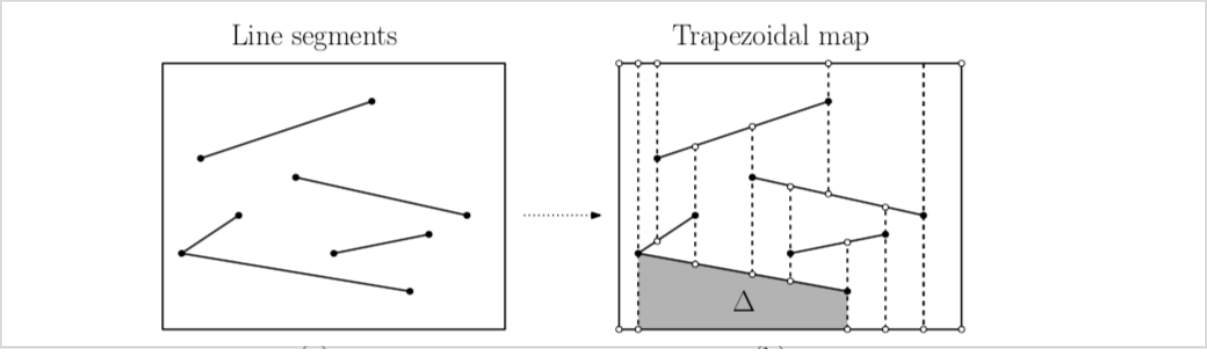
CS 6301.002.20S Lecture 9—February 11, 2020

Main topics are #trapezoidal_maps and #planar_point_location.

Trapezoidal Maps

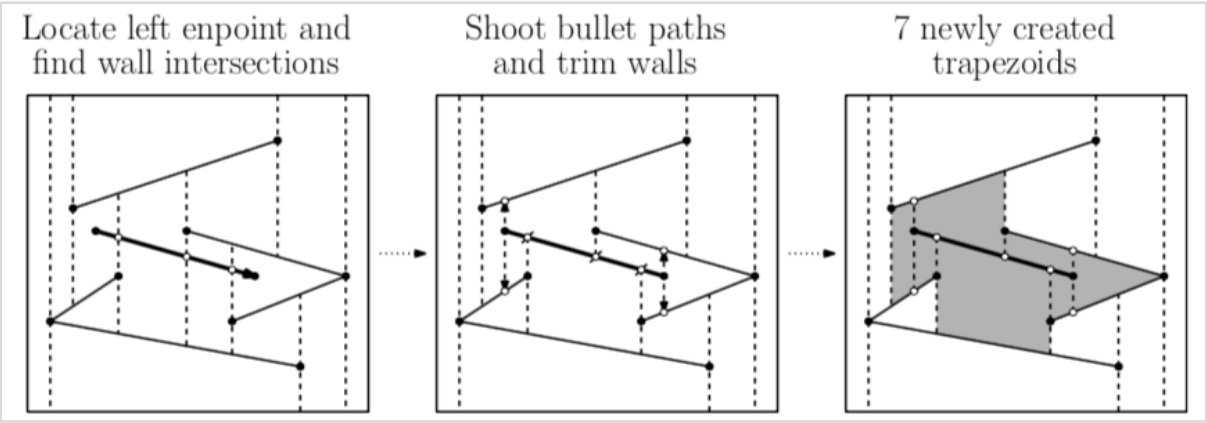
- Let S = {s_1, …, s_n} be a set of line segments that do not intersect except possibly their endpoints.
- We'll assume all *distinct* segment endpoints have distinct x-coordinates.



- We'll build a subdivision of space that respects the line segments.
- Start by adding a big bounding rectangle.
- Next, we send two *vertical extensions* or *bullet paths* from each endpoint. They go until they run into another segment or the boundary of the box.
- The combination of original segments and vertical extensions form a *trapezoidal map* also known as a *trapezoidal decomposition*.
- Generally, the faces of the map look like trapezoids with vertical sides called *walls*.

