CS 6301.008.18S Lecture—February 27, 2017

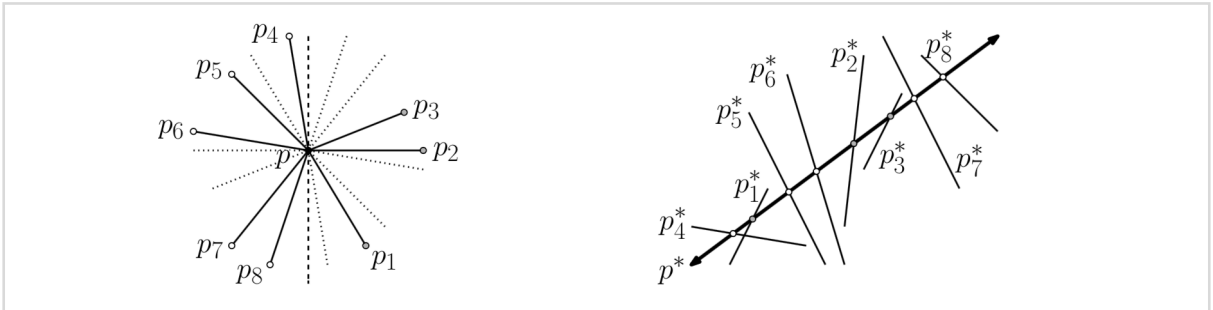
Main topics are #line_arrangement_applications, #topological_plane_sweep, #sorting_angular_sequences, #narrowest_k-corridor, and #halfplane_discrepancy, and #levels.

Prelude

- Project proposals are due next Tuesday. Feel free to email me or visit office hours if you want to discuss ideas.

Line Arrangement Recap and Sorting Angular Sequences

- Last week, we discussed line arrangements. A set of L lines induces a planar subdivision with vertices at intersections and edges following lines.
- An arrangement of n lines has complexity Theta(n^2) and can be built in O(n^2) time. Today, we're going to discuss some more interesting applications.
- Let's start with a relatively easy one: sorting angular sequences.
- Given n points in the plane, for each point p, sort the other n - 1 points into their counterclockwise order around p.
- This is straightforward to do in O(n^2 log n) time total, by just running your favorite sorting algorithm once per point. But we can get it down to O(n^2) using line arrangements and point-line duality.
- Recall point-line duality. A point p = (x, y) is dual to a line p^* : b = xa - y, and a line ell : y = ax - b is dual to the point ell^* = (a, b).
- Point p lies above line ell at vertical distance h if and only if line p^* lies below ell^* at vertical distance h.
- So take a point p from the input and look at dual line p^*. The duals from the other lines intersect p at various points, each of which is the dual of a line through p and another input point.

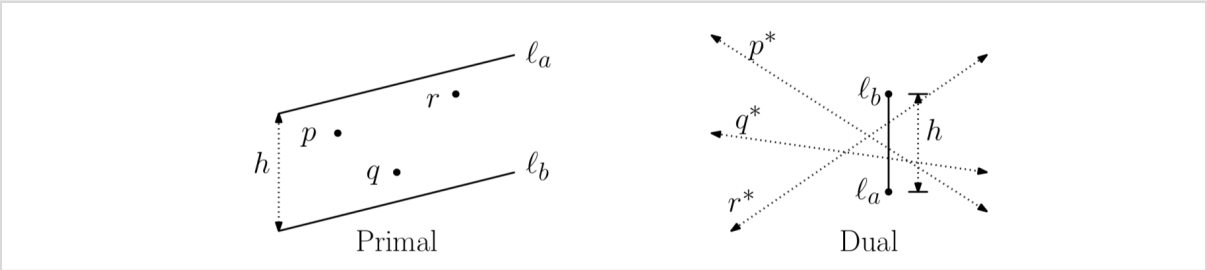


- From the definition of duals, the slope of a primal line is equal to the a-coordinate of its dual point, so the intersections with p^* go left to right in accordance with increasing slopes of the primal lines.

- So, here's what we'll do:
  - Compute the line arrangement *in the dual*.
  - For each point p, partition other points into those left of p and those right of p.
    - Walk along dual line p^* to learn order of slopes. The counterclockwise ordering of points around p is those to the right in increasing order of slope followed by those to the left in increasing order of slope.
- So now it takes O(n) time to figure out the counterclockwise order around point p for O(n^2) time total.

Narrowest k-corridor

- For this problem, we are given a set of n points in the plane and an integer k, 1 ≤ k ≤ n.
- We want to compute the narrowest pair of parallel lines that enclose *at least* k points. By narrowest, I mean minimize their vertical distance.
- Let's assume k = 3 for now to make things more simple. Let's also assume general position so no three points are collinear and no two points share the same x-coordinate.
- These assumption imply there will be exactly three points between the lines we seek.
- In the dual, two parallel lines ell_a above and ell_b below form a pair of vertically aligned points ell_a^* and ell_b^*. The vertical distance between the points is the vertical distance between the lines.
- If there are three points between ell_a and ell_b, then their dual lines pass above ell_a^* and below ell_b^*.



