CS 6301.008.18S Lecture—January 23, 2017

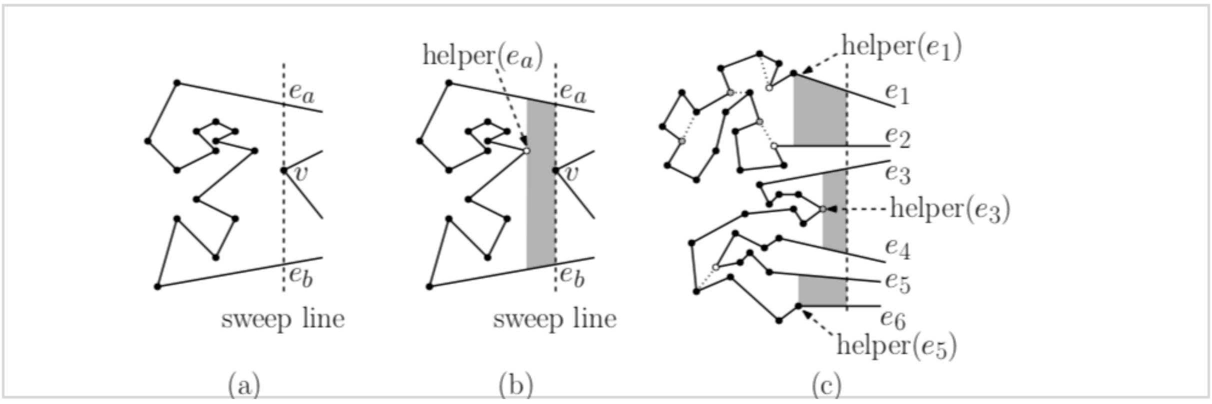
Main topics are #polygon_triangulation, #halfplane_intersection, and #point-line_duality.

Prelude

- Homework 1 is out today on eLearning and the website. You may work in groups of up to three people. I need at least one person in each group to turn in your answers to eLearning, but you should probably all do it just in case. See the homework and website for policies and suggestions.
- The homework 1 due before class starts on Tuesday, February 6th.

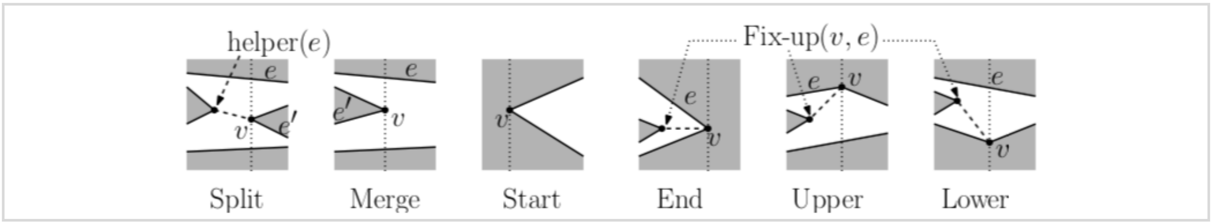
Monotone Subdivision

- Last time, we discussed triangulating a planar graph. We looked at an O(n) time algorithm to do so given an x-monotone polygon. Today, we'll look at an O(n log n) time algorithm to subdivide a polygon into x-monotone pieces.
- Again, we'll use a plane sweep algorithm, sweeping from left to right.
- The algorithm is largely based on the following observation: A polygon is not horizontally monotone if and only if there is a *scan reflex vertex*, a reflex vertex where both incident edges go left or both incident edges go right.
- The book calls the first kind *merge vertices* and the second kind *split vertices*. The goal is for the sweep line to discover these two kinds of vertices, and add diagonals to them when possible.
- For merge vertices, we'll need to wait for some point in the future to add the diagonal.
- For split vertices, though, we need to add the diagonal immediately to some vertex further left. But how will we know where to add that diagonal?
- Consider a split vertex v. There's an edge e_a immediately above it and an edge e_b immediately below. If we sweep *left* from v, we'll encounter some first vertex below e_a that is visible.
- helper(e_a): Suppose the polygon interior is below e_a. helper(e_a) is the rightmost vertex u left of the sweep line such that the vertical segment between e_a and u is entirely in the polygon.
- Again, I'm only defining helper(e_a) if the polygon interior is below e_a. Mount's notes define it for all edges. The helper could be incident to e_a or e_b or neither, and it changes as the sweep line moves.



