Simplicibus itaque verbis gaudet Mathematica Veritas, cum etiam per se simplex sit Veritatis oratio. [And thus Mathematical Truth prefers simple words, because the language of Truth is itself simple.]

— Tycho Brahe (quoting Seneca (quoting Euripides))
Epistolarum astronomicarum liber primus (1596)

When a jar is broken, the space that was inside
Merges into the space outside.
In the same way, my mind has merged in God;
To me, there appears no duality.

— Sankara, *Viveka-Chudamani* (c. 700), translator unknown

CHAPTER I

Simplex Algorithm(s)

[Read Chapter H first.]

In this chapter I will describe several variants of the *simplex algorithm* for solving linear programming problems, first proposed by George Dantzig in 1947. Although most variants of the simplex algorithm perform well in practice, no deterministic simplex variant is known to run in sub-exponential time in the worst case.[1] However, if the dimension of the problem is considered a constant, there are several variants of the simplex algorithm that run in *linear* time. I'll describe a particularly simple randomized algorithm due to Raimund Seidel.

My approach to describing these algorithms relies 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.

[1]However, there are *randomized* variants of the simplex algorithm that run in subexponential *expected* time, most notably the RandomFacet algorithm analyzed by Gil Kalai in 1992, and independently by Jiří Matoušek, Micha Sharir, and Emo Welzl in 1996. No randomized variant is known to run in polynomial time. In particular, in 2010, Oliver Friedmann, Thomas Dueholm Hansen, and Uri Zwick proved that the worst-case expected running time of RandomFacet is superpolynomial.

© Copyright 2017 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms for the most recent revision.

I.1 Bases, Feasibility, and Local Optimality

Consider the canonical linear program $\max\{c \cdot x \mid Ax \leq b, x \geq 0\}$, 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: Every subset of d constraint hyperplanes intersects in a single point; at most d constraint hyperplanes pass through any point; and objective vector is linearly independent from any $d - 1$ constraint vectors.

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 subset of d hyperplanes (that is, 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. If there are no feasible bases, the linear program is ***infeasible***.

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 ⨿. For this reason, locally optimal bases are sometimes also called ***dual feasible***. If there are no locally optimal bases, the linear program is ***unbounded***.[2]

Two bases are ***neighbors*** if they have $d - 1$ constraints in common. Equivalently, in geometric terms, two vertices are neighbors if they lie on a *line* determined by some $d-1$ constraint hyperplanes. Every basis is a neighbor of exactly dn other bases; to change a basis into one of its neighbors, there are d choices for which constraint to remove and n choices for which constraint to add. The graph of vertices and edges on the boundary of the feasible polyhedron is a subgraph of the basis graph.

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; equivalently, every feasible vertex is higher than every locally optimal vertex. The Strong Duality Theorem implies that

[2] For non-degenerate linear programs, the feasible region is unbounded in the objective direction if and only if no basis is locally optimal. However, there are degenerate linear programs with no locally optimal basis that are infeasible.

(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.

I.2 The Simplex Algorithm

Primal: Falling Marbles

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

```
PRIMALSIMPLEX(H):
  if ∩H = ∅
      return INFEASIBLE
  x ← any feasible vertex
  while x is not locally optimal
      ⟨⟨pivot downward, maintaining feasibility⟩⟩
      if every feasible neighbor of x is higher than x
          return UNBOUNDED
      else
          x ← any feasible neighbor of x that is lower than x
  return x
```

Let's ignore the first three lines for the moment. The algorithm maintains a feasible vertex x. At each so-called ***pivot*** operation, the algorithm moves to a *lower* vertex, so the algorithm never visits the same vertex more than once. Thus, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the feasible vertex x is locally optimal, and therefore the optimum vertex, or the feasible vertex x is not locally optimal but has no lower feasible neighbor, in which case the feasible region must be unbounded.

Notice that we have not specified *which* neighbor to choose at each pivot. Many different pivoting rules have been proposed, but for almost every known pivot rule, there is an input polyhedron that requires an exponential number of pivots under that rule. *No* pivoting rule is known that guarantees a polynomial number of pivots in the worst case, or even in expectation.[3]

Dual: Rising Bubbles

We can also geometrically interpret the execution of the simplex algorithm on the dual linear program II. Again, the input is a set H of halfspaces, and we want the lowest

[3]In 1957, Hirsch conjectured that for *any* linear programming instance with d variables and $n+d$ constraints, starting at any feasible basis, there is a sequence of ***at most n*** pivots that leads to the optimal basis. This long-standing conjecture was finally disproved in 2010 by Fransisco Santos, who described an counterexample with 43 variables and 86 constraints, where the worst-case number of required pivots is 44.

