In this lecture we discuss algorithms for solving linear programs. We give a high level overview of some techniques used to solve LPs in practice and in theory.

We then describe an algorithm for solving linear programming problems with two variables. The algorithm runs in linear time (expected) in the number of constraints. It can be extended to higher dimensions. The running time is still linear in the number of constraints, but blows up exponentially in the dimension.

1 Algorithms for Linear Programming

How can we solve linear programs? The standard algorithm for solving LPs is the Simplex Algorithm, developed in the 1940s. It's *not* guaranteed to run in polynomial time, and you *can* come up with bad examples for it, but in general the algorithm runs pretty fast. Only much later in 1980 was it shown that linear programs could always be solved in polynomial time by something called the Ellipsoid Algorithm (but it tends to be slow in practice). Later on, a faster polynomial-time algorithm called Karmarkar's Algorithm was developed, which is competitive with Simplex. There are many commercial LP packages, for instance LINDO, CPLEX, Gurobi, and Solver (in Excel).

We won't have time to describe most of these algorithms in detail. Instead, we will just give some intuition and the high-level idea of how they work by viewing linear programming as a geometrical problem. Then we'll talk about an elegant algorithm for low-dimensional problems.

1.1 The Geometry of LPs

Think of an n-dimensional space with one coordinate per variable. A solution is a point in this space. An inequality, like $x_1 + x_2 \leq 6$ is saying that we need the solution to be on a specified side of a certain hyperplane. The feasible region is the convex region in space defined by these constraints. Then we want to find the feasible point that is farthest in the "objective" direction.

Let's go back to our first example with S, P, and E from last lecture. To make this easier to draw, we can use our first constraint that $S + P + E = 168$ to replace S with $168 - P - E$. This means we can just draw in 2 dimensions, P and E. See Figure 1.

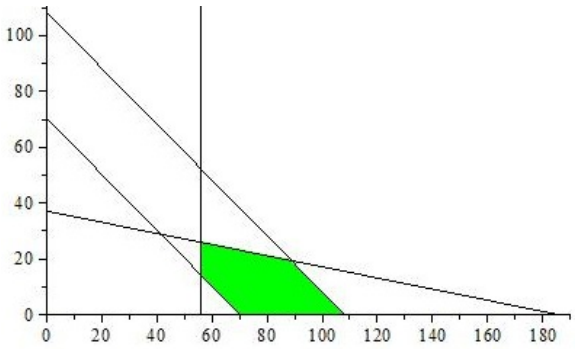


Figure 1: Feasible region for our time-planning problem. The constraints are: $E \geq 56$; $P + E \geq 70$; $P \geq 0$; $S \geq 60$ which means $168 - P - E \geq 60$ or $P + E \leq 108$; and finally $2S - 3P + E \geq 150$ which means $2(168 - P - E) - 3P + E \geq 150$ or $5P + E \leq 186$.

We can see from the figure that for the objective of maximizing P, the optimum happens at $E = 56, P = 26$. For the objective of maximizing $2P + E$, the optimum happens at $E = 88.5, P = 19.5$, as drawing the contours indicates (see Figure 2).

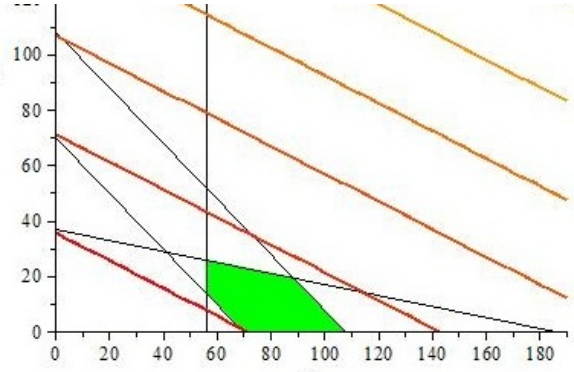


Figure 2: Contours for $2P + E$.

We can use this geometric view to motivate the algorithms.