- See how we can take a segment up to e_a and then down-left to helper(e_a)? That whole triangle lies in the polygon, so it's safe to send a diagonal from v back to helper(e_a).
- Our goal will be to add diagonals from each split vertex back to a helper vertex. We'll also keep track of helper merge vertices, since we can easily send diagonals back to them, even from vertices that aren't scan reflex.
- Let's formally define the sweep status. We'll store edges intersected by the sweep line with the polygon interior below them along with their helpers. We'll store them top to bottom in an ordered dictionary again, so we can easily figure out what e_a and helper(e_a) are relevant for each event.
- We'll be trying to help out merge vertices whenever we can, so we'll use a subroutine Fix-up(v, e) where a) e is above v or incident going up-left and b) the polygon interior is below e: if helper(e) is a merge vertex, add diagonal from v to helper(e).
- Now we just need to deal with lots of cases!
- MakeMonotone(P):
  - Sort vertices from left to right as v_1, …, v_n.
  - For i ← 1 to n:
    - v ← v_i
    - If v is a split vertex
      - Find edge e immediately above v on sweep line.
      - Add diagonal to helper(e).
      - helper(e) ← v
      - Let e' be the lower edge incident to v.
      - Add e' to sweep line status and set helper(e') ← v.
    - If v is a merge vertex
      - Let e' be the lower edge incident to v.
      - Fix-up(v, e')
      - Delete e' from sweep line status.
      - Find edge e immediately above v on sweep line.
      - Fix-up(v, e).
      - helper(e) ← v
    - If v is a *start vertex* (both incident edges go right and v is not reflex)

            - Let e be the higher incident edge on v.
            - Add e to sweep line status and set helper(e) 🡐 v.
        - If v is an *end vertex* (both incident edges go left and v is not reflex)
            - Let e be the higher incident edge on v.
            - Fix-up(v, e)
            - Delete e from sweep line status.
        - If v is an *upper-chain vertex* (incident edges go both directions and interior is below v)
            - Let e be incident edge left of v and e' be the incident edge to the right.
            - Fix-up(v, e)
            - Replace e with e' in sweep line status and set helper(e') 🡐 v.
        - If v i a *lower-chain vertex*
            - Find edge e immediately above v on sweep line.
            - Fix-up(v, e)
            - helper(e) 🡐 v

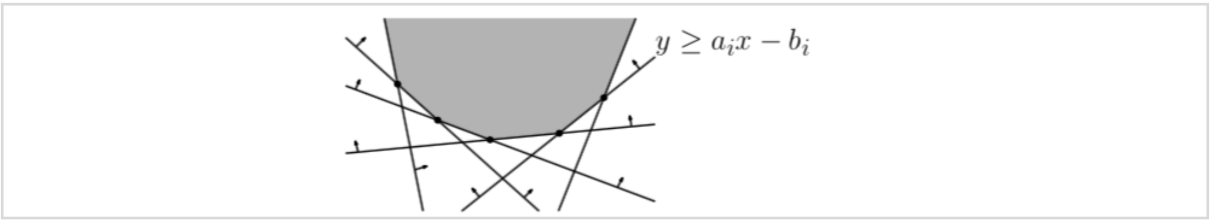


    - By only going for helpers of edges immediately above each v, we safely add diagonals.
    - We add diagonals going left for every split vertex, and we make sure to add a diagonal to helper merge vertices before their edges go away or they are replaced by new helpers. So the resulting polygons are horizontally monotone.
    - Finally, each event can be handled in O(log n) time and there are n events, so the whole thing + sorting takes O(n log n) time.

Halfplane Intersection

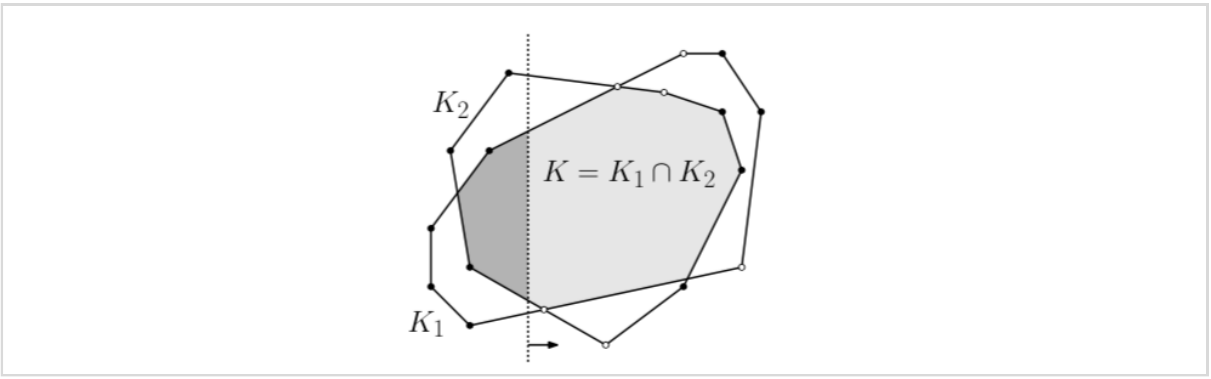
- For the rest of the day, we're going to focus on a another fundamental problem in computational geometry that has some nice connections to things you've seen before.
- A line in the plane partitions the plane into two regions called *halfplanes*.
- An *open* halfplane does not include the line. A *closed* halfplane includes the line.
- Today, we're going to work on the problem of computing the intersection of a set H = {h_1, …, h_n} of closed halfplanes.
- Since each halfplane is a convex set, this intersection will also be a convex set.
- Unlike in the convex hull problem, though, the convex set may be empty or unbounded.
- Like convex hull, halfplane intersection is useful for things like collision detection. It's also a vital component in optimization problems. I'll go into the latter point more on Thursday.