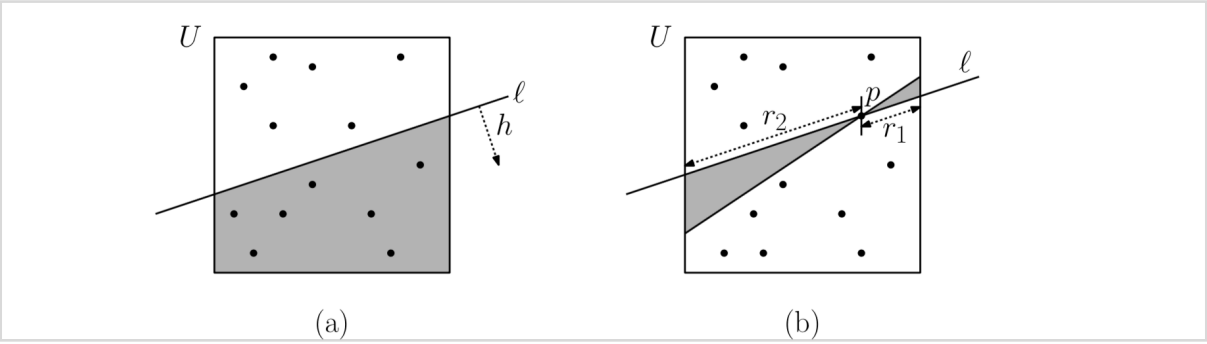
- Therefore, we want to solve the following problem in the dual with k = 3:
- Shortest vertical k-stabber: Given n lines, determine the shortest vertical segment that stabs k of the lines.
- Note the shortest vertical 3-stabber has one endpoint on the intersection of two lines and the other on another line. Otherwise, we could scoot it left or right to make it shorter.
  - (In the primal plane, this means the narrowest 3-corridor has two points on one of the lines and one point on the other.)
- So here's what we'll do:
  - Dualize the n input points and compute their arrangement.
  - Do a left to right plane sweep over the arrangement with vertices as event points. At each vertex, check the vertical distance to the edge immediately above and below it.
- There's O(n^2) events that we can take care of in O(log n) time each using the standard

data structures, so O(n^2 log n) time total.

- So what about k > 3? Generally, we should look k - 2 lines above or below each vertex.

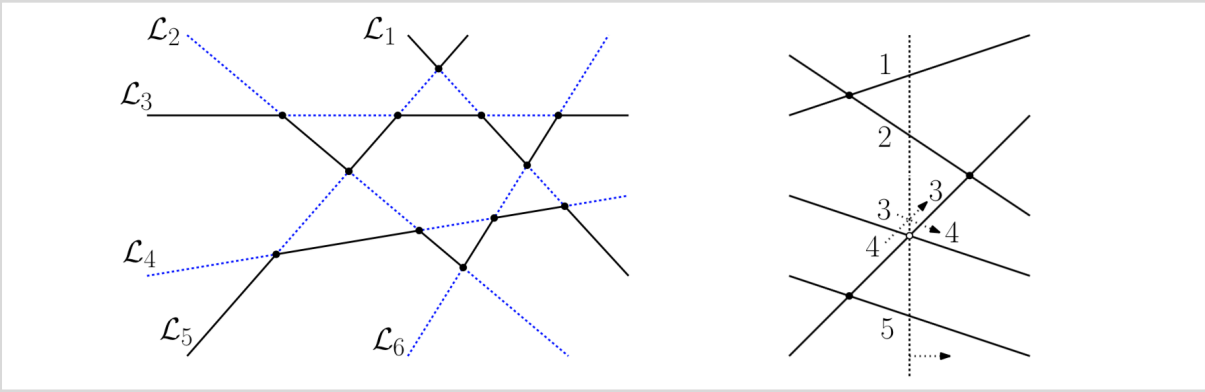
Halfplane Discrepancy

- Now we'll look at a problem motivated from computer graphics and sampling.
- Consider the unit square U = [0, 1]^2.
- We want to sample a collection of n points P subseteq U.
- But we want to know it's a good sample. Ideally, for any halfplane h, h should contain about the same fraction of points from P as it contains U.
- Precisely, let mu(h) be the area of h intersect U, and mu_P(h) = |P intersect h| / |P|.
- The *discrepancy* of P with respect to h is Delta_P(h) = |mu(h) - mu_P(h)|.
- In (a) below, mu(h) ≅ 0.625, while mu_P(h) = 7/13 ≅ 0.538. So Delta_P(h) ≅ 0.087.



- Ideally, the discrepancy for each halfplane is small. To measure how good our sample P is overall, we define the *halfplane discrepancy* as Delta(p) = sup_h Delta_P(h).
- So let's try find an algorithm to compute the halfplane discrepancy.
- Lemma: Let h denote the halfplane maximizing discrepancy with respect to P, and let ell denote the line that bounds h. Either:
  1. one point of P is the midpoint of line segment ell intersect U or
  2. ell passes through two points of P
- Proof:
  - If ell is disjoint from all points, then we can move it up or down to change the area without changing the number of points in the halfplane. One of the two directions increases the discrepancy.
  - If ell only intersects one point but not at its midpoint, then we can rotate it a bit around that point without sweeping over any other points.
  - Notice above how we sweep two triangles as we rotate. One is smaller than the other, so the area in h intersect U is changing as we sweep.
  - Again, one of the sweep directions increases discrepancy.
- One thing to note: since the line will go through some points, we need to decide if they belong in the halfplane or not. We'll just include or exclude them depending on which gives the worse discrepancy.

