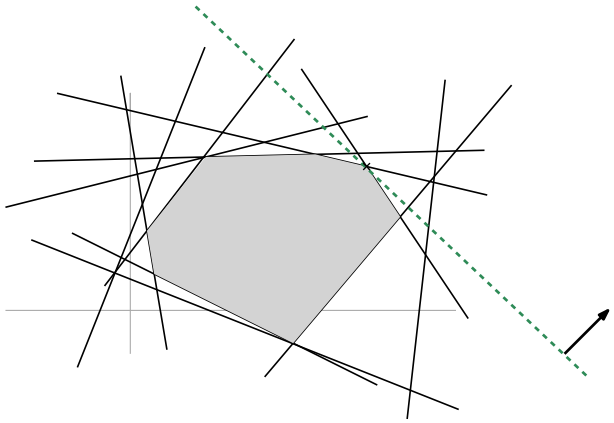
In this lecture we describe a very nice algorithm due to Seidel for Linear Programming in low-dimensional spaces.

1 Seidel's LP algorithm

We now describe a linear-programming algorithm due to Raimund Seidel that solves the 2-dimensional (i.e., 2-variable) LP problem in $O(m)$ time (recall, m is the number of constraints), and more generally solves the d-dimensional LP problem in 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.)

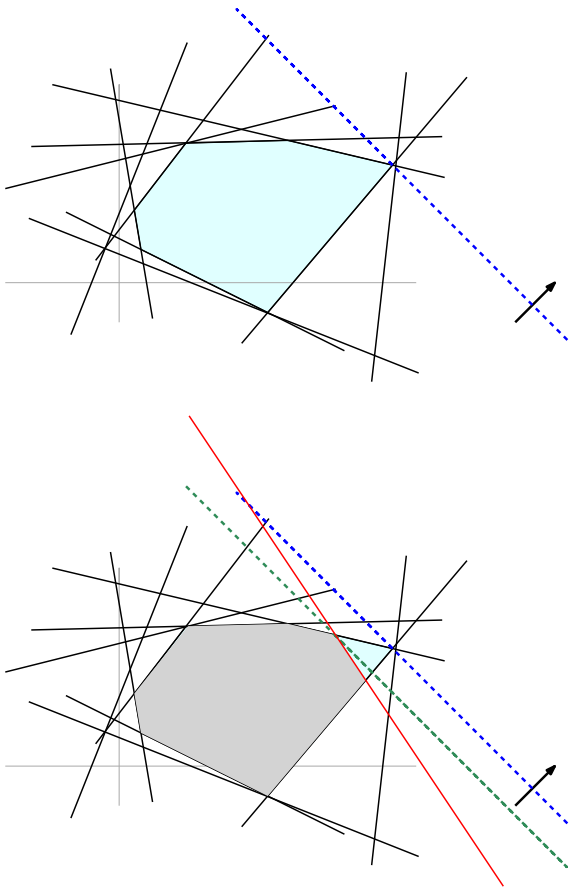
The idea: Here is the idea of Seidel's algorithm. [1] Let's add in the (real) constraints one at a time, and keep track of the optimal solution for the constraints so far. Suppose, for instance, we have found the optimal solution $\mathbf{x}^*$ for the first $m-1$ constraints, and now we add in the mth 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: $O(d)$.

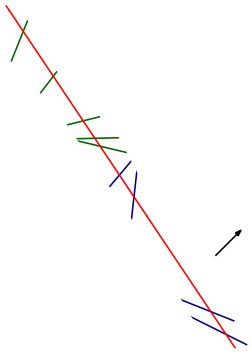
Case 2: If $\mathbf{x}^*$ doesn't satisfy the constraint, then the new optimal point will be on the $(d-1)$-dimensional hyperplane $\mathbf{a}_m \cdot \mathbf{x} = b_m$, or else there is no feasible point.

[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 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 1.1 for how to remove this assumption.

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 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. 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:

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

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 $O(m)$.

1.1 Handling Special Cases, and Extension to Higher Dimensions*

(We will not be testing you on this part, but you should try to understand it all the same.)

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 $T(d, m) = O(d!m)$.