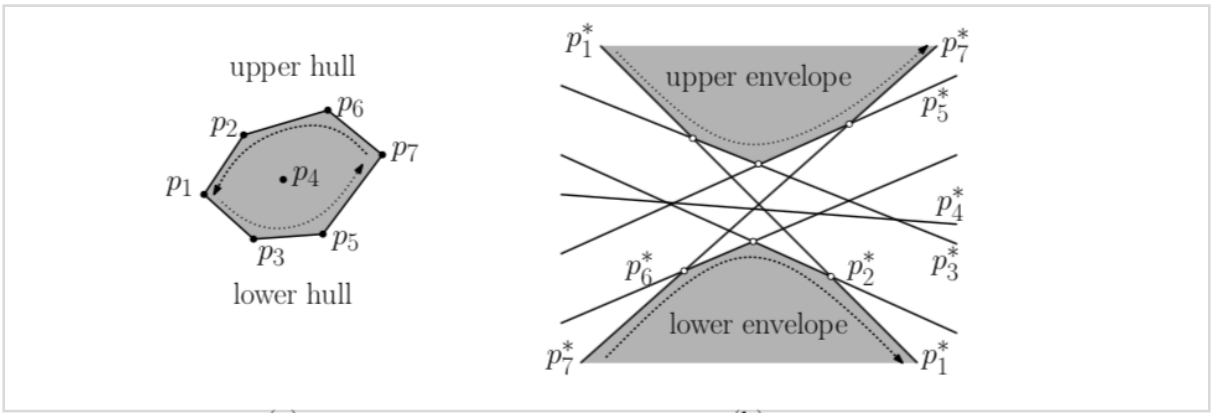
CS 6301.008.18S Lecture—January 25, 2017

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

Convex Hulls and Envelopes

- On Tuesday, we talked about point-line duality: line ell : y = ell_a x - ell_b is dual to ell^* : (ell_a, ell_b) and point p : (p_x, p_y) is dual to p^* : b = p_x a - p_y.
- We also talked about the *lower envelope* problem, where 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 why did I introduce both concepts at the same time? 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^*.



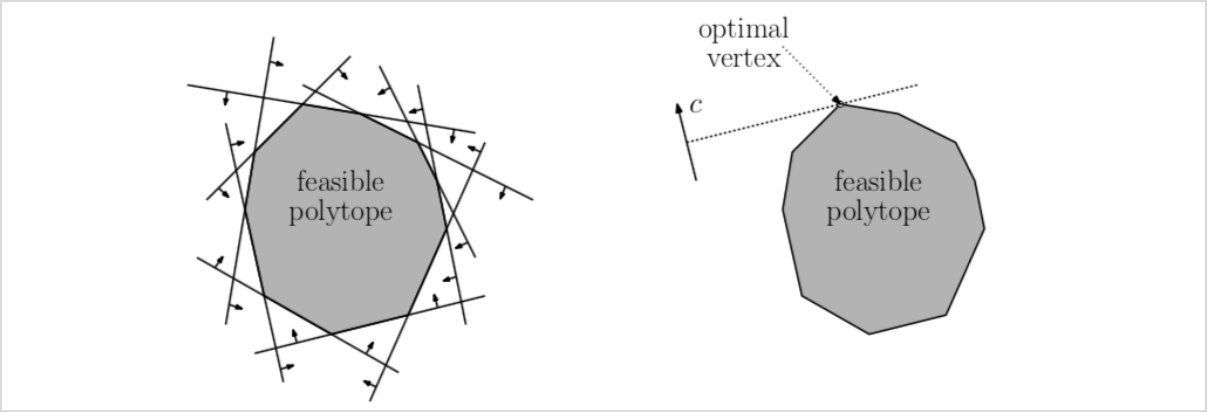
- Proof:
  - For simplicity, assume no three points are collinear.
  - Consider consecutive points p_i and p_j on the upper convex hull. All points lie below line ell_{i, j} that passes through them.
  - By the intersection preserving property, the dual point ell_{i, j}^* is the intersection of dual 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.

