I claim the expected search depth in the DAG for q is O(log n) where the expectation is taken over all random orderings of the line segments.
- Note this analysis just works for any one point q picked independently of the random choices. As far as this analysis is concerned, there may be *some* point q' with bad query time. But to find q', you'd have to know the random choices. I'll come back to this point later.
- OK, so suppose we search for q. Remember how the structure is built top down as we add new segments. You can imagine the location of q moving from top to bottom as well.
- Before adding any segments, q lies at the root. As we add each segment, q moves down at most three levels to reach its new leaf trapezoid.
- Consider the example above. Say q lies somewhere in final trapezoid D. q starts somewhere in the box. Oh, but segment s_1 was added so it moves three steps down to a leaf where that p_2 is. Oh but segment s_2 was added so it moves two steps down to the leaf for D.
- So, the search depth for q is proportional to the number of times q's trapezoid changes.
- Let X_i be a random variable equal to 1 if q changes its trapezoid after the ith insertion and 0 otherwise.
- Let D(q) denote the depth of q in the final search structure.
- D(q) ≤ 3 sum_{i=1}^n X_i, so E[D(q)] ≤ 3 sum_{i=1}^n E[X_i].
- We're again summing over the probabilities that each variable is 1.

- But how do we analyze that? Well, we already did in a way on Tuesday.
- Fix some S_i. Consider the trapezoid Delta_i in T_i containing q. This trapezoid depended on at most four segments, each of which has a 1/i probability of being the last segment added from S_i. So, the probability we created Delta_i with the last insertion is ≤ 4/i.
- E[D(q)] ≤ 3 sum_{i=1}^n E[X_i] ≤ 3 sum_{i=1}^n 4/i = 12 sum_{i=1}^n 1/i.
- This last summation is, by definition, equal to H_n, the nth member of the Harmonic series which grows asymptotically with Theta(log n).
- E[D(q)] = O(log n).
- So, given an arbitrary query point q, *chosen independently of the random segment order*, we expect the query time to be O(log n).
- This bound applies for individual segment endpoints added during map construction, but here the expectation is taken over the map for the *previously* added i - 1 segments. The total expected time spent building the trapezoidal map and data structure including both point location and creating new trapezoids is O(1) + sum_{i=1}^n {O(log i) + O(1)} = O(n log n).

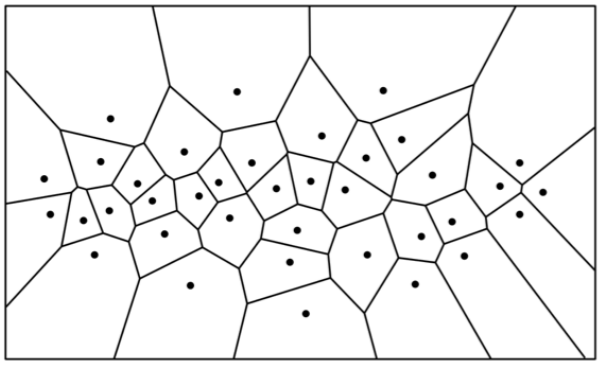
Guarantees on Search Time

- As far as we can tell, though, there may be *some* query point q for which a search takes much longer than O(log n).
- It turns out we *can* guarantee any search takes O(log n) time, but it requires a more complicated analysis.
- Lemma: Fix some value lambda > 0. The *maximum* search depth exceeds 3 lambda ln(n + 1) with probability at most 2 / (n + 1)^(lambda ln 1.25 - 3).
- So, for example, the probability any search path has more than 60 ln(n+1) nodes is at most 2/(n + 1)^1.5. That gets very small very quickly as n grows.
- You can argue a similar bound for the size of the data structure.
- And with this strong of bounds, you can even make a data structure that has good *worst-case* size and query time.
- Just run the construction algorithm tracking the depth and size of the data structure as you go. Restart if either get too large. With good probability, you'll only need to try a constant number of times. So the O(n log n) construction time is still in expectation, but the size and query time are worst-case guaranteed.

Voronoi Diagrams

- Time to start another problem!
- Let P = {p_1, …, p_n} be a set of n points in R^d we call *sites*.
- Let ||pq| = sqrt((p_x - q_x)^2 + (p_y - q_y)^2) be the Euclidean distance between p and q.

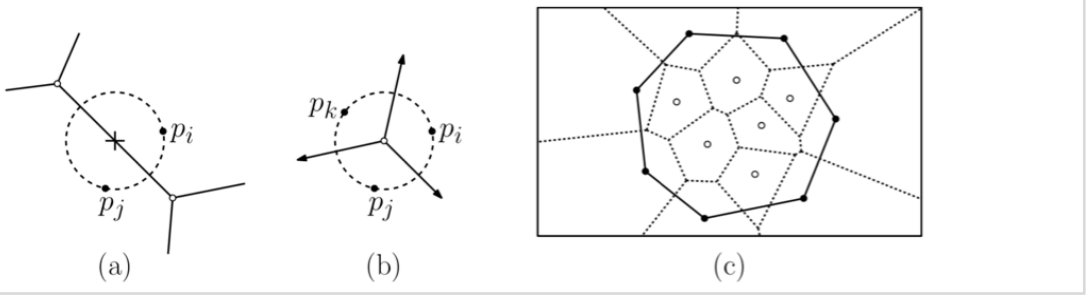
- The Voronoi cell of site p_i, denote V(p_i) is the set of points closer to p_i than any other site.
  - V(p_i) = {q in R^2 : ||p_i q|| < ||p_j q||, for all j ≠ i}
- The union of the closure of the Voronoi cells forms the *Voronoi diagram*.



