

18 Linear Programming Algorithms (December 6)

In this lecture, we'll see a few algorithms for actually solving linear programming problems. The most famous of these, the *simplex method*, was proposed by George Dantzig in 1947. Although most variants of the simplex algorithm performs well in practice, there is no sub-exponential upper bound on the running time of any simplex variant. However, if the dimension of the problem is considered a constant, there are several linear programming algorithms that run in *linear* time. I'll describe a particularly simple randomized algorithm due to Raimund Seidel.

My approach to describing these algorithms will rely much more heavily on geometric intuition than the usual linear-algebraic formalism. This works better for me, but your mileage may vary. For a more traditional description of the simplex algorithm, see Robert Vanderbei's excellent textbook *Linear Programming: Foundations and Extensions* [Springer, 2001], which can be freely downloaded (but not legally printed) from the author's website.

18.1 Bases, Feasibility, and Local Optimality

Consider the linear program $\max\{c \cdot x \mid Ax \geq b, x \geq 0\}$, where A is an $n \times d$ constraint matrix, b is an n -dimensional coefficient vector, and c is a d -dimensional objective vector. We will interpret this linear program geometrically as looking for the lowest point in a convex polyhedron in \mathbb{R}^d , described as the intersection of $n + d$ halfspaces. As in the last lecture, we will consider only *non-degenerate* linear programs: At most d constraint hyperplanes pass through any point, and no constraint hyperplane is normal to the objective vector.

A *basis* is a subset of d constraints, which by our non-degeneracy assumption must be linearly independent. The *location* of a basis is the unique point x that satisfies all d constraints with equality; geometrically, x is the unique intersection point of the d hyperplanes. The *value* of a basis is $c \cdot x$, where x is the location of the basis. There are precisely $\binom{n+d}{d}$ bases. Geometrically, the set of constraint hyperplanes defines a decomposition of \mathbb{R}^d into convex polyhedra; this cell decomposition is called the *arrangement* of the hyperplanes. Every d -tuple of hyperplanes (*i.e.*, every basis) defines a *vertex* of this arrangement (the location of the basis). I will use the words 'vertex' and 'basis' interchangeably.

A basis is *feasible* if its location x satisfies all the linear constraints, or geometrically, if the point x is a vertex of the polyhedron.

A basis is *locally optimal* if its location x is the optimal solution to the linear program with the same objective function and *only* the constraints in the basis. Geometrically, a basis is locally optimal if its location x is the lowest point in the intersection of those d halfspaces. A careful reading of the proof of the Strong Duality Theorem reveals that local optimality is the dual equivalent of feasibility; a basis is locally feasible for a linear program Π if and only if the same basis is feasible for the dual linear program Π^* . For this reason, locally optimal bases are sometimes also called *dual feasible*.

Every $(d - 1)$ -tuple of hyperplanes defines a line in \mathbb{R}^d that is broken into *edges* by the other hyperplanes. Most of these edges are line segments joining two vertices, but some are infinite rays, which we think of as segments joined to an artificial vertex called ∞ . This collection of vertices and edges is called the *1-skeleton* of the *graph* of the hyperplane arrangement. The graph of vertices and edges on the boundary of the feasible region is a subgraph of the arrangement graph.

Two bases are said to be *adjacent* if they have $d - 1$ constraints in common; geometrically, two vertices are adjacent if they are joined by an edge in the arrangement graph. A *pivot* changes a basis B into some basis adjacent to B , or equivalently, moves a point from one vertex in the arrangement to an adjacent vertex.

The Weak Duality Theorem implies that the value of every feasible basis is less than or equal to the value of every locally optimal basis; every feasible vertex is higher than every locally optimal vertex. The Strong Duality Theorem implies that (under our non-degeneracy assumption), if a linear program has an optimal solution, it is the *unique* vertex that is *both* feasible and locally optimal. Moreover, the optimal solution is both the lowest feasible vertex and the highest locally optimal vertex.

18.2 The Simplex Algorithm: Primal View

From a geometric standpoint, Dantzig's simplex algorithm is very simple. The input is a set of halfspaces H ; we want the lowest vertex in the intersection of these halfspaces.

```

SIMPLEX( $H$ ):
  if  $\cap H = \emptyset$ 
    return INFEASIBLE
   $x \leftarrow$  any feasible vertex
  while  $x$  is not locally optimal
     $\langle\langle$ pivot downward, maintaining feasibility $\rangle\rangle$ 
     $x \leftarrow$  any feasible neighbor of  $x$  that is lower than  $x$ 
    if  $x = \infty$ 
      return UNBOUNDED
  return  $x$ .

