Notes on Linear Programming

James Aspnes

April 4, 2004

1 Linear Programming

Linear programs are a class of combinatorial optimization problems involving minimizing or maximizing a linear function of a of some real-valued variables subject to constraints that are inequalities on additional linear functions of those variables. We will not discuss the details of the standard algorithms for solving linear programs much, for reasons that will be explained in more detail later. However, linear programming is a very powerful tool for representing a wide variety of optimization problems, and it is important to be able to recognize when a problem can be formulated in terms of a linear program so that it can be solved using these standard tools.

If you want to read more about linear programming, some good references are Chvatal's *Linear Programming* and Papadimitriou and Steiglitz's *Combinatorial Optimization: Algorithms and Complexity*. More advanced references are Schrijver's *Theory of Linear and Integer Programming* and Grötschel, Lovasz, and Schrijver's *Geometrical Algorithms and Combinatorial Optimization*.

1.1 Example

In order to understand the traditional examples of linear programs, it helps to think like a mid-level apparatchik in a Soviet economic planning bureaucracy of the 1950's or, equivalently, like a mid-level manager in a large U.S. manufacturing company of the same period. Imagine that you have a warehouse full of parts that can be assembled into either tanks or tractors. Assembling and operating each tank requires 8 wheels, 1 body, 1 turret, and 1 workers. Each tractor requires 4 wheels, 1 body, no turret, and 3 workers. Under a "guns equal butter" policy, tanks and tractors are equally valuable to the revolution (or, in the case of the manufacturing company, to the shareholders). In addition, every worker must be employed, even if

$$
\begin{array}{lrrcr}
\text{maximize} & & & & \\
\quad x_1 + x_2 & & & & \\
\text{subject to} & & & & \\
8x_1 & + & 4x_2 & \leq & 272 \\
x_1 & + & x_2 & \leq & 59 \\
x_1 & & & \leq & 48 \\
x_1 & + & 3x_2 & = & 112 \\
x_1 & & & \geq & 0 \\
 & & x_2 & \geq & 0
\end{array}
$$

Figure 1: Guns vs butter optimization problem. The number of tanks produced is given by x_1 and tractors is given by x_2. The constraints, in order, correspond to the limited supply of wheels, bodies, turrets, and workers, and the restrictions that we can't produce a negative number of tanks or tractors.

it means making a suboptimal number of tanks and tractors. In your warehouse you have 272 wheels, 59 bodies, 48 turrets, and 112 workers. Your goal is to maximize the total production subject to the limits of what parts are available in the warehouse.

Written as a linear program, the tanks-and-tractor problem might look something like the linear program in Figure 1. Feeding this program to your favorite linear program solver gives the optimal solution of 18.4 tanks and 31.2 tractors.

1.2 The general form of a linear program

The example in Figure 1 illustrates many of the features of a typical linear program. There is a single objective function $x_1 + x_2$ that we would like to maximize. This objective function is linear; it is of the form $\sum_j c_j x_j$ where the x_j are the *variables* of the linear program and the c_j are the objective function coefficients (in this case, 1 and 1). The rest of the program consists of *constraints*. Each constraint in a linear programming specifies a linear function of the variables and requires that that linear function be $\leq$, $\geq$, or $=$ some constant.[1]

[1] If you're wondering where $<$ and $>$ constraints might fit in, they are generally taken to be equivalent to $\leq$ or $\geq$ constraints. The reason is that linear programs are usually solved using floating-point numbers on real machines with round-off error. When x_1 is really $x_1 \pm \epsilon$, the distinction between $x_1 \pm \epsilon > 2$ and $x_1 \pm \epsilon \geq 2$ is largely theoretical.

1.3 Linear programs in canonical form

People who work in this area sometimes make a distinction between linear programs in *general form*, like the one in Figure 1, and those in *canonical form*. In canonical form, the objective function is always to be maximized, every constraint is a $\leq$ constraint, and all variables are implicitly constrained to be non-negative. Any linear program can be reduced to canonical form by simple transformations; for example, the constraint $x_1 + x_2 = 112$ can be replaced by two constraints $x_1 + x_2 \leq 112$ and $-x_1 - x_2 \leq -112$, and a variable y that is allowed to be negative or not can be replaced by an expression $y^+ - y^-$, where y^+ and y^- are both non-negative.