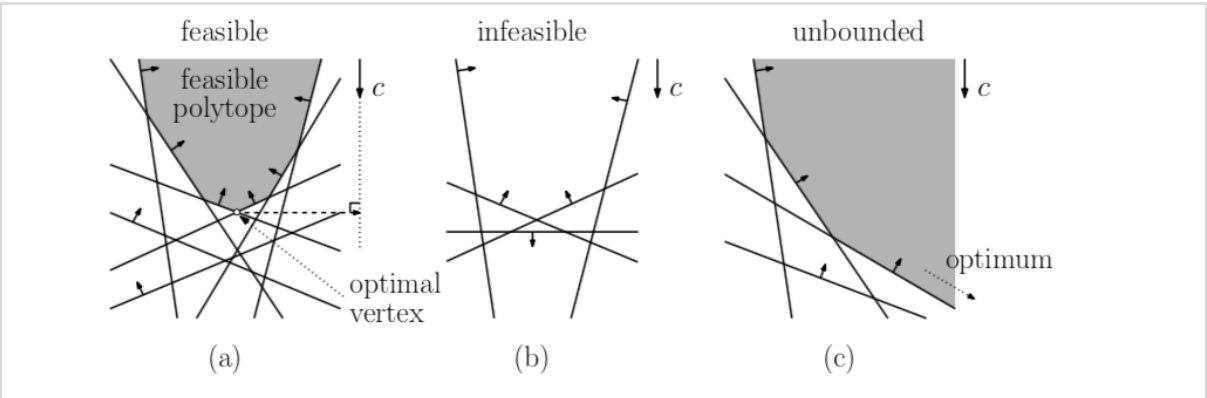
- One final point. Notice how the upper and lower convex hulls are connected, but the lower and upper envelopes aren't? If you're familiar with *projective geometry*, this may seem less surprising. Projective geometry kind of says as you go down to the bottom of the dual plane, you'll wrap around again 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.

Linear Programming

- Today, 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, …, 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 + … a_d x_d ≤ 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 ≥ instead, but we can always rewrite such constants with a ≤ 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*.
- We are also given a *linear objective function* c_1 x_1 + … + 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, …, c_d) is a vector in R^d. You want to find a vector (x_1, …, x_d) in the feasible polytope whose projection onto the objective vector 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.

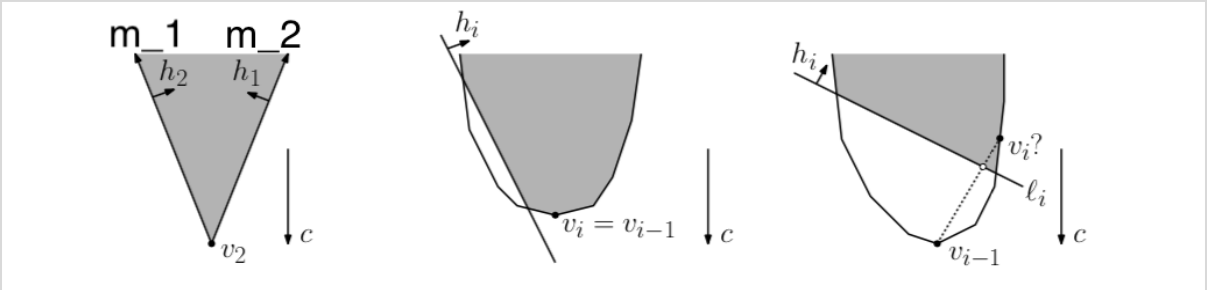


Algorithm for 2D

- 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 ell_i and the halfplane bounding h.
- What we want to do is
  - maximize f(x_1)
  - subject to
    - x_1 ≥ sigma(h, ell_i) for each h in H_{i-1} where ell_i ∩ h is bounded to the left
    - x_1 ≤ sigma(h, ell_i) for each h in H_{i-1} where ell_i ∩ 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, …, h_n}, c, m_1, m_2):
  - v_0 ← corner of C_0
  - for i ← 1 to n
    - if v_{i-1} in h_i
      - v_i ← v_{i-1}
    - else
      - p ← the point on ell_i maximizing objective subject to H_{i-1}
      - if p does not exist
        - return "Infeasible!"
      - else
        - v_i ← p
  - return v_n
- So how fast is it?
- Iteration i takes O(i) time, so the 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 feasible region.