- The cells are (possibly unbounded) convex polyhedra, since V(p_i) is the intersection of all the halfspaces for points closer to p_i than each other point
- Voronoi diagrams have *a lot* of uses, including
  - nearest neighbor queries: to find the nearest neighbor of a point q, just do point location in the Voronoi diagram
  - shape analysis: we can get a useful sketch of a polygonal shape called the media axis if we compute the Voronoi diagram of its vertices and edges
  - center-based clustering: we want to partition a set P into subsets of points that are close together. If we choose some centers for these subsets, then the Voronoi diagram over the centers tells us which points belong in each cluster

Properties

- Voronoi diagrams have several nice properties that make them useful and will be useful in their construction

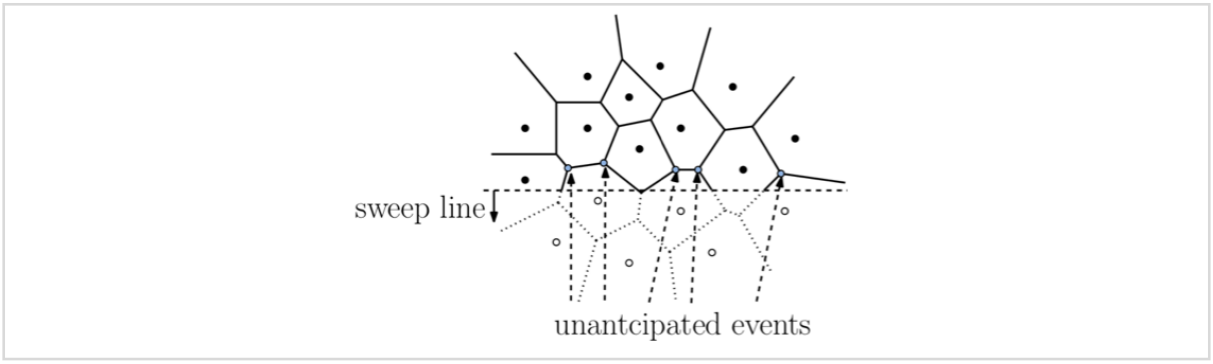


- Empty circle property: Each point on an edge of the Voronoi diagram is equidistant from its nearest neighbors. Therefore, there is a circle centered at that point through the neighbors with no other site interior to the circle.
- Voronoi vertices: Each Voronoi vertex is equidistant to three sites. Therefore, there is a circle centered at the vertex, passing through the three sites, with no other vertex interior.
- Assuming no four sites are cocircular, each vertex has degree 3.

- The Voronoi diagram has n faces, roughly 2n vertices, and roughly 3n edges. The book has a proof.

Computing the Voronoi Diagram

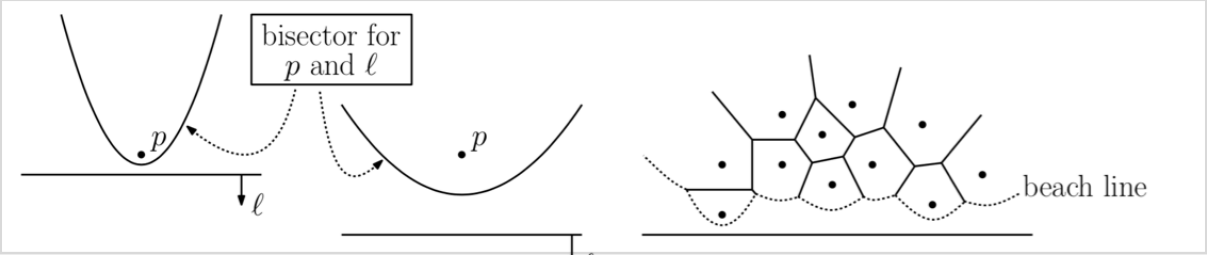
- We could compute the Voronoi diagram in O(n^2 log n) time by solving n halfspace intersection problems, one per site.
- Instead, I'll give an O(n log n) time sweep line algorithm due to Fortune ['87].
- There's also a randomized incremental construction algorithm. You should see a version of that next week when we discuss a related problem.
- So for this algorithm, I'm going to sweep top-down. Partly to match Mount's notes. Partly because certain drawings will be nicer that way.
- Now, it's temping to try designing the algorithm in the following way: sweep from top to bottom and maintain the whole Voronoi diagram from infinity down to the sweep line. Like in line segment intersection, we'll try to discover Voronoi vertices before we reach them and add them to the event queue if necessary.
- But it's not that simple:



- Sites we haven't reached yet will create Voronoi vertices we've already passed!
- So instead, we'll accept that we haven't computed everything up to the sweep line. Instead, we'll have computed everything up to an x-monotone curve called the *beach line* that lags behind the sweep line and essentially forms the boundary of what we know so far.
- More formally, the sweep line divides the plane into two halves, the stuff above it that we've swept already and the stuff below.
- We'll treat the sweep line as another infinitely long site. The beach line is just the boundary of its site: those points equidistant from their nearest neighbor above the line and the line itself.
- Anything we've computed strictly above the beach line belongs to the final Voronoi diagram: after all, the sweep line is already closer than any site that lies below it.
- OK, so what does the beach line look like and why do we call it a beach line?
- Given the sweep line ell and a site p, the points equidistant from both form an x-monotone *parabola*. As ell moves downward, the parabola gets fatter. In contrast, the parabola is a

vertical ray shooting up if p lies on ell.

- The beach line is the lower envelope of the parabolas for all the sites, so its composed of several parabolic arcs.



- The arcs intersect at *breakpoints* which are equidistance between the arcs' points and ell. Meaning breakpoints lie on Voronoi edges.
- Our goal is to simulate the movement of the beach line as the sweep line moves downward. Breakpoints will trace the path of the Voronoi edges.