The Simplex Algorithm: The earliest and most common algorithm in use is called the Simplex method. The idea is to start at some "corner" of the feasible region (to make this easier, we can add in so-called "slack variables" that will drop out when we do our optimization). Then we repeatedly do the following step: look at all neighboring corners of our current position and go to the best one (the one for which the objective function is greatest) if it is better than our current position. Stop when we get to a corner where no neighbor has a higher objective value than we currently have. The key facts here are that

1. since the objective is *linear*, the optimal solution will be at a corner (or maybe multiple corners), and
2. there are no local maxima: if you're *not* optimal, then some neighbor of you must have a strictly larger objective value than you have. That's because the feasible region is *convex*.

So, the Simplex method is guaranteed to halt at the best solution. The problem is that it is possible for there to be an exponential number of corners, and it is possible for Simplex to take an exponential number of steps to find the optimal corner. But, in practice this usually works well.

The Ellipsoid Algorithm: The Ellipsoid Algorithm was invented by Khachiyan in 1980 in Russia. This algorithm solves just the "feasibility problem," but you can then do binary search with the objective function to solve the optimization problem. The idea is to start with a big ellipse (called an ellipsoid in higher dimensions) that we can be sure contains the feasible region. Then, try the center of the ellipse to see if it violates any constraints. If not, you're done. If it does, then look at some constraint violated. So we know the solution (if any) is contained in the remaining at-most-half-ellipse. Now, find a new smaller ellipse that contains that half of our initial ellipse. We then repeat with the new smaller ellipse. One can show that in each step, you can always create a new smaller ellipse whose volume is smaller, by at least a $(1 - 1/n)$ factor, than the original ellipse. So, every n steps, the volume has dropped by about a factor of $1/e$. One can then show that if you ever get *too* small a volume, as a function of the number of bits used in the coefficients of the constraints, then that means there is no solution after all.

One nice thing about the Ellipsoid Algorithm is you just need to tell if the current solution violates any constraints or not, and if so, to produce one. You don't need to explicitly write them all down. There are some problems that you can write as a linear program with an exponential number of constraints if you had to write them down explicitly, but where there is an fast algorithm to determine if a proposed solution violates any constraints and if so to produce one. For these kinds of problems, the Ellipsoid Algorithm is a good one.

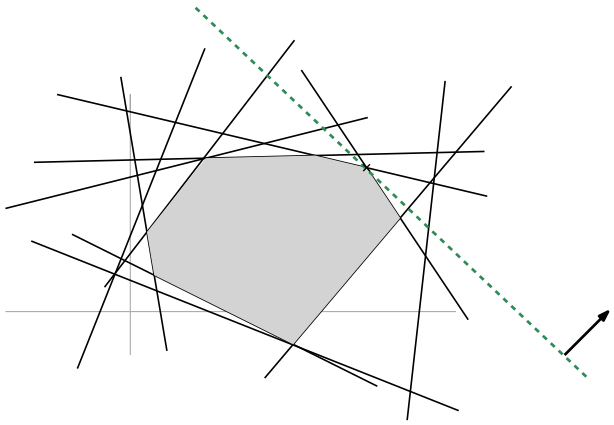
Interior-Point Algorithms: Interior-point Algorithms sort of have aspects of both. They works with feasible points but don't go from corner to corner. Instead they moves inside the interior of the feasible region. One of the first algorithms of this form was developed by Karmarkar in 1984, and now there are a large class of these "interior-point" methods.

The development of better and better algorithms is a big ongoing area of research. In practice, for all of these algorithms, you get a lot of mileage by using good data structures to speed up the time needed for making each decision.

2 Seidel's LP algorithm

Apart from the kinds of algorithms presented above, there are other "geometric" LP solvers. These use the geometry in some crucial way. In the rest of this lecture we now describe a linear-programming algorithm due to Raimund Seidel that solves the 2-dimensional (i.e., 2-variable) LP problem in expected $O(m)$ time (recall, m is the number of constraints), and more generally solves the d-dimensional LP problem in expected time $O(d!m)$.

Setup: We have d variables $x_1, \ldots, x_d$. We are given m linear constraints in these variables $\mathbf{a}_1 \cdot \mathbf{x} \leq b_1, \ldots, \mathbf{a}_m \cdot \mathbf{x} \leq b_m$ along with an objective $\mathbf{c} \cdot \mathbf{x}$ to maximize. (Using boldface to denote vectors.) Our goal is to find a solution $\mathbf{x}$ satisfying the constraints that maximizes the objective. In the example above, the region satisfying all the constraints is given in gray, the arrow indicates the direction in which we want to maximize, and the cross indicates the $\mathbf{x}$ that maximizes the objective.