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.

```
DualSimplex(H):
  if there is no locally optimal vertex
      return Unbounded
  x ← any locally optimal vertex
  while x is not feasbile
      ⟨⟨pivot upward, maintaining local optimality⟩⟩
      if every locally optimal neighbor of x is lower than x
          return Infeasible
      else
          x ← any locally-optimal neighbor of x that is higher than x
  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, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the locally optimal vertex x is feasible, and therefore the optimum vertex, or the locally optimal vertex x is not feasible but has no higher locally optimal neighbor, in which case the problem must be infeasible.

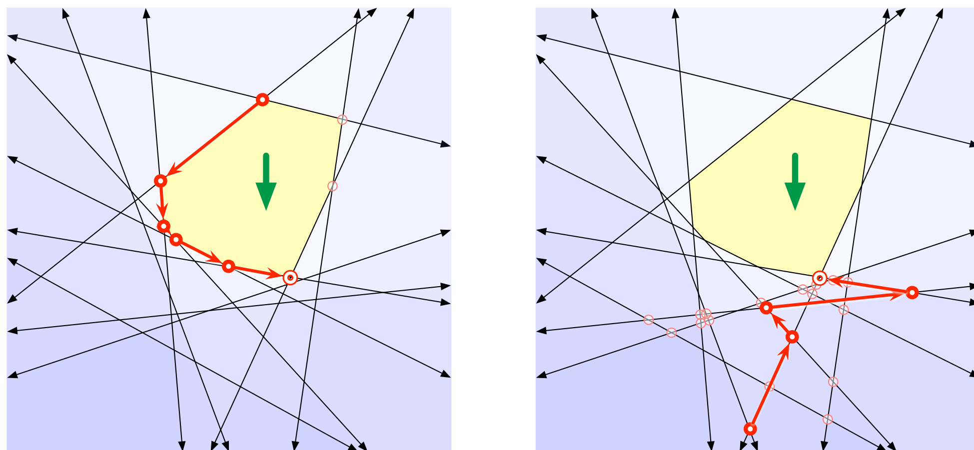


Figure I.1. The primal simplex (falling marble) and dual simplex (rising bubble) algorithms in action.

From the standpoint of linear algebra, there is absolutely no difference between running PrimalSimplex on any linear program Π and running DualSimplex on the dual linear program $\amalg$. The actual *code* is identical. The only difference between the two algorithms is how we interpret the linear algebra geometrically.

I.3 Computing the Initial Basis

To complete our description of the simplex algorithm, we need to describe how to find the initial vertex x in the third line of PrimalSimplex or DualSimplex. There are several methods to find feasible or locally optimal bases, but perhaps the most natural method uses the simplex algorithm itself. Our approach relies on two simple observations.

First, the feasibility of a vertex does not depend on the choice of objective vector; a vertex is either feasible for every objective function or for none. Equivalently (by duality), 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; a vertex is either locally optimal for every translation of the hyperplanes or for none. 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.

