15-451/651: Design & Analysis of Algorithms October 27, 2016
Lecture #17: Linear Programming in 2D last changed: October 30, 2016

In this lecture we 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.

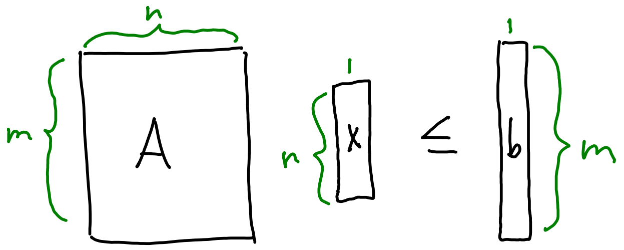
But before we do that we present some linear programming notation and terminology.

1 Standard Linear Programming Terminology and Notation

A linear program (LP) written in the following form is said to be in *Standard Form*:

$$\begin{aligned} \text{maximize } & c^T x \\ \text{subject to } & Ax \le b \\ & x \ge 0 \end{aligned}$$

Here there are n non-negative variables $x_1, x_1, \ldots, x_n$, and m linear constraints encapsulated in the $m \times n$ matrix A and the $m \times 1$ matrix (vector) b. The objective function to be maximized is represented by the $n \times 1$ matrix (vector) c.



Any LP can be expressed in standard form. Example 1: if we are given an LP with some linear equalities, we can split each equality into two inequalities. Example 2: if we need a variable x_i to be allowed to be positive or negative, we replace it by the difference between two variables x_i' and x_i''. Let $x_i = x_i' - x_i''$, and eliminate all occurrences of x in the LP by this substitution.

An LP is *feasible* if there exists a point x satisfying the constraints, and *infeasible* otherwise.

An LP is *unbounded* if $\forall B\ \exists x$ such that x is feasible and $c^T x > B$. Otherwise it is *bounded*.

An LP has an *optimal solution* iff it is feasible and bounded.

The job of an LP solver is to classify a given LP into these categories, and if it is feasible and bounded, it should return a point with the optimum value of the objective function.

2 LP in Two Dimensions

In this lecture we present an algorithm of Raimund Seidel to solve LPs with two variables and m constraints in expected $O(m)$ time.

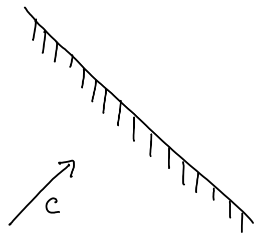
The input to the problem is an LP in the following form:

$$\begin{array}{c}\text{maximize } c^T x \\ \text{subject to } Ax \leq b\end{array}$$

Note that here we don't require that the variables be non-negative. This is strictly more general than standard form. If we do wish the variables to be non-negative, we can simply add those constraints to A (increasing m by two).

We will think about the algorithm geometrically. Each constraint is a half-space. We number the constraints $C_1, C_2, \ldots C_m$. The objective function is a 2D vector, and we are trying to find a point which is farthest in that direction.

To simplify the explanation of the algorithm we will make an additional assumption about the linear system. We assume that there is no constraint which is perpendicular to the objective function. In other words, we don't allow this kind of situation:



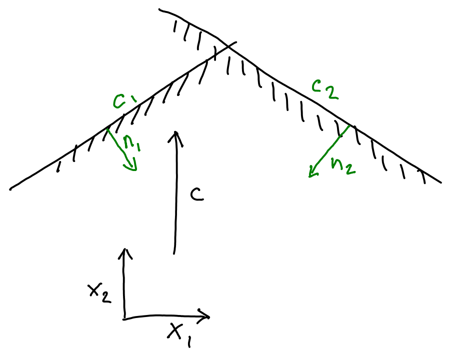
Seidel's 2D LP Algorithm:

Part 1: Getting Started

For the purposes of this part, rotate the entire system so that the c vector points in the positive x_2 direction. (Such a rotation has no effect on the boundedness of the LP.) Now search through all of the constraints, looking for one whose normal (in the direction of the satisfying side of the constraint) is pointing in the negative x_2 direction. If there is no such constraint then the LP is unbounded.

Now suppose there is such a constraint, C_i. Let L_i be the line of points on the boundary of C_i. In one of the two directions along L_i the objective function increases. (Remember the objective function is not perpendicular to L_i.) Now intersect every other constraint C_J with L_i. Case (1): if C_j is parallel to C_i then if their intersection is empty we know the LP is infeasible, so output that. If it's non-empty, then ignore the constraint C_j (it will be processed in part 2 of the algorithm below.) Case (2) L_j and L_i are not parallel. Here we map the constraint C_j onto the line L_i. In one dimension this is simply a point along L_i and a direction from that point. If this 1-D constraint upper bounds the objection function, we stop and return C_i and C_j as the bounding constraints for the objective function.

If we never find such a bounding constraint C_j, this means that the LP is unbounded.



Now reorder the constraints so that the two bounding ones that we just found are numbered C_1 and C_2. The remaining constraints are $C_3, \ldots, C_m$.

