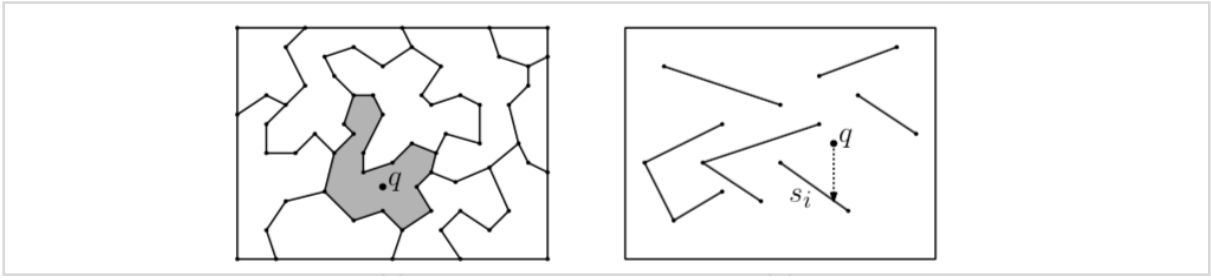
CS 6301.008.18S Lecture—February 8, 2017

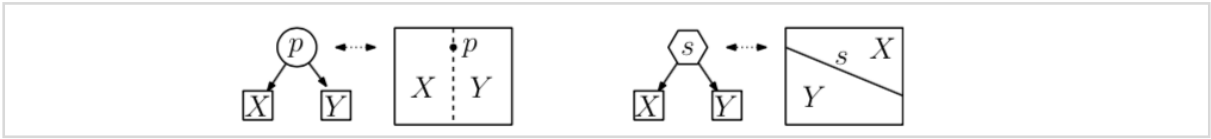
Main topics are #trapezoidal_maps, and #planar_point_location.

Planar Point Location

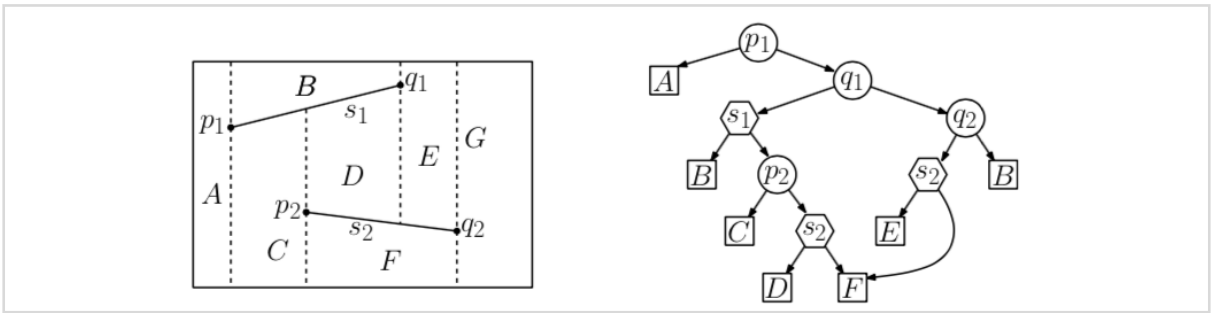
- Tuesday, we started to discuss the problem of planar point location.
- We're given segments S = {s_1, …, s_n}.
- Assuming those segments form a planar subdivision, it's natural to ask, given a query point q, which face it lies in.
- But instead we'll ask more general *vertical ray-shooting queries*.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



- For point location, we'll build a directed acyclic graph. Meaning a directed graph with no *directed* cycles.
- There are two types of internal nodes:
  - *x-nodes* contain an endpoint p from one of the segments. It has two outgoing edges/ children corresponding to points lying left or right of the vertical line through p.
  - *y-nodes* contain a pointer to an input segment and it's left and right outgoing edges/ children correspond to points above or below the segment's line, respectively.



- The leaves of the data structure correspond to trapezoids in a trapezoidal map.