Second, *every* basis is locally optimal for *some* objective vector. 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, *every* basis is feasible for some offset vector. Specifically, it suffices to translate the d chosen hyperplanes so that they pass through the origin, and then translate all other halfspaces so that they strictly contain the origin.

Our strategy for finding an initial feasible vertex for the primal simplex algorithm 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 for the dual simplex algorithm, we choose *any* vertex, translate the hyperplanes so that that vertex becomes feasible, and then find the optimal vertex for those translated constraints using the (primal) simplex algorithm. This vertex must be locally optimal, even after we restore the hyperplanes to their original locations!

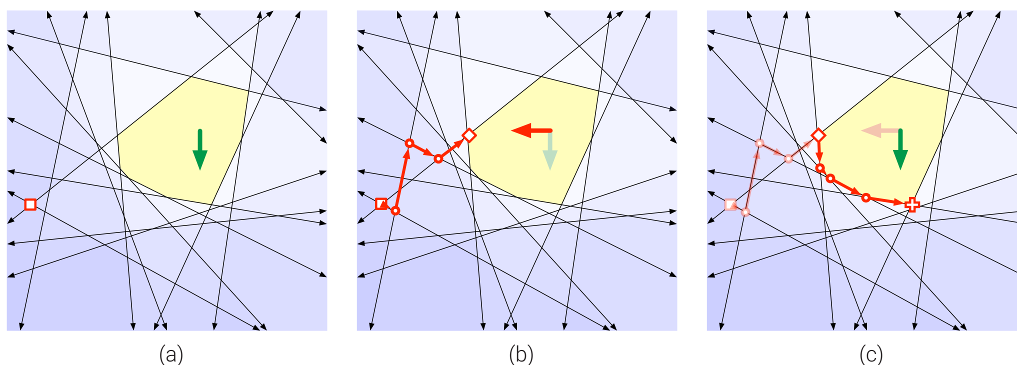


Figure I.2. Primal simplex with dual initialization: (a) Choose any basis. (b) Rotate the objective to make the basis locally optimal, and pivot "up" to a feasible basis. (c) Pivot down to the optimum basis for the original objective.

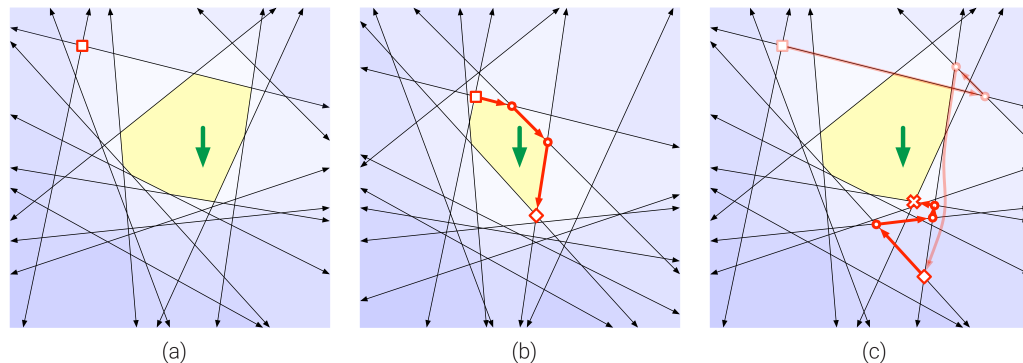


Figure I.3. Dual simplex with primal optimization: (a) Choose any basis. (b) Translate the constraints to make the basis feasible, and pivot down to a locally optimal basis. (c) Pivot up to the optimum basis for the original constraints.

Pseudocode for both algorithms is given in Figures I.4 and I.5. As usual, the input to both algorithms is a set H of halfspaces, and the algorithms either return the lowest vertex in the intersection of those halfspaces, report that the linear program is infeasible, or report that the linear program is unbounded.