Part 2: Finding the Optimum Solution

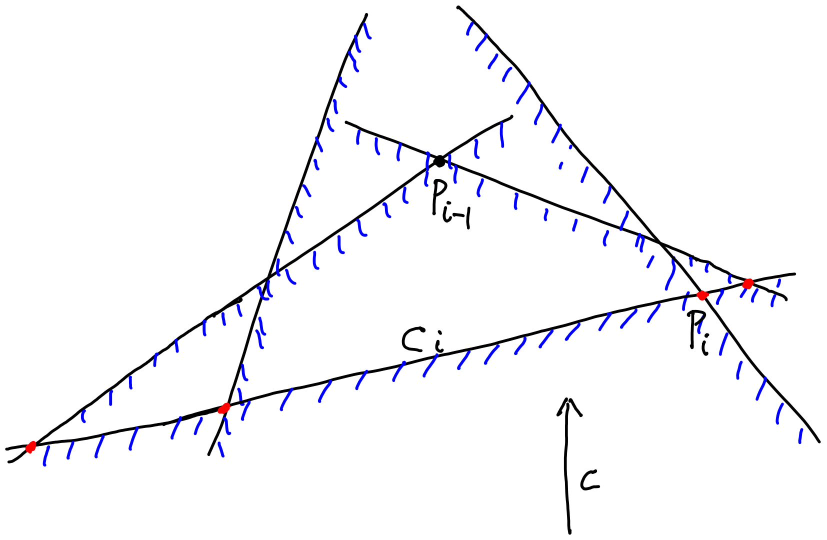
Now randomly permute the remaining constraints $C_3, \ldots, C_m$. We're going to process them in that order. We're going to generate a sequence of points $P_2, P_3, \ldots, P_m$ such that P_i is the optimum solution to the first i constraints. We can immediately compute P_2, which is the intersection point between C_1 and C_2.

For each i from 3 to m do:

Test if P_{i-1} satisfies constraint C_i. If it does, let $P_i = P_{i-1}$ and continue the loop.

So P_{i-1} violates the constraint C_i. Now we try to generate a new point P_i that satisfies all the constraints $C_1, \ldots, C_i$.

Let L_i be the line of the boundary of the constraint C_i. Each of the constraints $C_1, \ldots, C_{i-1}$ that is not parallel to L_i maps to a 1D constraint inside the line L_i. (The ones that are parallel to L_1 are irrelevant.) In addition, the objective function also maps to a direction inside of L_i that optimizes it. (Here again we use the assumption that c is not perpendicular to L_i.) This 1D LP problem is easily solved by constructing the feasible interval. If it contains no points then our LP is infeasible, so return "infeasible". Otherwise take the end of the feasible interval that has the maximum objective function value. This is our point P_i. In the figure below, the intersection between all the prior constraints and L_i are shown in red.



If the loop completes then the optimum solution is P_m. (If it does not complete, then it must have returned "infeasible" already.)

Theorem: Seidel's algorithm runs in expected $O(m)$ time.

Proof: Note that at any point in time we have a point P_i which is the optimum solution to the constraints $C_1, \ldots, C_i$. We also have two constraints among these, call them C_k and C_l, which are the ones that prove the bound on the objective function, and whose intersection point is P_i.

The time it takes to go to compute P_i from P_{i-1} depends on whether or not P_{i-1} satisfies C_i. If it does, then the step is $O(1)$ time. Otherwise the algorithm must look at all the previous constraints and takes $O(i)$ time.

So let's use backward analysis. Suppose we are at a point in time when we just computed P_i as the optimum solution to $C_1, \ldots, C_i$. We now randomly remove one of the constraints $C_3, \ldots, C_i$. What is the probability that P_i differs from P_{i-1}? In order for this to happen we must remove one of the two constraints that constrain the current P_i. What is an upper bound on the probability of this happening?

The probability of this happening is at most $2/(i-2)$. This happens when just two constraints go through P_i, and those are from among $C_3, \ldots, C_i$. In which case the probability of removing one of those two is at most $2/(i-2)$. In all other cases the probability is lower. For example if several constraints conspire to bound P_i. This only lowers the probability that P_i changes. It's also lowered if C_1 and/or C_2 are involved in constraining P_i, cause these will not be removed.

So the cost of computing P_i is at most $2i$ with probability $2/(i-2)$, and 1 with the remaining probability. So the expected cost of the step is at most $2i/(i-2) \leq 6$ because $i \geq 3$. So the expected running time of the algorithm is $O(m)$. QED.

Extending to higher dimensions

This algorithm naturally extends to higher dimensions. Say we're operating in d dimensions. The main loop of the algorithm incorporates a new constraint. When the current point P_i satisfies the new constraint C_i nothing happens. When it violates C_i, then we project all other constraints onto the hyperplane of C_i, and we apply the algorithm in $d-1$ dimensions to solve it (or determine if no solution exists). This lower dimensional optimal solution is projected back up to d dimensions and the algorithm continues. It turns out that the running time is $O(d!m)$.