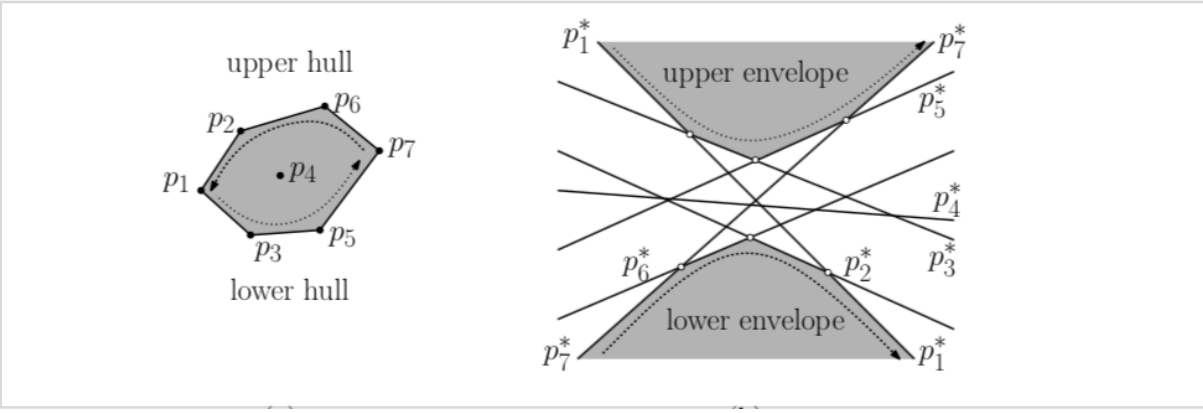
CS 6301.002.20S Lecture 7—February 4, 2020

Main topics are #point-line_duality and #linear_programming.

Point-line Duality

- Let's start with that claim I ended with last time.
- 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.
- The lower envelope program is related to computing the upper convex hull through point-line duality.
- Given primal point p = (p_x, p_y), it's dual line is p^* : b = p_x a - p_y.
- Given a primal line ell : y = ell_a x - ell_b, it's dual point is ell^* = (ell_a, ell_b).
- 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^*.



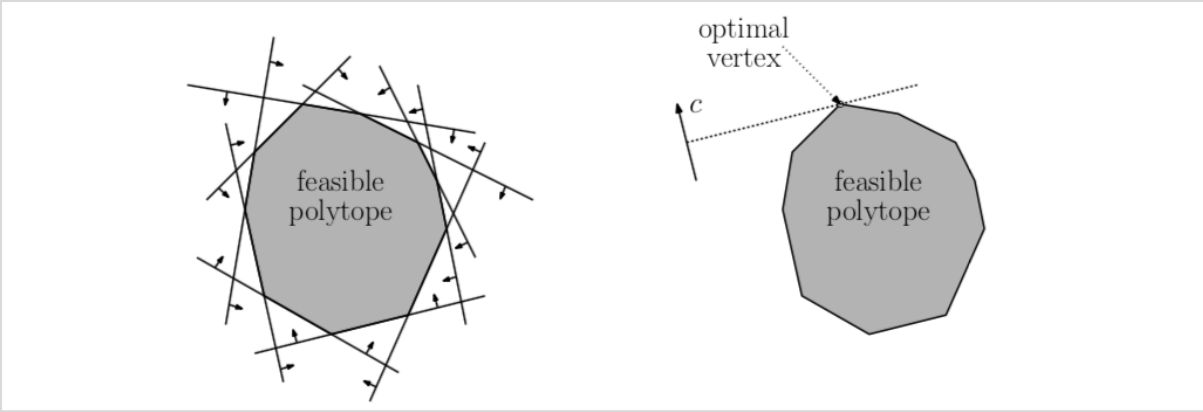
- Proof:
  - For simplicity, assume no three points are collinear.
  - Consider consecutive points p_i and p_j on the upper convex hull. All points lie below line ell_{i, j} that passes through them.
  - By the intersection preserving property, the dual point ell_{i, j}^* is the intersection of dual lines p_i^* and p_j^*.
  - By order reversing, ell_{i, j}^* lies below all the dual lines of P^*. Because ell_{i, j}^ * 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 points's x-coordinate decreases monotonically. So their dual lines' slopes are decreasing, placing them in left-to-right order.
- Remember, the dual of the dual is the primal again, and this leads to a nice algorithm for computing a lower envelope:
- Given L, compute the dual points L^*. Compute the upper convex hull of L^ * using your

favorite algorithm. The dual of the points on the upper hull are the lines on the lower envelope in order.