```
DUALPRIMALSIMPLEX(H):
  x ← any vertex
  H̃ ← any rotation of H that makes x locally optimal

  while x is not feasible
      if every locally optimal neighbor of x is lower (wrt H̃) than x
          return INFEASIBLE
      else
          x ← any locally optimal neighbor of x that is higher (wrt H̃) than x

  while x is not locally optimal
      if every feasible neighbor of x is higher than x
          return UNBOUNDED
      else
          x ← any feasible neighbor of x that is lower than x
  return x
```

Figure I.4. The primal simplex algorithm with dual initialization.

I.4 Network Simplex

Our first natural examples of linear programming problems were shortest paths, maximum flows, and minimum cuts in edge-weighted graphs. It is instructive to reinterpret the behavior of the abstract simplex algorithm in terms of the original input graphs; this reinterpretation allows for a much more efficient implementation of the simplex algorithm, which is normally called ***network simplex***.

```
PrimalDualSimplex(H):
  x ← any vertex
  H̃ ← any translation of H that makes x feasible
  while x is not locally optimal
      if every feasible neighbor of x is higher (wrt H̃) than x
          return Unbounded
      else
          x ← any feasible neighbor of x that is lower (wrt H̃) than x
  while x is not feasible
      if every locally optimal neighbor of x is lower than x
          return Infeasible
      else
          x ← any locally-optimal neighbor of x that is higher than x
  return x
```

Figure I.5. The dual simplex algorithm with primal initializzation.

As a concrete working example, let's consider a special case of the minimum-cost flow problem called the ***transshipment problem***. The input consists of a directed graph $G = (V, E)$, a ***balance*** function $b : V \to \mathbb{R}$, and a ***cost*** function $\$: E \to \mathbb{R}$, but no capacities or lower bounds on the edges. Our goal is to compute a ***flow*** function $f : E \to \mathbb{R}$ that is non-negative everywhere, satisfies the balance constraint

$$\sum_{u \to v} f(u \to v) - \sum_{v \to w} f(v \to w) = b(v)$$

at every vertex v, and minimizes the total cost $\sum_e f(e) \cdot \$(e)$.

We can easily express this problem as a linear program with a variable for each edge and constraints for each vertex and edge.

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{u \to v} \$(u \to v) \cdot f(u \to v) & \\ \text{subject to} & \displaystyle\sum_{u \to v} f(u \to v) - \sum_{v \to w} f(v \to w) = b(v) & \text{for every vertex } v \neq s \\ & f(u \to v) \geq 0 & \text{for every edge } u \to v \end{array}$$

Here I've omitted the balance constraint for some fixed vertex s, because it is redundant; if f is balanced at every other vertex, then f must be balanced at s as well. By interpreting the balance, cost, and flow functions as vectors, we can write this linear program more succinctly as follows:

$$\boxed{\begin{array}{rl} \max & \$ \cdot f \\ \text{s.t.} & Af = b \\ & f \geq 0 \end{array}}$$

Here A is the ***vertex-edge incidence matrix*** of G; this matrix has one row for each edge and one column for each vertex, and whose entries are defined as follows:

$$A(x{\to}y, v) = \begin{cases} 1 & \text{if } v = y \\ -1 & \text{if } v = x \\ 0 & \text{otherwise} \end{cases}$$

Let $\overline{G} = (V, \overline{E})$ be the undirected version of G, defined by setting $\overline{E} = \{uv \mid u{\to}v \in E\}$. In the following arguments, I will refer to "undirected cycles" and "spanning trees" in G; these phrases are shorthand for the subset of directed edges in G corresponding to undirected cycles and spanning trees in $\overline{G}$.

To simplify the remaining presentation, I will make two non-degeneracy assumptions:

- The cost vector $\$$ is non-degenerate: No residual cycle has cost 0.
- The balance vector is non-degenerate: No non-empty proper subset of vertices has total balance 0.

Because the transshipment LP has E variables, a basis consists of E linearly independent constraints. We call a basis ***balanced*** if it contains all $V-1$ balance constraints; any flow consistent with a balanced basis is balanced at *every* vertex of G. Every balanced basis contains exactly $E-V+1$ edge constraints, and therefore *omits* exactly $V-1$ edge constraints. We call an edge *fixed* if its constraint is included in the basis and *free* otherwise. Any flow consistent with a balanced basis is zero on every fixed edge and non-negative on every free edge.

Lemma 1. *For every balanced basis, the free edges define a spanning tree of G; conversely, for every spanning tree T of G, there is a balanced basis for which T is the set of free edges.*[4]

Proof: First, fix an arbitrary balanced basis, let f be any flow consistent with that basis, and let T be the set of $V-1$ free edges for that basis. (The flow f need not be feasible.) For the sake of argument, suppose T contains an undirected cycle. Then by pushing flow around that cycle, we can obtain another (not necessarily feasible) flow f' that is still consistent with our fixed basis. So the basis constraints do not determine a unique flow, which means the constraints are not linearly independent, contradicting the definition of basis. We conclude that T is acyclic, and therefore defines a spanning tree of G.