(You should think of sweeping the green dashed line, to which the vector $\mathbf{c}$ is normal (i.e., perpendicular), in the direction of $\mathbf{c}$, until you reach the last point that satisfies the constraints. This is the point you are seeking.)

The idea: Here is the idea of Seidel's algorithm. [1] Let's add in the (real) constraints one at a

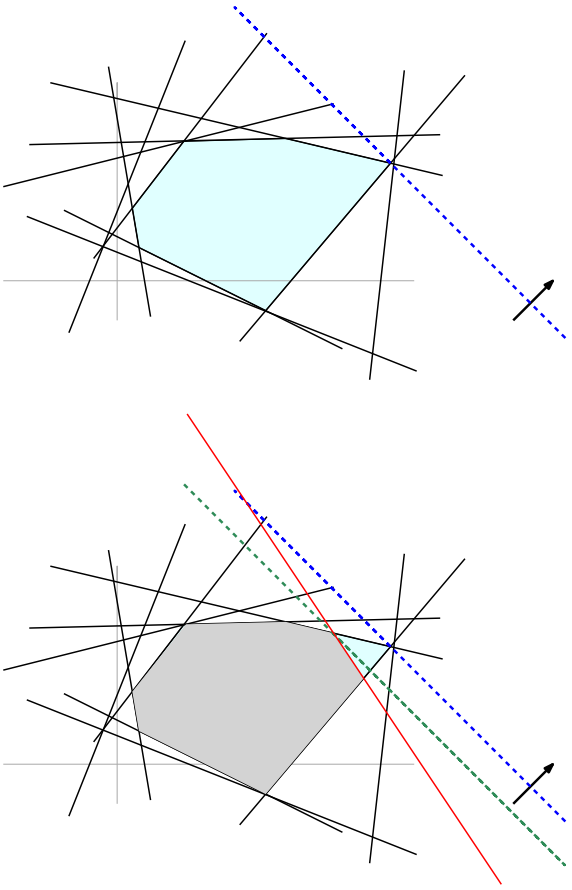
[1] To keep things simple, let's assume that we have inequalities of the form $-\lambda \leq x_i \leq \lambda$ for all i with sufficiently

time, and keep track of the optimal solution for the constraints so far. Suppose, for instance, we have found the optimal solution $\mathbf{x}^*$ for the first $m-1$ constraints, and now we add in the mth constraint $\mathbf{a}_m \cdot \mathbf{x} \leq b_m$. There are two cases to consider:

Case 1: If $\mathbf{x}^*$ satisfies the constraint, then $\mathbf{x}^*$ is still optimal. Time to perform this test: $O(d)$.

Case 2: If $\mathbf{x}^*$ doesn't satisfy the constraint, then the new optimal point will be on the $(d-1)$-dimensional hyperplane $\mathbf{a}_m \cdot \mathbf{x} = b_m$, or else there is no feasible point.

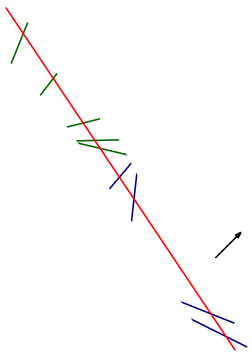
As an example, consider the situation below, before and after we add in the linear constraint $\mathbf{a}_m \cdot \mathbf{x} \leq b_m$ colored in red. This causes the feasible region to change from the light blue region to the smaller gray region, and the optimal point to move.



Let's now focus on the case $d = 2$ and consider the time it takes to handle Case 2 above. With $d = 2$, the hyperplane $\mathbf{a}_m \cdot \mathbf{x} = b_m$ is just a line, and let's call one direction "right" and the other "left". We can now scan through all the other constraints, and for each one, compute its intersection point with this line and whether it is "facing" right or left (i.e., which side of that point satisfies the constraint). We find the rightmost intersection point of all the constraints facing to the right and the leftmost intersection point of all that are facing left. If they cross, then there is no solution. Otherwise, the solution is whichever endpoint gives a better value of $\mathbf{c} \cdot \mathbf{x}$ (if they give

large λ which are separate from the "real" constraints, so that the starting optimal point is one of the corners of the box $[-\lambda, \lambda]^2$. See Section 2.1 for how to remove this assumption.

the same value — i.e., the line $\mathbf{a}_m \cdot \mathbf{x} = b_m$ is perpendicular to $\mathbf{c}$ — then say let's take the rightmost point). In the example above, the 1-dimensional problem is the one in the figure below, with the green constraints "facing" one direction and the blue ones facing the other way. The direction of $\mathbf{c}$ means the optimal point is given by the "lowest" green constraint.