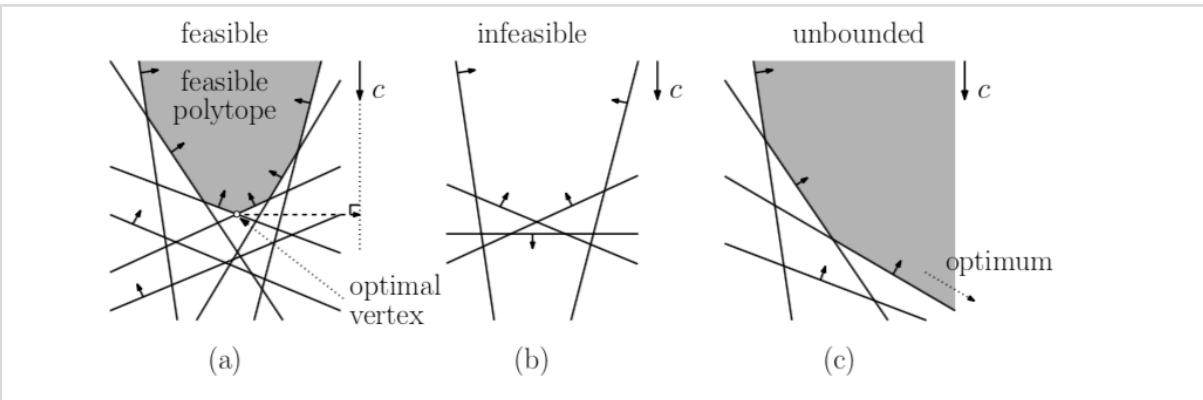
- Or, you could look very closely at what Graham's scan would do with those dual points and figure out how to directly manipulate the given lines in the same way.
- One final point. Notice how the upper and lower convex hulls are connected, but the lower and upper envelopes aren't? If you're familiar with *projective geometry*, this may seem less surprising. Projective geometry kind of says as an ant follows a path down to the bottom of the dual plane, it will wrap around again to the top but what it considers left and right are flipped. Another way to think of it is that the ant ends up *underneath* the plane, so it's perspective becomes backwards from ours. So for the envelopes, the ant would follow the line segments with the lower envelope on its right, flip around to the top, and then following the line segments with the upper envelope on what it thinks is right but the picture suggests is left.

Linear Programming

- For the rest of today and most of Thursday, we're going to discuss a problem closely related to halfplane intersection called *linear programming*.
- Linear programming is a fairly general way to describe certain kinds of optimization problems.
- The goal is to find a good point $(x_1, \ldots, x_d)$ in d-dimensional space R^d where each dimension represents some quantity of a thing you want to acquire or do. For a silly example, say we're trying to sell different kinds of chocolates, and each dimension represents how much of each type, caramel, super dark, milk, etc. we want to sell.
- We are limited by certain *linear constraints*. These are represented as linear functions of x with a constant maximum like $a_1 x_1 + \ldots a_d x_d \leq b$. So maybe the b value is the total amount of cocoa you have on hand and the a values are how much cocoa each kind of chocolate uses.
- Yes, we could use $\geq$ instead, but we can always rewrite such constants with a $\leq$ just by negating both sides of the inequality.
- Geometrically, each constraint defines a closed halfspace in R^d. Their intersection forms a convex region called the *feasible polytope*. (Some might prefer the term polyhedron.)
- We are also given a *linear objective function* $c_1 x_1 + \ldots + c_d x_d$ that we want to maximize (again, you can change a minimization problem into a max by negating the objective). Maybe these are the profits generated from making each kind of chocolate.
- Geometrically, $(c_1, \ldots, c_d)$ is a vector in R^d. You want to find a vector $(x_1, \ldots, x_d)$ in the feasible polytope whose projection onto the objective vector (the dot product) is maximized. In other words, go as far that direction as you can, please. Assuming general position, the optimal solution is at a vertex of the feasible region called the *optimal vertex*.



- All together, a d-dimensional linear programming problem looks like
  - maximize c_1 x_1 + … + c_d x_d
  - subject to
    - a_1, 1 x_1 + … + a_1, d x_d ≤ b_1
    - a_2, 1 x_1 + … + a_2, d x_d ≤ b_2
    - …
    - a_n, 1 x_1 + … + a_n, d x_d ≤ b_n
  - where each a_i, j; c_i; and b_j are given as real numbers.
- You could also think of it in matrix notation as
  - maximize c^T x
  - subject to Ax ≤ b
  - where c and x are d-dimensional vectors, b is an n-dimensional vector, and A is an n X d matrix.
- There are three possible outcomes for a given linear programming problem:
  - Feasible: An optimal point exists. Assuming general position, it is unique and lies at the optimal vertex.
  - Infeasible: The feasible polytope is empty and no feasible solution exists.
  - Unbounded: The feasible polytope is unbounded in the direction of the objective function, so no finite optimal solution exists.

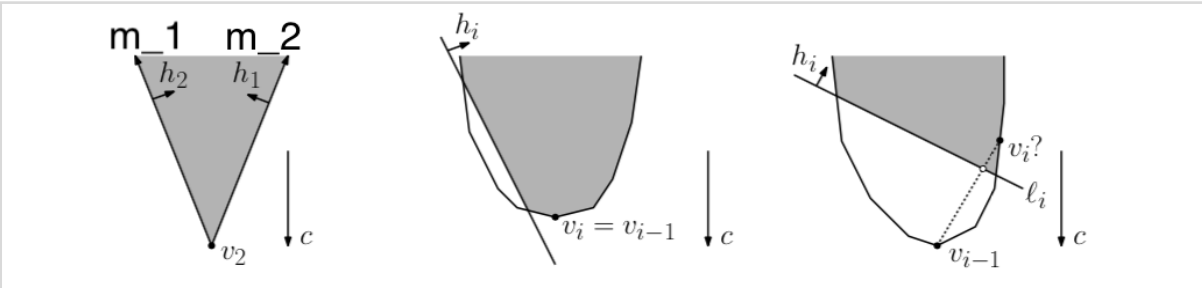


Algorithm for 2D

- Often, linear programming is used in situations where there are many constraints and d is

very large.