```

Let's ignore the first three lines for the moment. The algorithm maintains a feasible vertex x . At each pivot operation, it moves to a *lower* vertex, so the algorithm never visits the same vertex more than once. Thus, after at most $\binom{n+d}{d}$ pivots, the algorithm either finds the optimal solution, or it discovers that the polyhedron is unbounded.

Notice that we have left open the method for choosing *which* neighbor to choose at each pivot. There are several natural pivoting rules, but for most rules, there are input polyhedra that require an exponential number of pivots. *No* pivoting rule is known that guarantees a polynomial number of pivots in the worst case.

18.3 The Simplex Algorithm: Dual View

We can also geometrically interpret the execution of the simplex algorithm on the dual linear program II. Algebraically, there is no difference between these two algorithms; the only change is in how we interpret the linear algebra geometrically. Again, the input is a set of halfspaces H , and we want the lowest vertex in the intersection of these halfspaces. By the Strong Duality Theorem, this is the same as the highest locally-optimal vertex in the hyperplane arrangement.

```

SIMPLEX( $H$ ):
  if there is no locally optimal vertex
    return UNBOUNDED
   $x \leftarrow$  any locally optimal vertex
  while  $x$  is not feasible
     $\langle\langle$ pivot upward, maintaining local optimality $\rangle\rangle$ 
     $x \leftarrow$  any locally-optimal neighbor of  $x$  that is higher than  $x$ 
    if  $x = \infty$ 
      return INFEASIBLE
  return  $x$ .

```

Let's ignore the first three lines for the moment. The algorithm maintains a locally optimal vertex x . At each pivot operation, it moves to a *higher* vertex, so the algorithm never visits the same vertex more than once. Thus, after at most $\binom{n+d}{d}$ pivots, the algorithm either finds the optimal solution, or it discovers that the linear program is infeasible.

18.4 Computing the Initial Basis

To complete our description of the simplex algorithm, we need to describe how to find the initial vertex x . Our algorithm relies on the following simple observations.

First, the feasibility of a vertex does not depend at all on the choice of objective vector; a vertex is either feasible for every objective function or for none. No matter how we rotate the polyhedron, every feasible vertex stays feasible. Conversely (or by duality, equivalently), the local optimality of a vertex does not depend on the exact location of the d hyperplanes, but only on their normal directions and the objective function. No matter how we translate the hyperplanes, every locally optimal vertex stays locally optimal. In terms of the original matrix formulation, feasibility depends on A and b but not c , and local optimality depends on A and c but not b .

The second important observation is that *every* basis is locally optimal for *some* objective function. Specifically, it suffices to choose any vector that has a positive inner product with each of the normal vectors of the d chosen hyperplanes. Equivalently, we can make *any* basis feasible by translating the hyperplanes appropriately. Specifically, it suffices to translate the chosen d hyperplanes so that they pass through the origin, and then translate all the other halfspaces so that they contain the origin.

Our strategy for finding our initial feasible vertex is to choose *any* vertex, choose a new objective function that makes that vertex locally optimal, and then find the optimal vertex for *that* objective function by running the (dual) simplex algorithm. This vertex must be feasible, even after we restore the original objective function! Equivalently, to find an initial locally optimal vertex, we choose *any* vertex, translate the hyperplanes so that that vertex becomes feasible, and then find the optimal vertex for those translated constraints. This vertex must be locally optimal, even after we restore the hyperplanes to their original locations!

Here are more complete descriptions of the simplex algorithm with this initialization rule, in both primal and dual forms. $\text{Argh}(H)$ denotes the hyperplane arrangement induced by the halfspaces H .

```

SIMPLEX( $H$ ):
   $x \leftarrow$  any vertex
   $\tilde{H} \leftarrow$  rotation of  $H$  that makes  $x$  locally optimal
  while  $x$  is not feasible
     $\langle\langle$  pivot upward maintaining local optimality  $\rangle\rangle$ 
     $x \leftarrow$  any locally optimal neighbor of  $x$  in  $\text{Argh}(\tilde{H})$  that is higher than  $x$ 
    if  $x = \infty$ 
      return INFEASIBLE
  while  $x$  is not locally optimal
     $\langle\langle$  pivot downward, maintaining feasibility  $\rangle\rangle$ 
     $x \leftarrow$  any feasible neighbor of  $x$  in  $\text{Argh}(H)$  that is lower than  $x$ 
    if  $x = \infty$ 
      return UNBOUNDED
  return  $x$ .