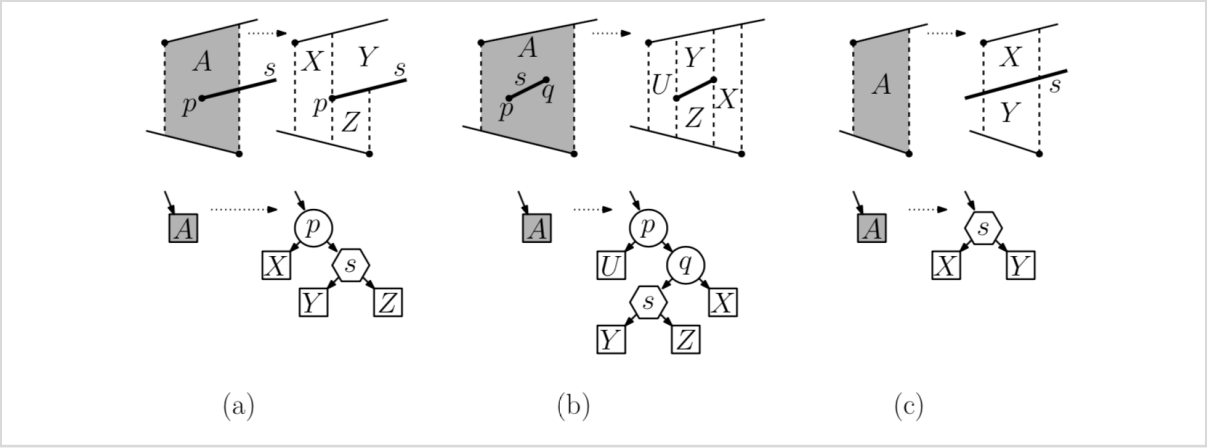


- Recall, for a trapezoidal map we sent vertical extensions from each segment endpoint, partitioning the plane into a collection of internally disjoint trapezoids.
- To perform a query, you simply start at the unique source node / root and follow the correct child from each node until you hit a sink / leaf.

- Query time is proportional to the distance you must travel to reach a leaf.
- So how do we build a DAG that allows for small query times?