- There are still interesting problems, though, where the number of variables (or dimension) d is small and the number of constraints n is arbitrarily large.
- For the rest of the lecture, I'm going to focus on 2D. I'll keep the constraints as upper halfplanes and assume the objective function always points straight down.
- Let {h_1, …, h_n} be the halfplanes defined by our linear constraints.
- What I'm going to discuss is an incremental construction algorithm by Seidel ['91]. Similar to Graham's scan, we're going to add these the constraints one by one by finding the optimal solution in the intersection of the first i halfplanes for each i from 0 to n.
- To that end, we need to start with a feasible solution before we've added any constraints. For the lecture, we'll use the following trick.
- Let M be some really really big number. So big that the optimal vertex has coordinates less than M. We'll add two constraints
  - m_1 := {x_1 ≤ M if c_1 > 0, -x_1 ≤ M otherwise}
  - m_2 := {x_2 ≤ M if c_2 > 0, -x_2 ≤ M otherwise}
- Our initial optimal solution will lie at the intersection of m_1 and m_2.



- Let H_i := {m_1, m_2, h_1, h_2, …, h_i} be our two new halfplanes and the first i original halfplanes. Let ell_i be the line bounding h_i. Let C_i := m_1 ∩ m_2 ∩ h_1 ∩ h_2 ∩ … ∩ h_i be their intersection. Let v_i be the optimal vertex in C_i.
- When we add halfplane h_i, the intersection C_i becomes smaller. There's two cases that come up.
- In the first case, the old optimal vertex v_{i-1} is still feasible. Well, we already couldn't do better with fewer constraints, so v_i = v_{i-1}.
- In the second case, v_{i-1} is not longer feasible. So where did it go?
- Lemma: If C_i is feasible but v_{i-1} is not in C_i, then v_i lies on ell_i.
- Proof: Let v_i be the new optimal vertex, and suppose it's not on ell_i. The line segment between v_{i-1} and v_i must pass through h_i at some point. And as we walk along the segment, the objective value must decrease; otherwise, we wouldn't have claimed v_{i-1} to be optimal before. We should have taken the intersection of the line segment and ell_i.
- So that suggests the following strategy: when we add h_i, we check if v_{i-1} is still feasible. If so, set v_i := v_{i-1}. If not, find the optimal feasible point on ell_i.
- And that second problem isn't too hard!
- OK, so assuming ell_i isn't vertical, it has one point per x_1-value.

- Let $f(x_1)$ be the objective value for that point. You can find a linear equation for it using a bit of algebra.
- For halfplane h, let $\sigma(h, \ell_i)$ be the x_1-coordinate for the intersection of ℓ_i and the halfplane bounding h.
- What we want to do is
  - maximize $f(x_1)$
  - subject to
    - $x_1 \geq \sigma(h, \ell_i)$ for each h in H_{i-1} where $\ell_i \cap h$ is bounded to the left
    - $x_1 \leq \sigma(h, \ell_i)$ for each h in H_{i-1} where $\ell_i \cap h$ is bounded to the right
- Oh, it's another linear program.
- But this one is in 1D and therefore easy to solve in $O(i)$ time. We find a value x_{left} that is the max over all the left constraints and x_{right} that is a min over all right constraints.
- There is no feasible solution if $x_{left} > x_{right}$. Otherwise, the optimal solution lies at one of those two extremes.
- So now we know how to find the next optimal vertex v_i! Here's the overall algorithm:
- 2DBoundedLP($H = \{h_1, \ldots, h_n\}$, c, m_1, m_2):
  - $v_0 \leftarrow$ corner of C_0
  - for i ← 1 to n
    - if v_{i-1} in h_i
      - $v_i \leftarrow v_{i-1}$
    - else
      - p ← the point on ℓ_i maximizing objective subject to H_{i-1}
      - if p does not exist
        - return "Infeasible!"
      - else
        - $v_i \leftarrow p$
  - return v_n
- So how fast is it?
- Iteration i takes $O(i)$ time, so the worst-case total running time is $\sum_{i = 1}^{n} O(i) = O(n^2)$.
- And it is possible to set up situations where the algorithm does take quadratic time. Bad running times occur when you need to move the optimal point often. You'll spend only $O(n)$ time total dealing with iterations where the optimal vertex doesn't move.

- And this running time is somehow worse than just doing halfplane intersection in $O(n \log n)$ time and then walking along the boundary of the feasible region.

- The only reason that example performs badly is because the optimal vertex keeps moving. If we found the correct optimal vertex after a couple iterations, the rest of the algorithm would run in O(n) time.
- But that would require picking the halfplanes in a better order. We'll see how to do so on Thursday.