



LinearProgramming

[FrontPage] [TitleIndex] [WordIndex]

Note: You are looking at a static copy of the former PineWiki site, used for class notes by James Aspnes from 2003 to 2012. Many mathematical formulas are broken, and there are likely to be other bugs as well. These will most likely not be fixed. You may be able to find more up-to-date versions of some of these notes at <http://www.cs.yale.edu/homes/aspnes/#classes>.

A **linear program** consists of a collection of linear inequalities in a set of variables, together with a linear objective function to maximize (or minimize). A simple linear program might look like:

```
maximize
  x + z
subject to
  x <= 12
  y <= 14
  x >= 0
  y >= 0
  -y + z = 4
  2x - 3y >= 5
```

The *solution* to a linear program is an assignment to the variables that satisfies all the constraints while maximizing (or minimizing) the objective function; for example, the above linear program has solution $x = 12$, $y = 19/3$, $z = 31/3$, which produces an objective function value of $67/3$.

1. General vs canonical form

A linear program in **general form** permits \leq , \geq and $=$ constraints, allows the objective function to be maximized or minimized, and puts no constraints on the values of the variables other than the constraints that appear in the program. A linear program in **canonical form** permits only \leq constraints, requires that the objective function be maximized, and requires that all variables be non-negative. There is no loss of generality in moving from general to canonical form, because we can always (a) replace any constraint of the form $ax \geq b$ with $-ax \leq -b$ and any constraint of the form $ax = b$ with two constraints $ax \leq b$ and $-ax \leq -b$; (b) change a minimization problem to a maximization problem by negating the objective function; and (c) replace any unconstrained variable x with the difference between two new variables $x^+ - x^-$. The result is that we can describe any linear program in canonical form very succinctly using matrix notation:

$$\max \{ cx \mid x \geq 0, Ax \leq b \},$$

where c is a row vector with n elements, x is a column vector with n elements, b is a column vector with m elements (where m is the number of constraints), and A is an m by n matrix.

Having a simple standardized representation for linear programs is useful when we want to state theorems about linear programs without having to wade through a lot of special cases. However, for the purposes of reducing an algorithmic problem to a linear program, it's not

necessary to go all the way to canonical form. Anything that is clearly a linear program is fine.

2. Infeasibility and unboundedness

Not all linear programs have solutions. For example

```
maximize
  x
subject to
  x >= 3
  x <= 1
```

has no solution, because there is no assignment to the variables that satisfies all the constraints: it is **infeasible**.

On the other hand

```
maximize
  x
subject to
  x >= 3
```

has no optimum solution because for any choice of x , there is a larger x that gives a better objective function value. This linear program is **unbounded**.

Infeasibility and unboundedness are, fortunately, the only things that can go wrong in a linear program. Any linear program that is neither infeasible nor unbounded has at least one optimum. A good linear programming algorithm should find this optimum, or, if no optimum exists, report whether the problem is infeasibility or unboundedness.

3. Fractional vs integer solutions

Suppose I have a graph G , and I want to mark as many nodes in G as I can without ever marking two adjacent nodes (this is the IndependentSet problem). I could try to solve this problem using a linear program:

```
maximize
  sum xi
subject to
  0 <= xi <= 1 for all i
  xi + xj <= 1 whenever ij is an edge in G
```

The interpretation of this program is that x_i is 1 if and only if vertex i is marked. The constraints enforce that each vertex is enforced at most once, and that each edge has at most one mark on it. What happens if we try to solve this LP? For many graphs, there is a very good chance that our LP solver will come back with every x_i set to $1/2$, giving an objective function of $|V|/2$. This **fractional solution** will in many cases be better than the best **integer solution** using zeroes and ones; for example, if G is a triangle, we can only mark one vertex in the IndependentSet problem, but the LP solver happily marks one and a half.

What went wrong? Nothing in the LP forces the variables to be integers. Indeed, nothing can if we want to be able to solve the LP efficiently---it is generally believed that there is no efficient algorithm for IndependentSet, so there can't be an efficient algorithm for such **integer linear programming** since this would give us an efficient algorithm for

IndependentSet (see PvsNp). So in general you can't count on an LP giving you an integer solution.

Sometimes we can get lucky; for some LPs, if the right-hand sides of each constraint (the elements of b in the matrix formulation) are integral, then all solutions are integral. The way to show that your LP is in this desirable class is to show that the constraint matrix A is **totally unimodular**, which means that the determinant of every square submatrix of A is -1 , 0 , or $+1$. This is still a painful task in general, but some such matrices arise naturally. For example:

- If every row (alternatively, every column) of A contains either (a) no nonzero entries; (b) one nonzero entry, which is $+1$ or -1 , or (c) two nonzero entries, one of which is -1 and the other $+1$, then A is totally unimodular.
- (Special case of above) If A is the incidence matrix of a graph, where $A_{ij} = -1$ when vertex i is the source of edge j , $+1$ when vertex i is the sink of edge j , and 0 otherwise, then A is totally unimodular.
- If A is totally unimodular, then any array obtained by duplicating or deleting rows or columns from A is also totally unimodular.

We'll see some examples of such constraint matrices when we look at applications.

4. Linear programming algorithms

There are two principal algorithms for linear programming.

