CS 6301.008.18S Lecture–February 6, 2017

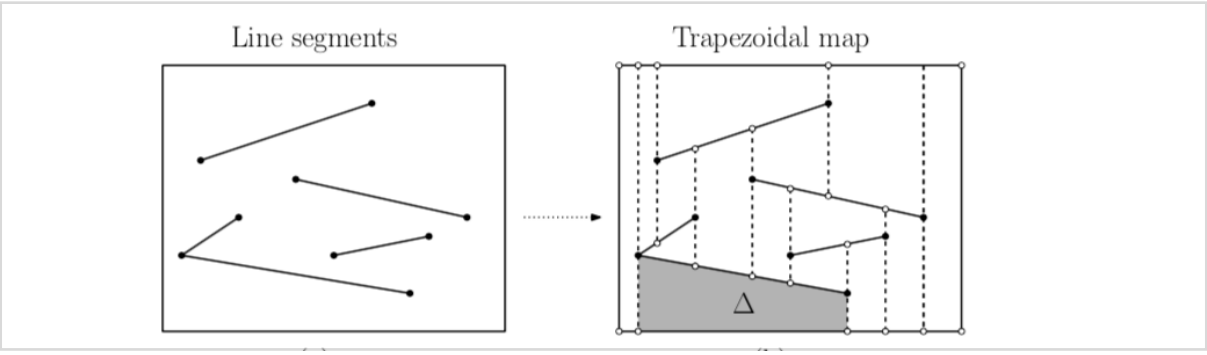
Main topics are #trapezoidal_maps.

Prelude

- Homework 1 is due today.
- Homework 2 is due in two weeks: Tuesday February 20th by the start of class.

Trapezoidal Maps

- A couple weeks ago, we discussed polygon triangulation: partitioning a simple polygon into triangles.
- Today, we're going to discuss another partitioning scheme that works in even more general settings.
- Building this scheme will be the first step in building a data structure for planar point location queries: given a point p, which region of a planar subdivision does it live in?
- 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.

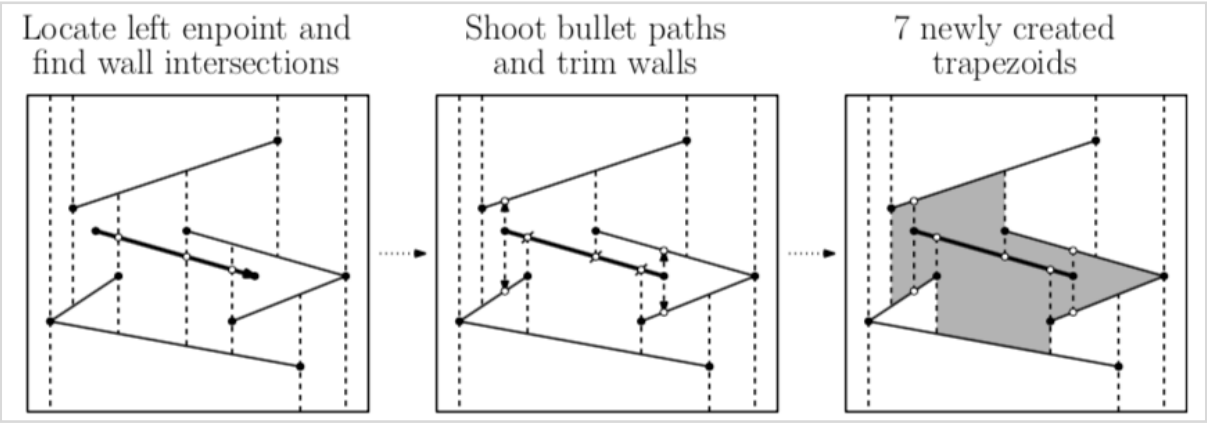


- We'll build a subdivision of space that respects the line segments.
- Start by adding a big bounding rectangle.
- Next, we send two *vertical extensions* or *bullet paths* from each endpoint. They go until they run into another segment or the boundary of the box.
- The combination of original segments 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*. The top and bottom of trapezoids are from the original input segments; the left and right sides come from the vertical extensions. However, if an endpoint was shared by two segments, then the face between them will look like a triangle.
- Also, note the top or bottom side of a trapezoid may have multiple vertices from where several vertical extensions landed on the same input segment.