Randomized Incremental Construction

- But 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 have run in O(n) time.
- So adding halfplanes in an arbitrary order is bad. There's no obvious order to add them, though, so why don't we just pick an order at random! We can pick a random permutation in only O(n) time.
- What we end up with is a *randomized algorithm*. Randomized algorithms come in two flavors:
  - Las Vegas: Always correct, but the running time is a random variable. Our algorithm finds the optimal vertex irrespective of the ordering of the halfplanes, so it is a Las Vegas algorithms.
  - Monte Carlo: Has a fixed running time, but whether or not the algorithm is correct is a random event.
- Now, there are still inputs and permutations that give us the Omega(n^2) running time. So the worst-case running time is O(n^2).
- But for Las Vegas algorithms, we're usually more concerned with the *expected* running time, the average running time over all random choices (permutations).
- So how do we analyze it? Again, the time spend performing the random permutation and doing iterations that *don't* move the optimal vertex is O(n).
- Let X_i be a random variable where X_i = 1 if v_{i-1} not in h_i and X_i = 0 otherwise.
- The time spent solving 1D linear programs to move the optimal vertex is itself a random variable equal to O(n) + sum_{i = 1}^n O(i) * X_i.
- So we want the expected value of that sum over all permutations. The thing that makes this possible is the *linearity of expectation*: the expected value of a (weighted) sum of random variables is the (weighted) sum of those variables' expectations.
- In other words E[O(n) + sum_{i=1}^n O(i) * X_i] = O(n) + sum_{i=1}^n O(i) * E[X_i].
- Each E[X_i] is equal to the probability that v_{i-1} not in h_i.
- But how do we compute that? It may be tempting to look at C_{i-1} and then ask what is the probability any of the remaining n - i + 1 halfplanes change the optimal vertex. But now we're conditioning on what C_{i-1} looks like and things are bit ugly.
- Instead, we'll use something called *backwards analysis* where we look at C_i and just ask about what happens before and up to its creation.
- Consider v_i. It lies at the intersection of two halfplane boundaries. If *neither* boundary was h_i, then v_i was already the lowest feasible vertex before adding h_i, and v_{i-1} = v_i.
- Now, if you fix v_i and the set of halfplanes h_1 through h_i that led to it, *but not their order* then all permutations of that subset h_1 through h_i are equally likely, and all choices for h_i out of that subset are equally likely.

- In particular, one of those two halfplanes is chosen with probability at most 2/i. (The probability is actually less if one of the two halfplanes was m_1 or m_2, since neither can be the last halfplane chosen.)
- This upper bound on the probability is true no matter which subset of halfplanes makes up h_1 through h_i, so E[X_i] ≤ 2/i.
- Therefore, O(n) + sum_{i=1}^n O(i) * E[X_i] ≤ O(n) + sum_{i = 1}^n O(i) * 2 / i = O(n).
- The expected running time is linear!

Higher Dimensions and Closeness to Expectation

- So what if d > 2?
- At a high level, the algorithm changes very little. That step where we find a point on a line turns into finding a point on a hyperplane of dimension d -1. To do that, we call the same algorithm recursively but in one lower dimension. The base case is d = 1, which is just that line searching procedure.
- The run time analysis is similar but a fair bit more messy. You can look at the book or notes if you're curious. The running time comes out to be O(d! n). So for constant d, the algorithm still runs in linear time. But d! grows so fast that it's really only practical for very small values of d.
- Now, you may be worried that we only analyzed the expected running time. Is there some small probability that the algorithm is still slow?
- Yes, but it's extremely unlikely. The probability you exceed the expected running time by a factor of b is O((1/c)^{bd!}) for any fixed constant c. This get's very very small as d or b increase.
- Even in 2D, the probably you exceed the expectation by a factor of 10 is less than the probability of getting hit by lightening *twice* in your lifetime.