The advantage of making all the constraints less-than-or-equal constraints is that we can write the entire constraint set succinctly using matrix notation. If we similarly use vector notation to represent the variables x and the object coefficients c, even huge linear programs turn into nice little expressions like

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

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, and A is an $m \times n$ matrix. Note that vectors are compared componentwise; $x \leq y$ if and only if $x_i \leq y_i$ for all indices i.

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.

1.4 Infeasibility and unboundedness

Not all linear programs have solutions. The linear program in Figure 2 has an inconsistent set of constraints: there is no assignment to x_1 and x_2 that makes all the constraints simultaneously true. Such linear programs are called *infeasible*.

An equally troublesome difficulty is illustrated by the linear program in Figure 3. Here there exist solutions that satisfy the constraints, but there still does not exist an optimum solution. The reason is that the objective function $x_1 + x_2$ can be made arbitrarily large. Such linear programs are called *unbounded*.

Infeasibility and unboundedness are, fortunately, the only things that can go wrong in a linear program. Any linear program that is neither infeasible

$$
\begin{array}{l}
\text{maximize} \\
\quad x_1 + x_2 \\
\text{subject to} \\
\begin{array}{rcrcl}
x_1 & + & x_2 & \leq & 9 \\
x_1 & & & \geq & 7 \\
 & & x_2 & \geq & 5
\end{array}
\end{array}
$$

Figure 2: An infeasible linear program.

$$
\begin{array}{l}
\text{maximize} \\
\quad x_1 + x_2 \\
\text{subject to} \\
\begin{array}{rcl}
x_1 & \geq & 7 \\
x_2 & \geq & 5
\end{array}
\end{array}
$$

Figure 3: An unbounded linear program.

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.

1.5 Duality

Let's suppose we have a linear program in canonical form:

$$\max\{cx | x \geq 0, Ax \leq b\} \tag{1}$$

and suppose further that we have a candidate solution x^* that we think is optimal. How do we prove this?

We'd like to combine the inequalities in $Ax \leq b$ to get a new inequality of the form $cx \leq z$. We can combine inequalities in the usual way by adding them together, possibly after scaling each one by a non-negative ratio (negative ratios would reverse the sign of the inequality, which we don't want). In matrix notation, this corresponds to multiplying each side of $Ax \leq b$ from the left by a row vector y; yA is a linear combination of the rows of A and yb is the corresponding linear combination of the rows of b. If $y \geq 0$, we have $yAx \leq yb$. So now we just need to arrange for yA to equal c, while making yb as small as possible.

This is a linear program, the *dual* of our original program (1):

$$\min\{yb|y \geq 0, yA = c\}. \tag{2}$$

Linear programming duality says that the solution to the dual and the solution to the original, or *primal* program match exactly:

$$\max\{cx|x \geq 0, Ax \leq b\} = \min\{yb|y \geq 0, yA = c\}. \tag{3}$$

Linear programming duality shows how to prove that x is an optimal solution; we just exhibit the corresponding y with $cx = by$, and we know we can't do better. This relationship is the basis of a family of linear programming-based algorithms called *primal-dual algorithms* that solve problems by alternating between improving separate primal and dual solutions.

An important feature of the relationship between the primal and dual is called *complementary slackness*. Complementary slackness says that any constraint corresponding to a nonzero element of y is tight; or, conversely, that slack constraints are assigned zero weight by y. This is not entirely surprising if we think if the dual as a way to find the best bound on cx by combining constraints; those constraints that are not tight will not give the best bound, so we want to leave them out by multiplying them by zero.

1.6 Fractional vs integral solutions

Sometimes we write a linear program that we hope will give us an integer solution, and it gives us fractional glop instead. Unfortunately, insisting on integral solutions may make finding an optimum much harder.