```

```

SIMPLEX( $H$ ):
 $x \leftarrow$  any vertex
 $\tilde{H} \leftarrow$  translation of  $H$  that makes  $x$  locally optimal
while  $x$  is not locally optimal
   $\langle\langle$ pivot downward, maintaining feasibility $\rangle\rangle$ 
   $x \leftarrow$  any feasible neighbor of  $x$  in  $\text{Argh}(\tilde{H})$  that is lower than  $x$ 
  if  $x = \infty$ 
    return UNBOUNDED
while  $x$  is not feasible
   $\langle\langle$ pivot upward maintaining local optimality $\rangle\rangle$ 
   $x \leftarrow$  any locally optimal neighbor of  $x$  in  $\text{Argh}(H)$  that is higher than  $x$ 
  if  $x = \infty$ 
    return INFEASIBLE
return  $x$ 

```

18.5 Linear Expected Time for Fixed Dimensions

In most geometric applications of linear programming, the number of variables is a small constant, but the number of constraints may still be very large.

The input to the following algorithm is a set H of n halfspaces and a set B of b hyperplanes. (B stands for *basis*.) The algorithm returns the lowest point in the intersection of the halfspaces in H and the hyperplanes B . At the top level of recursion, B is empty. I implicitly assume here that the linear program is bounded; if necessary, we can guarantee boundedness by adding a single halfspace to H . A point x *violates* a constraint h if it is not contained in the corresponding halfspace.

```

SEIDELLP( $H, B$ ):
if  $|B| = d$ 
   $x \leftarrow \bigcap B$ 
  if  $x$  violates any constraint in  $H$ 
    return INFEASIBLE
  else
    return  $x$ 
if  $|H \cup B| = d$ 
  return  $\bigcap(H \cup B)$ 
 $h \leftarrow$  random element of  $H$ 
 $x \leftarrow$  SEIDELLP( $H \setminus h, B$ )    (*)
if  $x = \text{INFEASIBLE}$ 
  return  $x$ 
else if  $x$  violates  $h$ 
  return SEIDELLP( $H \setminus h, B \cup \partial h$ )
else
  return  $x$ 

```

The point x recursively computed in line (*) is not the optimal solution precisely when the random halfspace h is one of the d halfspaces that define the optimal solution. In other words, the probability of calling SEIDELLP($H, B \cup h$) is exactly $(d - b)/n$. Thus, we have the following recurrence for the expected number of recursive calls for this algorithm:

$$T(n, b) = \begin{cases} 1 & \text{if } b = d \text{ or } n + b = d \\ T(n - 1, b) + \frac{d - b}{n} \cdot T(n - 1, b + 1) & \text{otherwise} \end{cases}$$

The recurrence is somewhat simpler if we let $\delta = d - b$:

$$T(n, \delta) = \begin{cases} 1 & \text{if } \delta = 0 \text{ or } n = \delta \\ T(n-1, \delta) + \frac{\delta}{n} \cdot T(n-1, \delta-1) & \text{otherwise} \end{cases}$$

It's easy to prove by induction that $T(n, \delta) = O(\delta! n)$:

$$\begin{aligned} T(n, \delta) &= T(n-1, \delta) + \frac{\delta}{n} \cdot T(n-1, \delta-1) \\ &\leq \delta! (n-1) + \frac{\delta}{n} (\delta-1)! \cdot (n-1) && \text{[ind. hyp.]} \\ &= \delta! (n-1) + \delta! \frac{n-1}{n} \\ &\leq \delta! n \end{aligned}$$

At the top level of recursion, we perform one violation test in $O(d)$ time. In each of the base cases, we spend $O(d^3)$ time computing the intersection point of d hyperplanes, and in the first base case, we spend $O(dn)$ additional time testing for violations. More careful analysis implies that the algorithm runs in $\boxed{O(d! \cdot n)}$ expected time.