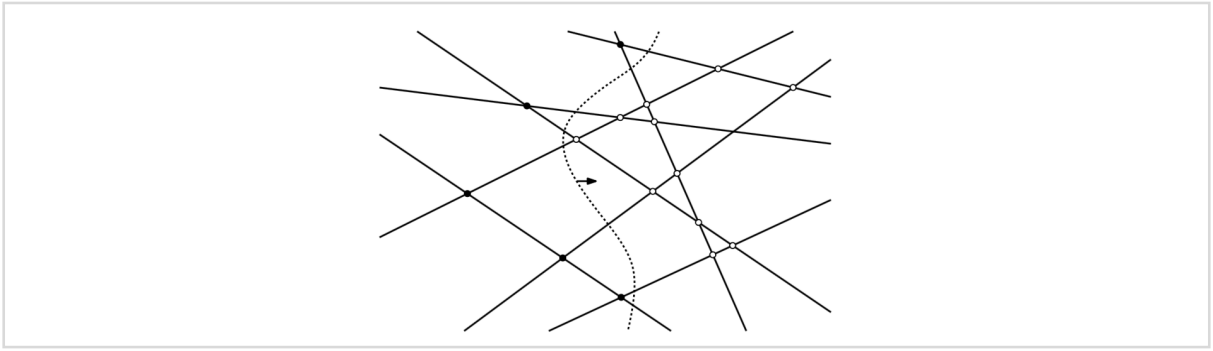
- For each point of type 1., there are only a constant number of lines we need to check. In short, as we rotate a line around a point p, there are moments where the intersection area becomes a maximum or minimum, and there are only a constant number of these.
- There are O(n^2) lines of type 2. We'll use line arrangements to pick the worst one.
- The dual of the line ell through p_i and p_j is a vertex ell^* in the line arrangement of dual lines P^*. So we can check all those O(n^2) vertices.
- Given a primal line/dual vertex, we can compute the area of the halfplane intersection in O(1) time. How do we compute the number of points in the halfplane?
- Well, the points in ell's upper halfspace lie below ell^* in the dual plane, and the points in ell's lower halfspace lie above ell^* in the dual plane. We need a way to count those dual lines.
- For this we have a concept called *levels* in an arrangement. A point is in level k, denoted L_k, if there are at most k - 1 lines above the point and at most n - k lines below (these definitions are exactly one greater than the book's).
- The levels are bounded by x-monotone polygonal curves. For example, the upper envelope is level 1, and the lower envelope is level n. Assuming general position, vertices of the arrangement lie on two levels.
- We can use a plane sweep to compute the level of each vertex. At x = -infty, we can use the slope of the lines to determine their levels. When we sweep a vertex, we swap the level number associated with the vertex's lines.



- The plane sweep takes O(n^2 log n) time.
- By order reversing, the smaller level number of a vertex minus 1 is the number of primal points strictly below the corresponding primal line. The larger level number is the number of primal points on or below the line. We can say similar things for the primal upper halfplanes.
- Meaning we have an O(n^2 log n) time algorithm to compute the discrepancy for all O(n^2) halfplanes that may be the worst, meaning O(n^2 log n) time compute the overall halfplane discrepancy.

Topological Plane Sweeps

- Let's finish up with a some discussion on all these plane sweeps.
- One down side of the algorithms I talked about today is that they use O(n^2) space. For outputting the line segments based on angles, that can't be helped. But there's no reason to think we need Omega(n^2) space to do narrowest k-corridor or halfplane discrepancy.
- Well, it turns out we can save the space by not precomputing the whole line arrangement.
- Since we have to plane sweep everything anyway, we can just remember the part of the arrangement that intersects our sweep line. That takes only O(n) space, and looks almost identical to the line segment intersection algorithm we discussed.
- In fact, it's a bit easier. The only events occur at vertices, which mean the only changes to the sweep line are swapping adjacent intersecting lines. We can store the sweep line as a basic array, ordering the lines top to bottom, and do each swap in constant time.
- But we still have to deal with the event queue, so the running time remains O(n^2 log n).
- But there's another trick we can play called a *topological plane sweep*.
- Essentially, you relax the order in which you process events. The vertices on each line are processed in left to right order, but vertices on different lines may be swept in different orders.
- Instead of the sweep line being a real vertical line, it's now more accurate to think of it as a *pseudoline* that intersects each line of the arrangement exactly once.



- Now events take constant time each for O(n^2) time overall and O(n) space.
- We can apply topological plane sweep to guarantee the same time and space bounds for narrowest k-corridor and halfplane discrepancy.