- To make the problem concrete, we'll represent each halfplane in the following way. First, let's just assume by general position that there are no vertical lines. Any other line can be represented by an equation y = ax - b.. Time permitting, the reason I'm using -b will be clear by the end of today's lecture.
- The *lower* halfplane for that line is y ≤ ax - b and the *upper* halfplane is y ≥ ax - b. So, each h_i is some y ? a_i x - b_i.



Divide-and-Conquer

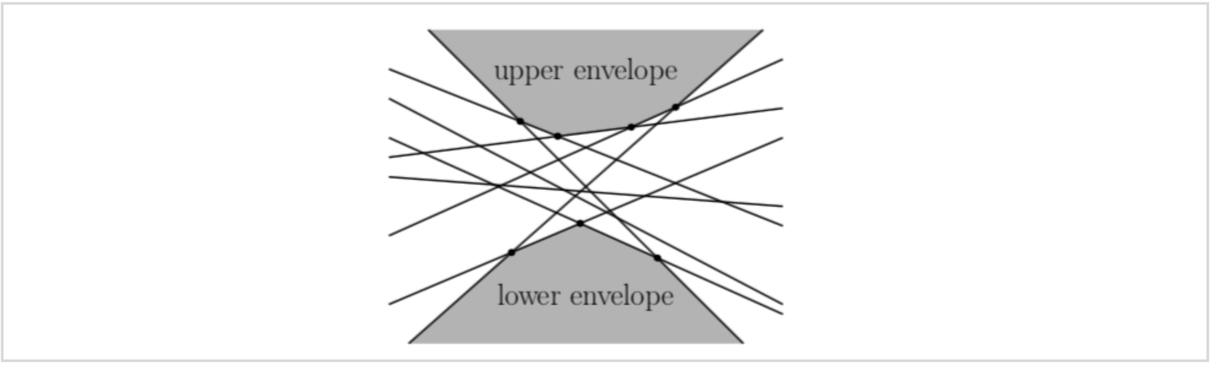
- We'll start with a divide-and-conquer algorithm described in detail in the book. I won't spend too much time, because the combine step is just another example of plane sweep.
- So, divide-and-conquer is paradigm for making recursive algorithms. You divide the input into simple pieces, conquer each piece by solving the problem recursively, and then combine them into the solution for your original problem.
- HalfplaneIntersectionDAC(H):
  - If n = 1, return h_1
  - Divide H into subsets H_1 and H_2 of size floor(n/2) and ceil(n/2).
  - Compute intersections of H_1 and H_2 recursively as K_1 and K_2.
  - Let K be the intersection of (possibly unbounded) convex polygons K_1 and K_2; return K.
- If T(n) is the running time, then it follows the recurrence T(n) =
  - 1 if n = 1
  - 2T(n/2) + M(n) if n > 1
- where M(n) is the time it takes to merge.
- We'll use an O(n) time merge procedure, so the whole thing will take T(n) = O(n log n) time just like merge sort.
- The merge procedure is a sweep line algorithm.
- For the status, we need to store the segments of K_1 and K_2 intersected by the algorithm. However, there are at most four segments total, so we can just use a 4-element list to hold the status and do everything in constant time.
- Similarly, there are at most 8 event points we need to care about; the endpoints of the current segments and the up to four intersections between these segments. We'll just manually search the constant sized event queue for the next event each time we need one.
- There are only a constant number of possible cases for events and they're relatively easy to handle in constant time each.



- So the running time is proportional to the number of intersections between K_1 and K_2's segments, but that is only O(n) since each segment can enter or leave the other intersection at most once.
- Again, you can look at the book for details.