- If these are going to be useful for a data structure, they'd better not be too big.
- Claim: Given an n-element set S of line segments, the resulting trapezoid map has at most 6n + 4 vertices and 3n + 1 trapezoids.
- Proof:
  - Each vertex shoots off two vertical extensions that create another two vertices in the final map. So 3 map vertices per segment vertex. There are 2n segment vertices, so these account for 6n total. The other 4 come from the bounding box.
  - The left side of every trapezoid (except for the leftmost) is bounded by a vertex from S. Left endpoints of input segments can bound two trapezoids and right endpoints can bound 1. So a single segment has vertices on the left side of at most 3 segments for 3n total. The extra 1 comes from the leftmost trapezoid bounded by the bounding box.

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.
- 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.

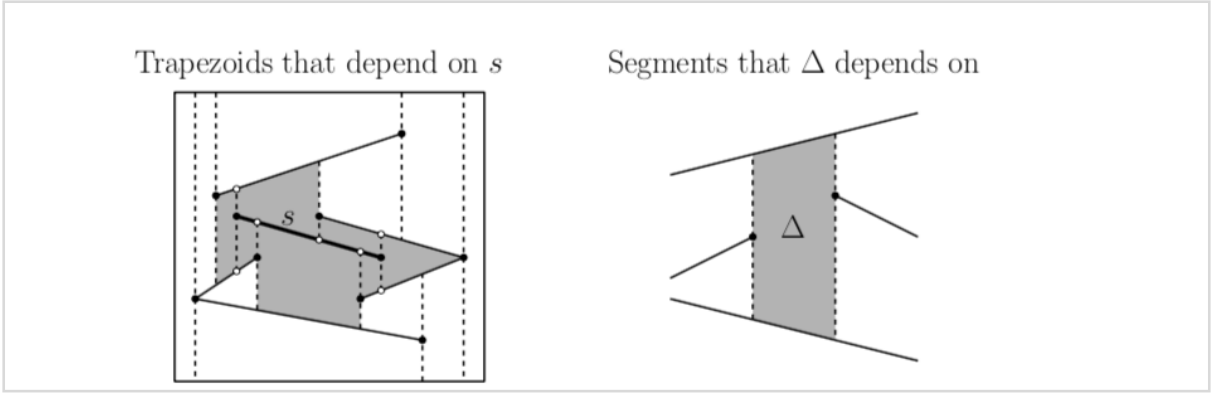


- Let k_i be the number of newly created trapezoids. Ignoring the time taken to locate the left endpoint, this operation takes O(k_i) time. In short, we add 4 new segments and trim back k_i - 4 walls. We can use a suitable representation of the trapezoidal map like a DCEL

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.