The total time taken here is $O(m)$ since we have $m-1$ constraints to scan through and it takes $O(1)$ time to process each one.

Right now, this looks like an $O(m^2)$-time algorithm for $d = 2$, since we have potentially taken $O(m)$ time to add in a single new constraint if Case 2 occurs. Indeed, that is the worst-case running time for this algorithm. But, suppose we add the constraints in *a random order*? What is the probability that constraint m goes to Case 2?

Notice that the optimal solution to all m constraints (assuming the LP is feasible and bounded) is at a corner of the feasible region, and this corner is defined by two constraints, namely the two sides of the polygon that meet at that point. If both of those two constraints have been seen already, then we are guaranteed to be in Case 1. So, if we are inserting constraints in a random order, the probability we are in Case 2 when we get to constraint m is at most $2/m$. This means that the *expected* cost of inserting the mth constraint is at most:

$$\text{E[cost of inserting } m\text{th constraint]} \leq (1 - 2/m)O(1) + (2/m)O(m) = O(1).$$

This is sometimes called "backwards analysis" since what we are saying is that if we go backwards and pluck out a random constraint from the m we have, the chance it was one of the constraints that mattered was at most $2/m$.

So, Seidel's algorithm is as follows. Place the constraints in a random order and insert them one at a time, keeping track of the best solution so far as above. We just showed that the expected cost of the ith insert is $O(1)$ (or if you prefer, we showed $T(m) = O(1) + T(m-1)$ where $T(i)$ is the expected cost of a problem with i constraints), so the overall expected cost is $O(m)$.

2.1 Handling Special Cases, and Extension to Higher Dimensions*

What if the LP is infeasible? There are two ways we can analyze this. One is that if the LP is infeasible, then it turns out this is determined by at most 3 constraints. So we get the same as above with $2/m$ replaced by $3/m$. Another way to analyze this is imagine we have a separate account we can use to pay for the event that we get to Case 2 and find that the LP is infeasible. Since that can only happen once in the entire process (once we determine the LP is infeasible, we stop), this just provides an additive $O(m)$ term. To put it another way, if the system is infeasible,

then there will be two cases for the final constraint: (a) it was feasible until then, in which case we pay $O(m)$ out of the extra budget (but the above analysis applies to the the (feasible) first $m-1$ constraints), or (b) it was infeasible already in which case we already halted so we pay 0.

What about unboundedness? We had said for simplicity we could put everything inside a bounding box $-\lambda \leq x_i \leq \lambda$. E.g., if all c_i are positive then the initial $\mathbf{x}^* = (\lambda, \ldots, \lambda)$. However, what value of λ should we choose? We could actually do the calculations viewing λ symbolically as a limiting quantity which is arbitrarily large. For example, in 2-dimensions, if $\mathbf{c} = (0, 1)$ and we have a constraint like $2x_1 + x_2 \leq 8$, then we would see it is not satisfied by (λ, λ), and hence intersect the contraint with the box and update to $\mathbf{x}^* = (4 - \lambda/2, \lambda)$.

So far we have shown that for $d = 2$, the expected running time of the algorithm is $O(m)$. For general values of d, there are two main changes. First, the probability that constraint m enters Case 2 is now d/m rather than $2/m$. Second, we need to compute the time to perform the update in Case 2. Notice, however, that this is a $(d-1)$-dimensional linear programming problem, and so we can use the same algorithm recursively, after we have spent $O(dm)$ time to project each of the $m-1$ constraints onto the $(d-1)$-dimensional hyperplane $\mathbf{a}_m \cdot \mathbf{x} = b_m$. Putting this together we have a recurrence for expected running time:

$$T(d, m) \;\leq\; T(d, m-1) + O(d) + \frac{d}{m}[O(dm) + T(d-1, m-1)].$$

This then solves to $T(d, m) = O(d!m)$.