On the other hand, suppose T is an arbitrary spanning tree of G. We define a function $\mathit{flow}_T : E \to \mathbb{R}$ as follows:

- For each edge $u{\to}v \in T$, we define $\mathit{flow}_T(u{\to}v)$ to be sum of balances in the component of $T \setminus u{\to}v$ that contains v. Our non-degeneracy assumption implies that $\mathit{flow}_T(u{\to}v) \neq 0$.
- For each edge $u{\to}v \notin T$, we define $\mathit{flow}_T(u{\to}v) = 0$.

[4]More generally, every basis (balanced or not) is associated with a spanning *forest* F; the basis contains edge constraints for every edge *not* in F and all but one vertex constraint in each component of F.

Routine calculations imply $flow_T$ is balanced at every vertex; moreover, $flow_T$ is the *unique* flow in G that is non-zero only on edges of T. We conclude that the $V-1$ balance constraints and the $E-V+1$ edge constraints for edges not in T are linearly independent; in other words, T is the set of free edges of a balanced basis. □

For any spanning tree T and any edges $u{\to}v \notin T$, let $\boldsymbol{cycle_T(u{\to}v)}$ denote the directed cycle consisting of $u{\to}v$ and the unique residual path in T from v to u. Our non-degeneracy assumption implies that the total cost $\$(cycle_T(u{\to}v))$ of this cycle is not equal to zero. We define the ***slack*** of each edge in G as follows:

$$slack_T(u{\to}v) := \begin{cases} 0 & \text{if } u{\to}v \in T \\ \$(cycle_T(u{\to}v)) & \text{if } u{\to}v \notin T \end{cases}$$

The function $flow_T : E \to \mathbb{R}$ is the location of the balanced basis associated with T; the function $slack_T : E \to \mathbb{R}$ is essentially the location of the corresponding *dual* basis. With these two functions in hand, we can characterize balanced bases as follows:

- The basis associated with any spanning tree T is *feasible* (and thus the dual basis is locally optimal) if and only if $flow_T(e) \geq 0$ (and therefore $flow_T(e) > 0$) for every edge $e \in T$.
- The basis associated with any spanning tree T is *locally optimal* (and thus the dual basis is feasible) if and only if $slack_T(e) \geq 0$ (and therefore $slack_T(e) > 0$) for every edge $e \notin T$.

Notice that the complementary slackness conditions are automatically satisfied: For any edge e, and for any spanning tree T, we have $flow_T(e) \cdot slack_T(e) = 0$. In particular, if T is the optimal basis, then either $flow_T(e) > 0$ and $slack_T(e) = 0$, or $flow_T(e) = 0$ and $slack_T(e) > 0$.

A pivot in the simplex algorithm modifies the current basis by removing one constraint and adding another. For the transshipment LP, a pivot modifies a spanning tree T by adding an edge $e_{\text{in}} \notin T$ and removing an edge $e_{\text{out}} \in T$ to obtain a new spanning tree T'.

- The leaving edge e_{out} must lie in the unique residual cycle in $T + e_{\text{in}}$. The pivot modifies the flow function by pushing flow around the unique residual cycle in $T + e_{\text{in}}$, so that some edge e_{out} becomes empty. In particular, the pivot decreases the overall cost of the flow by $flow_T(e_{\text{out}}) \cdot slack_T(e_{\text{in}})$.
- Equivalently, the entering edge e_{in} must have one endpoint in each component of $T - e_{\text{out}}$. Let S be the set of vertices in the component of $T - e_{\text{out}}$ containing the tail of e_{out}. The pivot subtracts $slack_T(e_{\text{in}})$ from the slack of every edge from S to $V \setminus S$, and adds $slack_T(e_{\text{in}})$ to the slack of every edge from $V \setminus S$ to S.

The primal simplex algorithm starts with an arbitrary feasible basis and then repeatedly pivots to a new feasible basis with smaller cost. For the transshipment LP, we can find an initial feasible flow using the FeasibleFlow algorithm from Chapter F. Each primal simplex pivot finds an edge e_{in} with negative slack and pushes flow around $cycle_T(e_{\text{in}})$

until some edge e_{out} is saturated. In other words, *the primal network simplex algorithm is an implementation of cycle cancellation*.

The dual simplex algorithm starts with an arbitrary locally optimal basis and then repeatedly pivots to a new locally optimal basis with larger cost. For the transshipment LP, the shortest-path tree rooted at any vertex provides a locally optimal basis. Each pivot operation finds an edge e_{out} with negative flow, removes it from the current spanning tree, and then adds the edge e_{in} whose slack is as small as possible.

I'm not happy with this presentation. I really need to reformulate the dual LP in terms of slacks, instead of the standard "distances", so that I can talk about pushing slack across cuts, just like pushing flow around cycles. This might be helped by a general discussion of cycle/circulation and cut/cocycle spaces of G: (1) orthogonal complementary subspaces of the edge/pseudoflow space of G, (2) generated by fundamental cycles and fundamental cuts of any spanning tree of G. Also, this needs examples/figures.

I.5 Linear Expected Time for Fixed Dimensions

This section needs careful revision.

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 will implicitly assume that the linear program is both feasible and bounded. (If necessary, we can guarantee boundedness by adding a single halfspace to H, and we can guarantee feasibility by adding a dimension.) A point x ***violates*** a constraint h if it is not contained in the corresponding halfspace.