4.1. The simplex method

This algorithm runs in $O(n^2m)$ time in the typical case, but may take exponential time in the worst case. It works by observing that the set of feasible solutions forms a **polytope** in \mathbb{R}^n , which is the intersection of m half-spaces and which looks like a cut diamond with many flat faces, each of which corresponds to some constraint. The optimal solution lies at a corner of this polytope that is farthest in the c direction.

To find this corner, the simplex method starts at some corner (any one will do), and walks uphill along edges until it can't go any further. The current location at each step is represented by a set of n constraints for which $ax=b$ (called the **basis**), which uniquely identify some corner (since we can solve for x by solving a system of n equations in n unknowns). Moving from one corner to the next involves swapping a new constraint for one of the constraints in the basis, a process called **pivoting**. There are many variants of the simplex algorithm based on different choices of **pivoting rules**, which determine which of the constraints to swap into and out of the basis, but the essential idea is that as long as we can avoid going in circles, we eventually reach the top of the polytope.

In the worst case we may take a long spiraling path that hits every vertex of the polytope; there are examples of linear programs with n variables and $m=n$ constraints in which simplex runs for 2^n steps. However, for good pivoting rules and typical inputs simplex tends to take $O(n)$ steps, each of which costs $O(mn)$ time. So in practice simplex behaves like an $O(n^2m)$ algorithm most of the time. Indeed, the few bad inputs are very brittle; a recent result of Spielman and Teng shows that slight random perturbations of the constraint matrix turn them into good inputs on which simplex finishes in polynomial time.

Simplex algorithms and their variants are the algorithms typically used in practice.

4.2. The ellipsoid method

The ellipsoid method is the most famous of a family of "interior-point" methods for solving linear programs. Unlike the simplex method, which crawls from corner to corner on the outside of the feasible set, interior-point methods move through the interior of the body by a process similar to binary search: one starts with a large geometrical object (e.g. an ellipsoid, which is a sphere that has been stretched in some directions) and replaces it at each step with a new geometrical object of the same type that (a) contains every point of the old object that is better than the previous object's center, and (b) has smaller volume than the old object. Eventually the object shrinks to contain only the optimal corner, which is computed and returned.

The ellipsoid method in its standard implementation takes roughly $O(n^8)$ time, which makes it much worse than simplex for almost all problems. But the $O(n^8)$ is an absolute worst-case guarantee, so the existence of the ellipsoid method means that reducing any other problem to linear programming gives a polynomial-time solution, as well as a reasonably efficient solution (depending on how much the reduction expands the problem) based on simplex.

5. Applications

5.1. Single-source shortest paths as a linear program

```

maximize
  sum du
subject to
  dv <= du + luv   (for all edges uv)
  ds = 0

```

The variables du represent the distances from s to each vertex u . Maximizing the sum of the du is done by maximizing each one individually, since increasing any single du never forces us to decrease some other dv . The set of constraints $dv \leq du + luv$ for fixed v and all u is equivalent to writing $dv \leq \min_u (du + luv)$. The maximum dv that satisfy these constraints are those for which $dv = \min_u (du + luv)$, which is just the definition of distance in a directed graph.

The LP has V variables and E constraints, and can be solved in $O(V^2E)$ time for typical cases and roughly $O(V^8)$ time using the ellipsoid method in the worst case. Both running times are worse than for specialized algorithms (e.g. Bellman-Ford at $O(VE)$).

If we rewrite the LP in canonical form, the constraints end up looking like

$$dv - du \leq luv$$

which produces a constraint matrix with one $+1$ and one -1 in each row (with the exception of the $ds = 0$ row, which turns into two rows with a -1 in one and a $+1$ in the other). So the constraint matrix is totally unimodular, and we get integral distances provided we have integral edge lengths, as we'd expect.

5.2. Max flow

This LP just implements the constraints directly:

```

maximize
  sum fsv
subject to
  sum(all v) fuv = 0   (for all u except s and t)

```

$f_{uv} \leq c_{uv}$	(for all uv)
$f_{uv} = -f_{vu}$	(for all uv)

Since all the constraints for max flow are linear, we get a linear program; its solution solves the max flow problem in $O(E^3)$ time if we use simplex and get lucky. It is not immediately obvious from this program that the solution will be integral even if all c_{uv} are integral, since the constraint matrix is not obviously totally unimodular (the sum(all v) rows have many more than two nonzero entries, and the f_{uv} variables appear four times in each column); but we know from the max flow problem that this is the case.

5.3. Min cut

```


minimize
    sum cij * yij
subject to
    0 <= xi <= 1
    xs = 0
    xt = 1
    yij >= xj - xi
    yij >= 0

```

Here $x_i = 0$ if i is on the s side of the cut, and 1 if it's on the t side. If i and j are both on the same side, or if i is on the t side and j on the s side, we can set y_{ij} to 0, which minimizes the objective function if all c_{ij} are non-negative; this corresponds to an edge that is not in the cut. But if j is on the t side and i is on the s side, then $y_{ij} = 1$ and we have to pay for it in the objective function.

The running time of this algorithm will be $O(E^3)$ in the typical case of simplex method and $O(E^8)$ in the worst case of the ellipsoid method.

6. Older notes

For more about linear programs, see  lp.pdf.

CategoryAlgorithmNotes

2014-06-17 11:58