For example, suppose we want to solve the *maximum independent set* problem. An independent set is a set of vertices in a graph such that no two vertices in the set are adjacent. A maximum independent set (MIS for short) is just an independent set containing the largest possible number of vertices. Figure 4 gives a linear program for finding a maximum independent set in a graph $G = (V, E)$.

Maximum independent set is a well-known NP-complete problem, so we don't really expect this to work. To see what goes wrong, let G be a triangle. Since all the vertices are adjacent to one another, any MIS consists of a single vertex. But the linear program of Figure 4 has a slightly better solution: let $x_1 = x_2 = x_3 = \frac{1}{2}$. This dubious MIS has a suspicious total size of $\frac{3}{2}$, and each vertex is half in and half out of it. Erwin Schroedinger might buy it, but graph theorists will not be impressed.

$$
\begin{array}{l}
\text{maximize} \\
\quad \sum_{v \in V} x_v \\
\text{subject to} \\
\quad \begin{array}{rcll}
x_u + x_v & \leq & 1 & \forall uv \in E \\
x_v & \geq & 0 & \forall v \in V \\
x_v & \leq & 1 & \forall v \in V
\end{array}
\end{array}
$$

Figure 4: Maximum independent set as a linear program. Each variable x_v indicates whether the corresponding vertex v is or is not in the independent set. Independence is enforced by the $x_u + x_v \leq 1$ constraints.

Insisting on integer solutions, even if they are worse than some fractional solution, gives *integer linear programming* problem. This problem is in general NP-hard, since it solves NP-hard problems like maximum independent set.

1.6.1 Total unimodularity

It is sometimes possible to show that a particular linear program or class of linear programs will always have integral solutions. A general way to do this is to show that the matrix A of constraint coefficients is *totally unimodular*, which means that the determinant of every square submatrix is -1, 0, or $+1$. If A is totally unimodular and b is integral, then for any c there is some integral solution x that maximizes cx while satisfying $Ax \leq b$.

Totally unimodular matrices are not always easy to recognize by inspection, though there exists a (complicated) general polynomial-time algorithm that determines whether a given matrix is totally unimodular. It is usually easier to detect when a matrix is *not* totally unimodular. Since the square submatrices of A include individual elements, this means that all individual coefficients must themselves be -1, 0, or $+1$. Similarly, if one can find a submatrix with a determinant other than -1, 0, or $+1$ then A is not totally unimodular. An example of a matrix that is not totally unimodular is the constraint coefficient matrix for the linear program in Figure 4 when G is a triangle.

A criterion that is handy for recognizing some easy cases that arise from graph problems is the following:

Lemma 1 *If every row (alternatively, every column) of a matrix A contains either no nonzero entries or exactly one -1 and exactly one $+1$, then A is*

totally unimodular.

The usefulness of Lemma 1 is that such matrices naturally arise from directed graphs, where the coefficient A_{ij} is -1 when vertex i is the source of edge j, $+1$ when it's the sink, and 0 otherwise.

When applying Lemma 1, remember to put the linear program in canonical form first. For some problems one may have to remove duplicate rows or columns first; this does not affect total unimodularity since any submatrix containing parts of two duplicate rows or columns has determinant 0.

1.7 Optimization problems that are not linear programs

A common trap is to write a would-be linear program that is not quite linear. Suppose that the leadership of our communist dictatorship and/or large manufacturing corporation decides to measure the productivity of our factory by multiplying the number of tanks by the number of tractors. The objective function is then $x_1 x_2$, which is a perfectly good objective function, only it's not linear. Because the objective function isn't linear, we can't use linear programming to solve this problem. Similarly, a constraint like $x_1^2 + x_2 \leq 312$ would violate the requirement that the constraints be linear. Allowing quadratic constraints is a sneaky way to permit integer linear programming, since we can force a variable x to be 0 or 1 by setting $x^2 - x = 0$ and can build bigger integers out of 0-1 variables by representing them in binary. So we can expect that deviating from linearity eliminates any guarantee that the problem can be solved efficiently.

2 Algorithms for linear programming

There are many algorithms for solving linear programs, some of which are practical, and some of which run in polynomial time in the worst case. Sadly, these two sets are (at the moment) disjoint. This split requires considering the possibility that asymptotic worst case analysis might not always be the best tool for identifying algorithms that are useful in practice, a possibility so horrifying that we will concentrate on reducing other problems to linear programming and skip over most of the details of how these linear programs are actually solved.