Constructing the Map

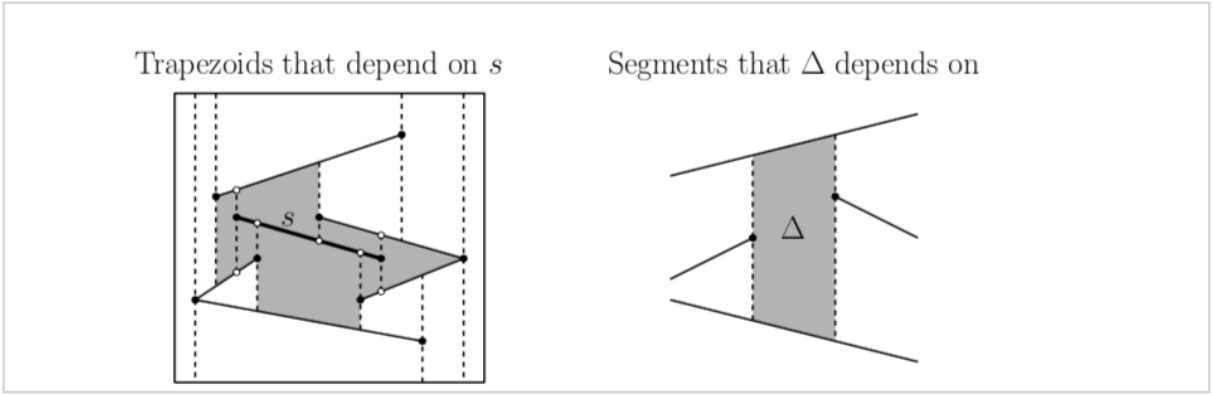
- We could use a straightforward plane sweep approach to construct the trapezoidal map.
- But instead, we'll do randomized incremental construction like we did for linear programming. Doing so will help with point location later.
- We'll start with a single trapezoid, the bounding box. Then we'll add segments one-by-one. Let S_i be the first i segments, and let T_i be the trapezoidal map for S_i.
- When we add segment s_i to S_{i-1}, we figure out which trapezoid of T_{i-1} contains the left endpoint. We'll discuss how to find that trapezoid later when we do point location.
- Next, we walk along the segment from left to right, taking note of which trapezoids we pass through.
- Finally, we fix up all the trapezoids we walked through by
  - firing vertical extensions from the left and right endpoints for segment s_i
  - trimming back each vertical extension of T_{i-1} crossed by s_i
- Doing these steps creates some new trapezoids that did not exist before.



- Let k_i be the number of newly created trapezoids. Ignoring the time taken to locate the left endpoint, this operation takes O(k_i) time. In short, we add 4 new segments and trim back k_i - 4 walls. We can use a suitable representation of the trapezoidal map like a DCEL to add segments or trim back walls in O(1) time each.

Analysis

- Now let's analyze the time it takes to build the map.
- In the worst case, each additional segment we add could result in us trimming back Omega(n) walls, leading to an Omega(n^2) running time.
- However, by picking the order of the segments uniformly at random, it turns out we'll trim back only O(1) walls per insertion in expectation.
- Later, we'll show how to do all the point locations in O(n log n) total expected time. So in total, we'll spend O(n log n) expected time building the trapezoidal decomposition.
- So, for now it suffices to count the total number of new trapezoids created with each insertion.
- Lemma: Let k_i be the number of new trapezoids created when segment i is added. E[k_i] = O(1).
- Like for linear programming, we'll use backwards analysis to get the result.
- Proof:
  - Let T_i be the trapezoidal map for S_i.
  - Each segment of S_i had a 1/i probability of being added last.
  - Let's count how many trapezoids were created when we added s_i to S_{i-1}.
  - Say trapezoid Delta in T_i *depends* on a segment s if adding s as the last segment would have caused Delta to be created.

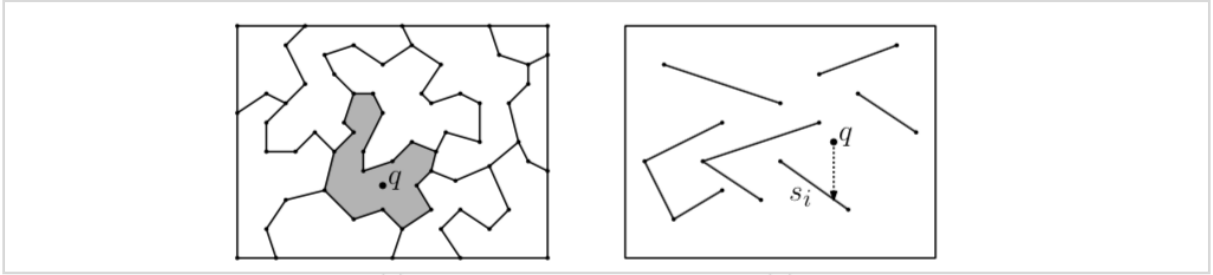


- Let delta(Delta, s) = 1 if Delta depends on s and 0 otherwise.
- E[k_i]
  - = 1/i sum_{s in S_i} (# of trapezoids that depend on s)
  - = 1/i sum_{s in S_i} sum_{Delta in T_i} delta(Delta, s)
- It's hard to get a handle on this sum of delta(Delta, s)s, because some segments intersect a lot of trapezoids and others intersect very few.
- So why don't we just reverse the order of the summation?
- E[k_i] = 1/i sum_{Delta in T_i} {s in S_i} delta(Delta, s)
- Each trapezoid Delta is bound by four sides, the top and bottom are bound by distinct segments s for which delta(Delta, s) = 1. The right and left side contain a single segment endpoint, for which delta(Delta, s) may be 1. Other segments don't matter (and if multiple segments share an endpoint, then none of them could have been the one to introduce that endpoint if added last).
- So, sum_{s in S_i} delta(Delta, s) ≤ 4.
- E[k_i] ≤ 1/i sum_{Delta in T_i} 4 = 4/i | T_i| ≤ (4/i) * (3i + 1) = O(1).