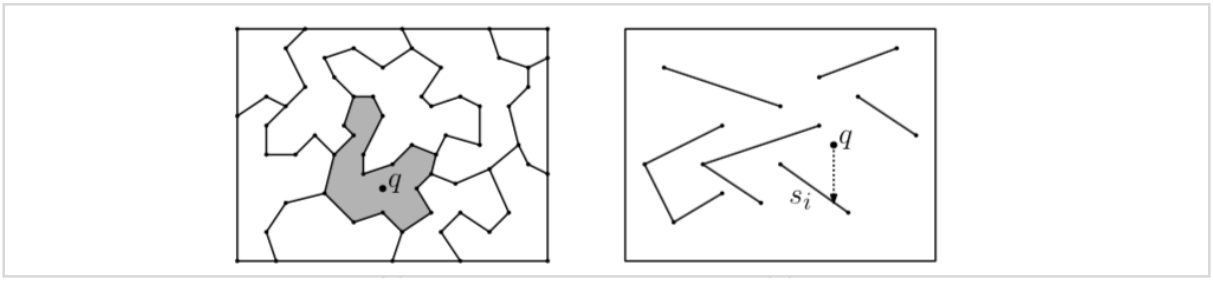


  - Let delta(Delta, s) = 1 if Delta depends on s and 0 otherwise.
  - E(k_i)
    - = 1/i sum_{s in S_i} (# of trapezoids that depend on s)
    - = 1/i sum_{s in S_i} sum_{Delta in T_i} delta(Delta, s)
  - It's hard to get a handle on delta(Delta, s), because some segments intersect a lot of trapezoids 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

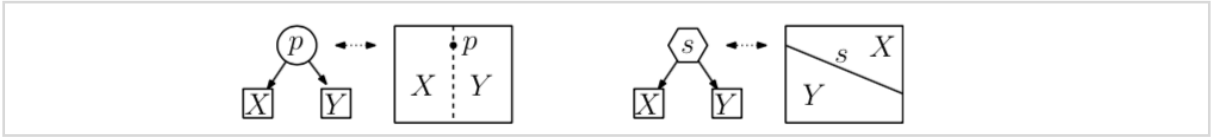
segments s for which delta(Delta, s) = 1. The right and left side contain a single segment endpoint, for which delta(Delta, s) may be 1. Other segments don't matter (and if multiple segments share an endpoint, then none of them could have been the one to introduce that endpoint if added last).
    - 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

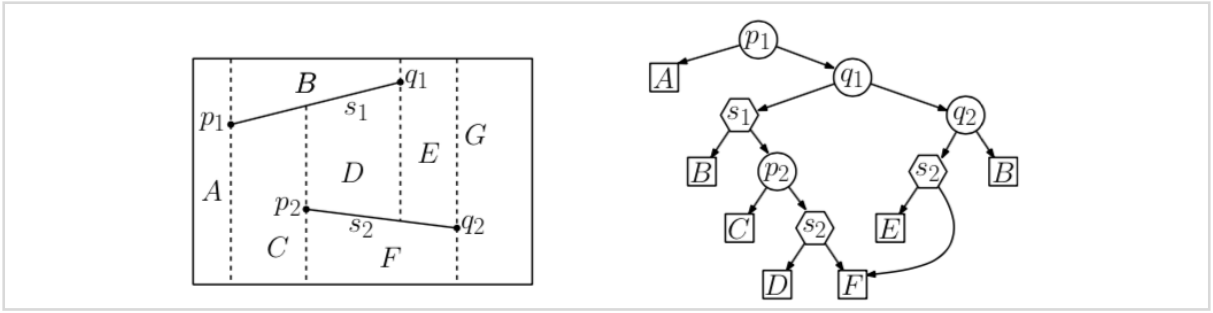
- With the remaining time, let's discuss the data structure for planar point location.
- We'll still assume we're given segments S = {s_1, …, s_n}.
- Assuming those segments form a planar subdivision, it's natural to ask, given a query point q, which face it lies in.
- But instead we'll ask more general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



- Earlier we were using rooted trees to build our data structures. 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* 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.



- 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.



- And the construction of the data structure is done simultaneously with building the trapezoidal map. Every time you add a new segment, you turn the destroyed trapezoids' leaves into internal nodes and append the new trapezoids as new leaves. Then you search the new structure when you insert the next point!
- Query time is proportional to the distance you must travel to reach a leaf.
- But because the actual structure depends upon the order in which we add segments, we can only talk about *expected* query times.
- Next time, we'll discuss how to modify the data structure as we add new segments.
- Also, we'll discuss how the expected time to query any point q, including segment endpoints, is O(log n). That implies the expected time for constructing a trapezoidal map is O(n log n) total.
- But let's save that for Thursday so we have something to talk about.