This section gives enough detail about the two methods to win the "Linear Programming" category on Jeopardy, but not really enough to understand what is actually going on at the level you might need, say, to implement the algorithms. If you don't want to read about linear programming

Method	Typical cost	Worst case cost
Simplex	$O(n^2m)$	Very bad
Ellipsoid	$O(n^8)$	$O(n^8)$

Table 1: Everything you need to know about solving linear programs.

trivia, skip to the convenient summary in Table 1. If you want to really understand these algorithms, see the references at the beginning of Section 1.

2.1 The Simplex Method

The simplex method, invented in 1951 by Dantzig, has the desirable property of running in time comparable to Gaussian elimination for typical inputs. It has the undesirable property of running in exponential time for carefully constructed pathological inputs.

The key idea of the simplex method is that the set of points that satisfy the constraints of a feasible linear program form a body in n-dimensional space called a *polytope*, which is a convex body with flat faces. These flat faces are shaved off by the constraints $ax \leq b$, each of which cuts the space in half along the hyperplane $\{x|ax = b\}$. A solution to the linear program is the corner of the polytope that is farthest in the c direction, where cx is the objective function. 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 talk 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.

2.2 The Ellipsoid Method

The ellipsoid method, invented in 1979 by Khachian, is an algorithm for solving linear programs that runs in polynomial time in the worst case. Sadly, the polynomial is no better than $O(n^5)$ even with very generous cost assumptions, such as being able to do arithmetic with $\Theta(n^3)$ bits of precision in constant time. Since we usually assume constant-time arithmetic only for numbers of reasonable size (no more than $O(\log n)$ bits), a more accurate asymptotic running time would be $O(n^8)$. Because the running time is usually much larger than the typical running time of simplex (and other similar algorithms), the ellipsoid method is not used much in practice. But it does mean that any problem that can be expressed as a linear program can be solved in polynomial time.

The key idea of the ellipsoid method is similar to binary search. An ellipsoid, which is a sphere that has been run through a linear transformation, is constructed that contains the solution of the linear program somewhere in its interior. At each step, one of the constraints of the linear program is used to slice off a piece of the ellipsoid that does not contain the solution, and a new ellipsoid is found that contains all the points in the remaining part of the old ellipsoid (plus a few more points, since it's bulgy). Since the volume of the ellipsoid shrinks by a guaranteed amount at each step, after $O(n^2)$ steps the ellipsoid converges to the corner corresponding to the optimal solution.

3 Applications of linear programming

Now that we know how to solve linear programs, let's see what we can do with them.

3.1 Max flow

Recall that in the maximum flow problem we are given a graph $G = (V, E)$, a designated source vertex s and sink vertex t, and a capacity c_{uv} for each edge $uv \in E$. We want to maximize the flow from s to t where the net flow into each other vertex is 0 and the flow across any edge uv is between 0 and c_{uv}.

This problem has a natural representation as a linear program, as shown in Figure 5. To simplify the presentation we treat missing edges as edges with capacity 0.

maximize

$$\sum_{v \in V} f_{sv}$$

subject to

$$\begin{aligned} \sum_{u \in V} f_{uv} - \sum_{u \in V} f_{vu} &= 0 && \forall v \in V - \{s, t\} \\ f_{uv} &\leq c_{uv} && \forall u, v \in V \\ f_{uv} &\geq 0 && \forall u, v \in V \end{aligned}$$

Figure 5: Max flow as a linear program.

maximize

$$\sum_{v \in V} f_{sv} - \sum_{uv \in E} m_{uv} f_{uv}$$

subject to

$$\begin{aligned} \sum_{u \in V} f_{uv} - \sum_{u \in V} f_{vu} &= 0 && \forall v \in V - \{s, t\} \\ f_{uv} &\leq c_{uv} && \forall u, v \in V \\ f_{uv} &\geq 0 && \forall u, v \in V \end{aligned}$$