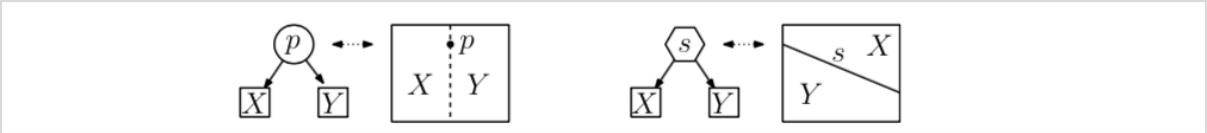
- The total number of trapezoids added (and maybe destroyed) across the entire algorithm is n * O(1) = O(n) by linearity of expectation.

Point Location

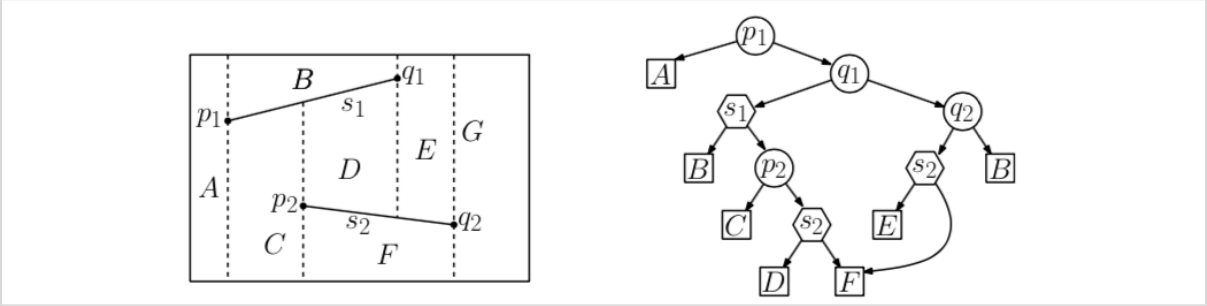
- With the remaining time, let's discuss the data structure for planar point location.
- We'll still assume we're given segments S = {s_1, …, s_n}.
- Assuming those segments form a planar subdivision, it's natural to ask, given a query point q, which face it lies in.
- But instead we'll ask more general *vertical ray-shooting queries*. Given q, which line segment s_i lies immediately below it?



- When searching for something, you might be used to the idea of searching a rooted tree. For point location, we'll instead use a directed acyclic graph. Meaning a directed graph with no *directed* cycles.
- 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.
- So here's where things come together. We can build the data structure so that each sink corresponds to a trapezoid in the trapezoidal map.

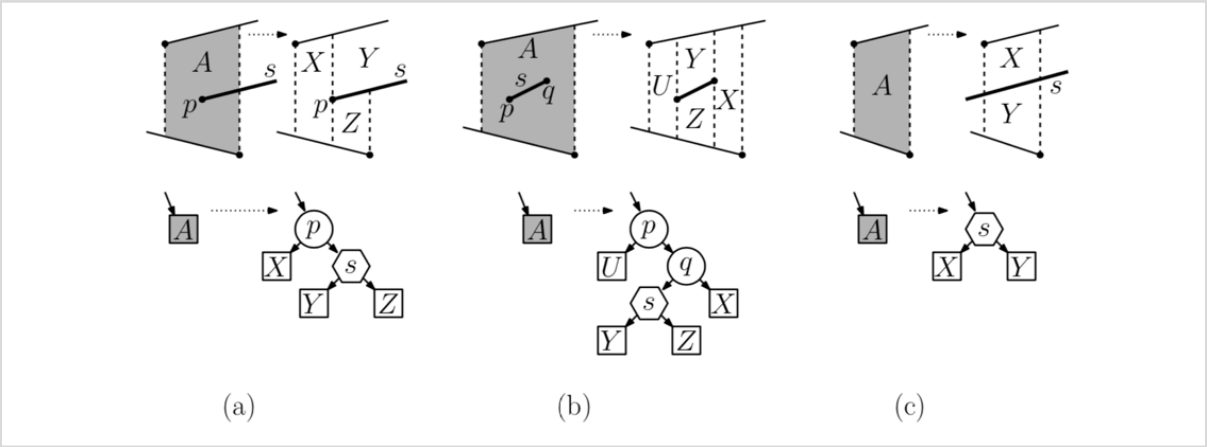


- Query time is proportional to the distance you must travel to reach a leaf.