```
SEIDELLP(H, B):
  if |B| = d
      return ⋂B
  if |H ∪ B| = d
      return ⋂(H ∪ B)
  h ← random element of H
  x ← SEIDELLP(H \ h, B)        (∗)
  if x violates h
      return SEIDELLP(H \ h, B ∪ ∂h)
  else
      return x
```

The point x recursively computed in line $(*)$ is the optimal solution if and only if the random halfspace h is *not* 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:

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

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

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

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

$$\begin{aligned} T(n,\delta) &= T(n-1,\delta) + \frac{\delta}{n} \cdot T(n-1,\delta-1) \\ &\le \delta!\,(n-1) + \frac{\delta}{n}(\delta-1)! \cdot (n-1) && \text{[induction hypothesis]} \\ &= \delta!\,(n-1) + \delta!\frac{n-1}{n} \\ &\le \delta!\, n \end{aligned}$$

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 ***$O(d! \cdot n)$ expected time***.

Exercises

1. Fix a non-degenerate linear program in canonical form with d variables and $n+d$ constraints.

   (a) Prove that every *feasible* basis has exactly d *feasible* neighbors.

   (b) Prove that every *locally optimal* basis has exactly n *locally optimal* neighbors.

2. (a) Give an example of a non-empty polyhedron $Ax \le b$ that is unbounded for *every* objective vector c.

   (b) Give an example of an infeasible linear program whose dual is also infeasible.

   In both cases, your linear program will be degenerate.

3. Describe and analyze an algorithm that solves the following problem in $O(n)$ time: Given n red points and n blue points in the plane, either find a line that separates every red point from every blue point, or prove that no such line exists.

4. In this exercise, we develop another standard method for computing an initial feasible basis for the primal simplex algorithm. Suppose we are given a canonical linear program Π with d variables and $n+d$ constraints as input:

$$\begin{array}{ll} \max & c \cdot x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0 \end{array}$$

To compute an initial feasible basis for Π, we solve a modified linear program Π' defined by introducing a new variable λ and two new constraints $0 \leq \lambda \leq 1$, and modifying the objective function:

$$\begin{array}{ll} \max & \lambda \\ \text{s.t.} & Ax - b\lambda \leq 0 \\ & \lambda \leq 1 \\ & x, \lambda \geq 0 \end{array}$$

   (a) Prove that $x_1 = x_2 = \cdots = x_d = \lambda = 0$ is a feasible basis for Π'.

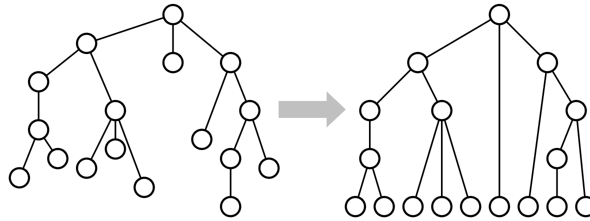
   (b) Prove that Π is feasible if and only if the optimal value for Π' is 1.

   (c) What is the dual of Π'?

5. Suppose you have a subroutine that can solve linear programs in polynomial time, but only if they are both feasible and bounded. Describe an algorithm that solves *arbitrary* linear programs in polynomial time. Your algorithm should return an optimal solution if one exists; if no optimum exists, your algorithm should report that the input instance is UNBOUNDED or INFEASIBLE, whichever is appropriate. *[Hint: Add one variable and one constraint.]*

6. Suppose your are given a rooted tree T, where every edge e has two associated values: a non-negative *length* $\ell(e)$, and a *cost* $\$(e)$ (which may be positive, negative, or zero). Your goal is to add a non-negative *stretch* $s(e) \geq 0$ to the length of every edge e in T, subject to the following conditions:

   - Every root-to-leaf path π in T has the same total stretched length $\sum_{e\in\pi}(\ell(e)+s(e))$
   - The total *weighted* stretch $\sum_e s(e) \cdot \$(\ell)$ is as small as possible.



(a) Give a concise linear programming formulation of this problem.

(b) Prove that in any optimal solution to this problem, we have $s(e) = 0$ for every edge on some longest root-to-leaf path in T. In other words, prove that the optimally stretched tree has the same depth as the input tree. *[Hint: What is a basis in your linear program? When is a basis feasible?]*

(c) Describe and analyze an algorithm that solves this problem in $O(n)$ time. Your algorithm should either compute the minimum total weighted stretch, or report correctly that the total weighted stretch can be made arbitrarily negative.

7. Recall that the single-source shortest path problem can be formulated as a linear programming problem, with one variable d_v for each vertex $v \neq s$ in the input graph, as follows:

$$\begin{array}{lrll} \text{maximize} & \sum_v d_v & \\ \text{subject to} & d_v \leq \ell_{s\to v} & \text{for every edge } s \to v \\ & d_v - d_u \leq \ell_{u\to v} & \text{for every edge } u \to v \text{ with } u \neq s \\ & d_v \geq 0 & \text{for every vertex } v \neq s \end{array}$$

This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of distances. Assume that the edge weights $\ell_{u\to v}$ are all non-negative and that there is a unique shortest path between any two vertices in the graph.

(a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?

(b) Show that in the optimal basis, every variable d_v is equal to the shortest-path distance from s to v.

(c) Describe the primal simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?

(d) Describe the dual simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?

(e) Is Dijkstra's algorithm an instance of network simplex? Is Shimbel-Bellman-Ford? Justify your answers.

(f) Using the results in problem 9, prove that if the edge lengths $\ell_{u\to v}$ are all integral, then the optimal distances d_v are also integral.

8. The maximum (s,t)-flow problem can be formulated as a linear programming problem, with one variable $f_{u\to v}$ for each edge $u \to v$ in the input graph:

$$\begin{array}{lll} \text{maximize} & \sum_w f_{s\to w} - \sum_u f_{u\to s} & \\ \text{subject to} & \sum_w f_{v\to w} - \sum_u f_{u\to v} = 0 & \text{for every vertex } v \neq s,t \\ & f_{u\to v} \leq c_{u\to v} & \text{for every edge } u \to v \\ & f_{u\to v} \geq 0 & \text{for every edge } u \to v \end{array}$$

This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of flows.

(a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?

(b) Show that the optimal basis represents a maximum flow.

(c) Describe the primal simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?

(d) Describe the dual simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?

(e) Is the Ford-Fulkerson augmenting-path algorithm an instance of network simplex? Justify your answer. *[Hint: There is a one-line argument.]*

(f) Using the results in problem 9, prove that if the capacities $c_{u\to v}$ are all integral, then the maximum flow values $f_{u\to v}$ are also integral.

9. A ***minor*** of a matrix A is the submatrix defined by any subset of the rows and any subset of the columns. A matrix A is ***totally unimodular*** if, for every square minor M, the determinant of M is -1, 0, or 1.

(a) Let A be an arbitrary totally unimodular matrix.

i. Prove that the transposed matrix $A^\top$ is also totally unimodular.

ii. Prove that negating any row or column of A leaves the matrix totally unimodular.

iii. Prove that the block matrix $[A \mid I]$ is totally unimodular.

(b) Prove that for any totally unimodular matrix A and any integer vector b, the canonical linear program $\max\{c \cdot x \mid Ax \leq b, x \geq 0\}$ has an *integer* optimal solution. *[Hint: Cramer's rule.]*

(c) The ***unsigned incidence matrix*** of an undirected graph $G = (V, E)$ is an $|V| \times |E|$ matrix A, with rows indexed by vertices and columns indexed by edges, where for each row v and column uw, we have

$$A[v, uw] = \begin{cases} 1 & \text{if } v = u \text{ or } v = w \\ 0 & \text{otherwise} \end{cases}$$

Prove that the unsigned incidence matrix of every *bipartite* graph G is totally unimodular. *[Hint: Each square minor corresponds to a subgraph H of G with k vertices and k edges, for some integer k. Argue that at least one of the following statements must be true: (1) H is disconnected; (2) H has a vertex with degree 1; (3) H is an even cycle.]*

(d) Prove for every *non*-bipartite graph G that the unsigned incidence matrix of G is *not* totally unimodular. *[Hint: Consider any odd cycle.]*

(e) The ***signed incidence matrix*** of a directed graph $G = (V, E)$ is also an $|V| \times |E|$ matrix A, with rows indexed by vertices and columns indexed by edges, where for each row v and column $u{\rightarrow}w$, we have

$$A[u, v{\rightarrow}w] = \begin{cases} 1 & \text{if } v = w \\ -1 & \text{if } v = u \\ 0 & \text{otherwise} \end{cases}$$

Prove that the signed incidence matrix of every directed graph G is totally unimodular.

© Copyright 2017 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://jeffe.cs.illinois.edu/teaching/algorithms for the most recent revision.