Figure 6: Max flow as a linear program, with edge charges.

maximize

$$\sum_{v \in V} f_{sv} - \sum_{uv \in E} m_{uv} f_{uv}$$

subject to

$$\begin{aligned} \sum_{u \in V} f_{uv} - \sum_{u \in V} f_{vu} &= 0 && \forall v \in V - \{s, t\} \\ f_{uv} &\leq c_{uv} && \forall u, v \in V \\ f_{uv} &\geq 0 && \forall u, v \in V \\ f_e &\geq 1 \end{aligned}$$

Figure 7: Max flow as a linear program, with edge charges and starving cousin constraint.

From this linear program we can conclude that maximum flow can be solved in polynomial time (which we probably already knew). However, having max flow as an LP allows us to do a few more tricks. Suppose we are charged a fixed rate m_{uv} for sending each unit of flow over uv, and we want to maximize the net flow minus all the edge charges. This just involves changing the objective function, as shown in Figure 6.

Or, perhaps, our cousin runs the trucking company that carries flow over a particular edge e and we want to guarantee that she gets at least one unit of business to keep her children from starving. The linear program for this problem is shown in Figure 7.

We could probably achieve similar results by tinkering with the standard max-flow algorithms, but by using a linear programming approach we can get these results with much less mental effort.

3.1.1 Integer solutions

It follows from Ford and Fulkerson's algorithm that if all the capacities of a graph are integral that there is a maximum flow that is also integral. Another way to get this result is to observe that the constraint coefficient matrix in Figure 5 is totally unimodular, by removing duplicate rows and applying Lemma 1 to the columns. A similar observation applies to the problems in Figures 6 and 7.

3.2 Shortest paths

Figure 8 gives a linear program for computing distances from a single source. All-pairs shortest paths can be solved using the linear program in Figure 9. Both constraint matrices are totally unimodular by Lemma 1 after removing duplicate rows; this is not surprising since we expect all shortest paths in an integral world to have integral length. Neither approach is as efficient as using a specialized algorithm. What happens with disconnected vertices or negative cycles?

3.3 Sorting

No algorithm can claim to be universal if it can't sort. But linear programming can! In the linear program in Figure 10, $x_{ij} = 1$ if item i is placed in location j. The key of each item is assumed to be a fixed value k_i (that is used in constructing the linear program). If all items have different keys, the unique optimum is the one in which larger items appear in larger locations, with all x_{ij} equal to 0 or 1. If two or more items have the same key, it is

$$
\begin{array}{l}
\text{maximize} \\
\quad \sum_{v \in V} d_v \\
\text{subject to} \\
\qquad\qquad d_s = 0 \\
\quad d_u + \ell_{uv} - d_v \leq 0 \quad \forall uv \in E
\end{array}
$$

Figure 8: Single-source shortest paths as a linear program.

$$
\begin{array}{l}
\text{maximize} \\
\quad \sum_{v \in V} d_{uv} \\
\text{subject to} \\
\qquad\qquad d_{vv} = 0 \quad \forall v \\
\quad d_{uv} + \ell_{vw} - d_{uw} \leq 0 \quad \forall u \in V, \forall vw \in E
\end{array}
$$

Figure 9: All-pairs shortest paths as a linear program.

possible to get a fractional solution in which the items are spread between two adjacent locations.

It is left as an exercise to the reader to adapt this program to do something more useful, like computing a maximum-value matching in a bipartite graph.

$$
\begin{array}{l}
\text{maximize} \\
\quad \sum_{i,j} j \cdot k_i \cdot x_{ij} \\
\text{subject to} \\
\quad \begin{array}{rcll}
\sum_j x_{ij} & = & 1 & \forall i \\
\sum_i x_{ij} & = & 1 & \forall j \\
x_{ij} & \geq & 0 & \forall i, j
\end{array}
\end{array}
$$

Figure 10: LP-sort, the world's second worst sorting algorithm. Sorting is mostly done by the objective function. Object i is in location j if $x_{ij} = 1$. The constraints make sure that no location gets more than one object, no object appears in more than one location, and no location contains a negative fraction of some object.