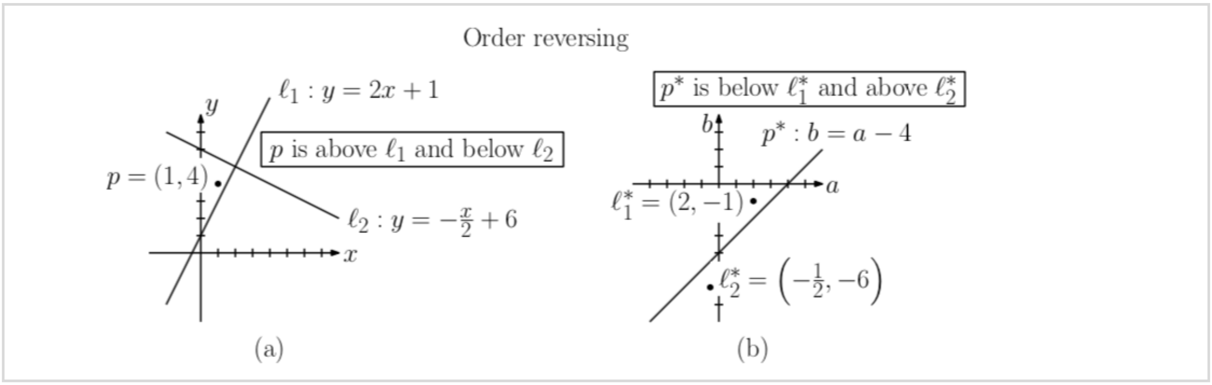
Point-line Duality

- For this lecture, I instead want to focus on a variant of halfplane intersection and its connections to an important topic in computational geometry.
- In the *lower envelope* problem, we're given L = {ell_1, …, ell_n} where each ell_i is of the form y = a_i x - b_i. We want to compute the intersection of their lower halfplanes y ≤ a_i x - b_i. The *upper envelope* problem has the symmetric definition.

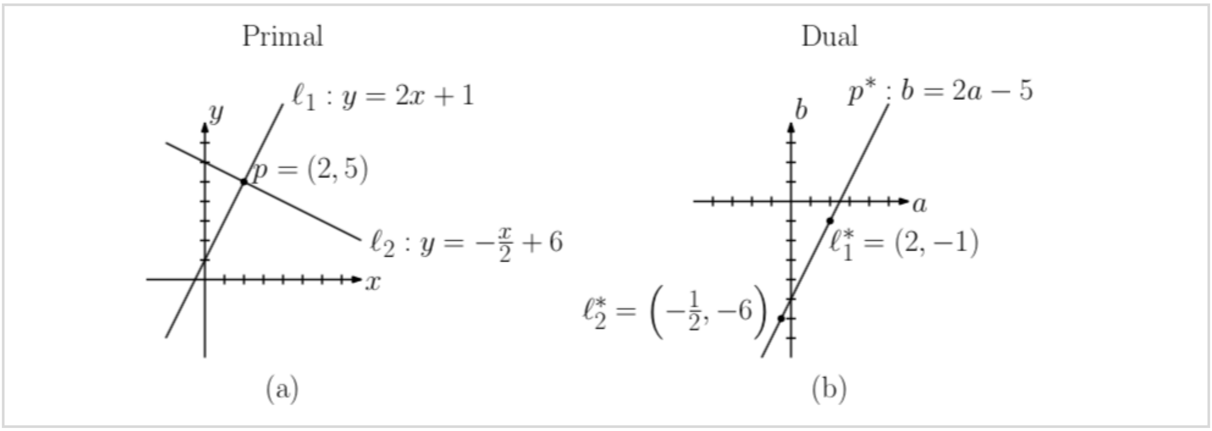


- So the lower envelope problem is just a special case of halfplane intersection. But the cool thing is that it is in some sense the exact same problem as computing an *upper* convex hull.
- So, consider a line ell: y = ell_ax - ell_b. We can represent it as the pair (ell_a, ell_b), so you can think of it as a point living in some other 2-dimensional space. We denote this point as ell*.
- We call the (x, y)-plane the *primal plane*. The new (a, b)-plane is the *dual plane*.
- Now, if you want to go backwards, you'd take the point (ell_a, ell_b) and turn it into the line y = ell_a x - ell_b. But we can do that exact same point to line transformation starting in the primal plane as well.
- So if p = (p_x, p_y), we can let p^* be the line b = p_x a - p_y.
- We have a way to turn primal lines and points into dual points and lines, respectively.