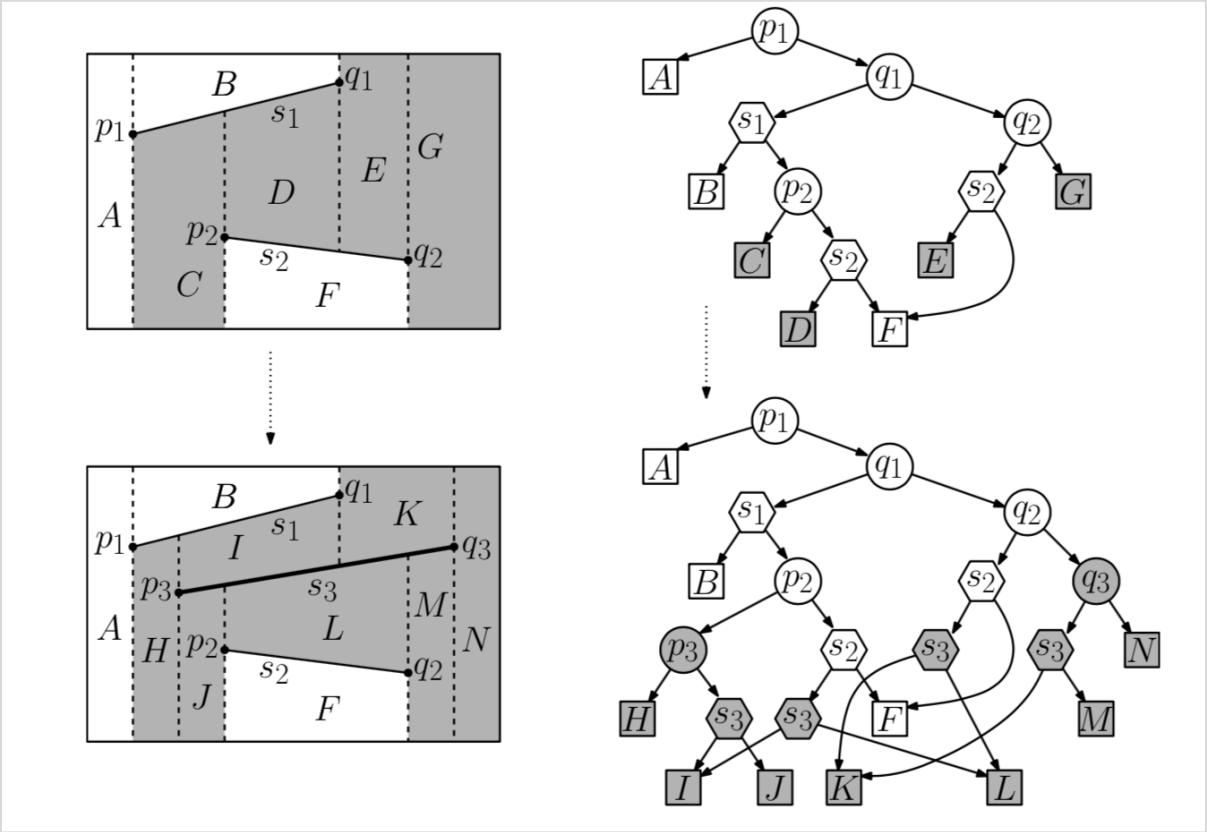
Construction

- 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. On Tuesday, I showed you 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, meaning the total 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.
- Don't worry, I won't ask you to do anything like this in the homework!
- Lemma: Fix a query point q and some value lambda > 0. The search depth exceeds 3 lambda ln(n + 1) with probability at most 1 / (n + 1)^(lambda ln 1.25 - 1).

- Earlier, we defined these random variables X_i and added them up. That was fine for computing an expectation. However, the events leading to these random variables are not independent, so we cannot easily learn the probability that few of them occur.
- So our first priority is to define some random variables that are independent that we can use instead.
- To do that, I'm going to play a weird trick. Let G be a directed acyclic graph. Its nodes are subsets of S. There is an arc from one node to another if the second contains exactly one more line segment. Therefore, it has one source emptyset and one sink S.
- Call the subsets of size i the ith layer of the DAG.
- Directed paths in G correspond to permutations of S, we keep adding segments one at a time as we walk along the path. These correspond to executions of the map building algorithm.
- Consider an arc from node S' to node S''; it represents inserting some segment s into S'. **mark** the arc if inserting it changes the trapezoid that q lies in.
- We already established that each node has at most four marked incoming arcs.
- Some nodes may actually have fewer than four marked incoming arcs. To make the next step work out, we'll go ahead and mark a couple others so every node has exactly four. For nodes in the bottom couple layers, we'll mark every incoming arc. The important thing is that every node in a single layer has the same number of marked incoming nodes and that this number is at most 4.
- A random permutation corresponds to a random source to sink path in our DAG. Let X_i = 1 if the ith arc on the path is marked and 0 otherwise.
- Every arc going into the ith layer is equally likely to be on the path. There are exactly i incoming arcs per node and at most 4 per node are marked, so E[X_i] ≤ 4/i.
- Remember how we marked the same number of arcs per node, though? That means even if you know X_i, you don't know which node lies on the path. Meaning you know nothing about the other random variables. They are independent!
- Alright, time for some algebra to take advantage of this independence.
- Let Y = sum_{i=1}^n X_i. The number of nodes on the search path for q is at most 3Y, so we want to know the probability that Y > lambda ln(n + 1).
- We'll use something called Markov's inequality: given a nonnegative random variable Z, Prob[X ≥ alpha] ≤ E[Z] / alpha.
- So for any t > 0, Prob[Y ≥ lambda ln(n + 1)]
  - = Prob[e^{tY} ≥ e^{t lambda ln(n + 1)}]
  - ≤ e^{-t lambda ln(n+1)} E[e^{tY}]
- Our variables are independent so E[e^{tY}]
  - = E[e^{sum_i t X_i}]
  - = E[prod_i e^{t X_i}]
  - = prod_i E[e^{t X_i}]

- Set t = ln 1.25. We have E[e^{t X_i}]
  - ≤ (4/i) e^t + (1 - 4/i) e^0 [by def. of expectation]
  - = (4/i) (1 + 1/4) + 1 - 4/i
  - = 1/i + 1
  - = (1+i)/i
- And thus prod_{i = 1}^n E[e^{t X_i}]
  - ≤ (2/1) (3/2) … (n + 1 / n)
  - = n + 1.
- All together, Prob[Y ≥ lambda ln(n + 1)]
  - ≤ e^{-lambda ln(1.25) ln(n + 1)} (n + 1)
  - = (n + 1) / (n + 1)^{lambda ln(1.25)}
  - = 1 / (n + 1)^{lambda ln(1.25) - 1}

- Alright, so we have a bound on the probability that any one point has a long search path.
- 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 imagine we partition the plane into vertical slabs by passing a vertical line through every segment endpoint. The intersection of these slabs and the n segments further partitions the plane into at most 2(n+1)^2 trapezoids.
- Any two points in one of these trapezoids must take the same path through the search structure.
- So take an arbitrary set of 2(n+1)^2 query points, one per trapezoid.
- If *they* all have short search paths, then every point has a short search path.
- And the probability that any one of them has a long search path is at most 2(n+2)^2 / (n + 1)^(lambda ln 1.25 - 1) = 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.