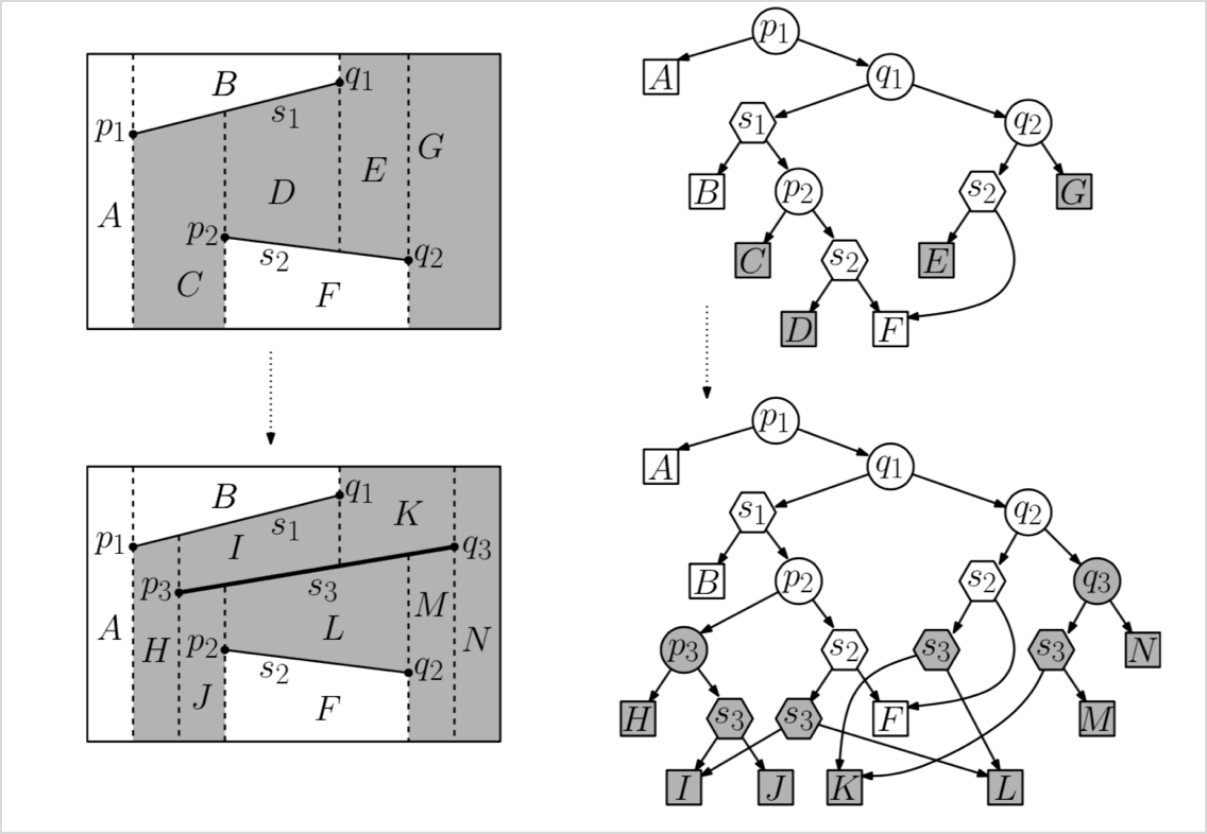
Construction

- 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'll do is replace the destroyed trapezoids' leaves with a handful of nodes leading to each newly created trapezoid.
- There's a couple cases to consider when removing an old trapezoid depending on how

many endpoints of new segment s lie inside it:



- (a) one endpoint:
  - trapezoid A is replaced by three trapezoids X, Y, and Z surrounding endpoint p in clockwise order.
  - Create an x-node for p with left child X
  - Right child is a y-node for s with children Y and Z.
- (b) two endpoints:
  - trapezoid A is replaced by four trapezoids U on left, X on right, Y above, Z below.
  - Create x nodes for p and q. U lies to left of both and X lies to right of both.
  - Create a y node for s between them, Y and Z are its children.
- (c) no endpoints:
  - trapezoid A is replaced by two trapezoids, X above and Y below
  - Create y-node for s with children X and Y.
- Now, you might notice that we create leaves for the same new trapezoid multiple times. To keep the space usage low, we'll create just one sink/leaf for that trapezoid that gets shared between all the structures that want to point to it. We still create multiple y-nodes for s, though, since the path to those nodes provides valuable information.
- Here's an example of what happens when you add a whole segment:



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, notice that we add a constant number of nodes every time we destroy or create a trapezoid. There are O(n) trapezoids created in expectation across the whole algorithm, so the expected size 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 above the root. As we add each new 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 I. 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. Oh, but s_3 was added so it moves another step to the leaf for I.
- 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 where the expectation is taken over the map for the previously added segments, meaning the total expected time spent building the trapezoidal map and data structure 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 get 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.