- This transformation is called *point-line duality*. It's a similar concept to other kinds of duality you may have seen such as linear programming duality and planar graph duality.
- And it comes with some interesting properties that are useful for solving certain problems in computational geometry:
  - Self inverse: p^ = p
    - Comes straight from the definition.
  - Order reversing: Point p is above*on*below line ell in the primal plane if and only if line p^* is below*on*above point ell^* in the dual plane
    - p is on or above ell <=> p_y ≥_ ell_a p_x - ell_b <=> ell_b ≥ p_x ell_a - p_y <=> p^* is on or below ell^*


Order reversing
$\ell_1 : y = 2x + 1$
y
p is above ℓ_1 and below ℓ_2
$p = (1, 4)$
$\ell_2 : y = -\frac{x}{2} + 6$
x
(a)
p^* is below ℓ_1^* and above ℓ_2^*
b
$p^* : b = a - 4$
a
$\ell_1^* = (2, -1)$
$\ell_2^* = \left(-\frac{1}{2}, -6\right)$
(b)

  - Intersection preserving: Lines ell_1 and ell_2 intersect at point p if and only if dual line p^* passes through points ell_1^* and ell_2^*.
    - (in other words, two lines determine a point and in the dual the two dual points determine its line)
    - Follows directly from order reversing.

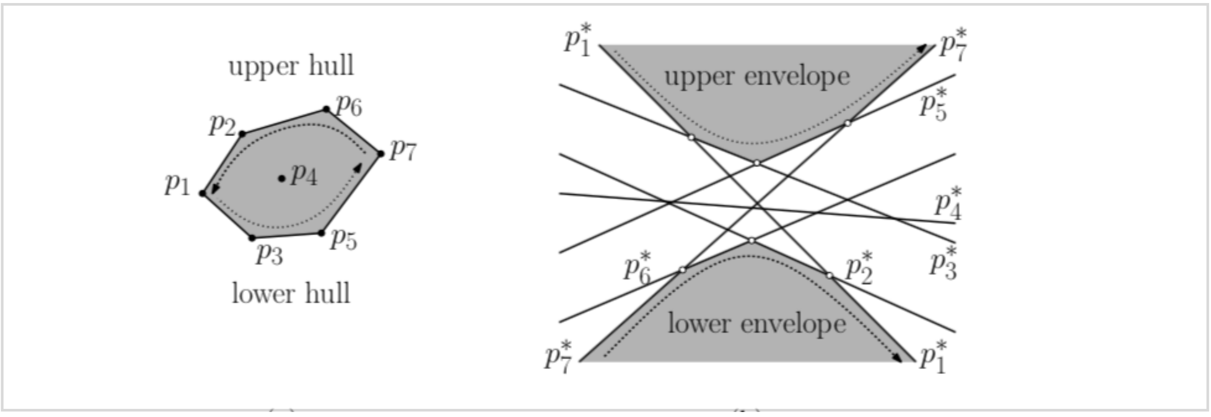


  - Collinearity/Coincidence: Three points are collinear in the primal plane if and only if their dual lines intersect at a common point.
    - Follows directly from intersection preserving.

Convex Hulls and Envelopes

- So why did I want to introduce point-line duality today? Because, it turns out lower envelope and upper convex hull are dual problems! A solution for one is a solution for the other, and you don't even need to change code.

- Lemma: Let P be a set of points in the plane. The counterclockwise order of points along the upper (lower) convex hull of P is equal to the left-to-right order of the sequence of lines on the lower (upper) envelope of the dual P^*.



- Proof:
  - For simplicity, assume no three points are collinear.
  - Consider consecutive points p_i and p_j on the upper convex hull. All points lie below line ell_{i, j} that passes through them.
  - By intersection preserving, the dual point ell_{i, j}^* is the intersection of dual lines p_i^* and p_j^*.
  - By order reversing, all the dual lines of P^* pass above ell_{i, j}^*. Since it intersects two of these lines, it must lie on the boundary of the lower envelope.
  - Finally, as we move along the upper convex hull in counterclockwise order, each line's slope increases monotonically. So their dual points' a-coordinates are increasing, placing them in left-to-right order.
- Remember, the dual of the dual is the primal again, so you can literally compute an upper hull in the dual plane and translate the points back if you want a lower envelope in the primal or just look at what Graham's scan is really doing and write a lower envelope algorithm that is secretly Graham's scan in the dual.
- One final point. Notice how the upper and lower convex hulls are connected, but the lower and upper envelopes aren't? If you're familiar with *projective geometry*, this may seem less surprising. Projective geometry kind of says as you go down to the bottom of the dual plane, you'll wrap around again at the top but left and right are flipped. So for the envelopes, we'd go from left to right along the bottom, follow the rightmost line down, and wrap around again on the left of the upper envelope.