Linear Programming and Network Optimization

Jonathan Turner

March 31, 2013

Many of the problem we have been studying can be viewed as special cases of the more general *linear programming problem* (LP). In the linear programming problem, we seek to optimize some linear function of a set of non-negative real variables $x_1, \ldots, x_n$, subject to a set of linear constraints on those variables. A typical instance of linear programming takes the form.

$$\text{maximize } \sum_j c_j x_j \qquad \text{subject to } \sum_j a_{i,j} x_j \leq b_i \quad \text{for all } i$$

Here, the c_j, $a_{i,j}$ and b_i are numerical values defined by the specific problem instance. The objective of the problem is to assign values to the x_js that maximize the summation at left, which is known as the *cost function*. Linear programs can be stated more concisely in matrix format.

$$\text{maximize } C^T X \qquad \text{subject to } AX \leq B$$

Here, C denotes a column vector of cost coefficients, X a column vector of variables, $A = [a_{i,j}]$ is a coefficient matrix, B is a column vector representing the bounds in the inequalities and the multiplication operation is matrix multiplication.

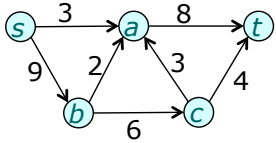
Linear programs arise in a wide range of applications and can be solved efficiently. The classical *simplex algorithm* has been very successful in practice, although its worst-case running time is exponential in the size of the problem instance. So-called *interior point methods* can be used to solve linear programs in polynomial time, although they are often slower than the simplex algorithm, in practice. It's importat to note that the variables x_j are defined as real-valued. If some or all of the variables are required to be integers we get an *Integer Linear Program* and in general, these problems are NP-hard.

While most of the network optimization algorithms we have studied can be formulated as linear programs, it is usually more efficient to solve them using more specialized algorithms. Still, it can be useful to view them from the perspective of linear programming, as this perspective can sometimes be used to discover algorithms that we might otherwise fail to notice. We will see an example of this when we consider the extension of Edmond's algorithm to the problem of maximum weight matching in general graphs.

Let's start by considering the maximum flow problem, as a linear program. In this case, the variables are the flows on the edges. We seek to maximize the total flow leaving the source, subject to the capacity constraints and the flow conservation condition. This can be written as follows:

$$\text{maximize} \sum_e f_e \quad \text{subject to} \quad 0 \le f_e \le cap_e \quad \text{and} \quad \sum_{e=(w,u)} f_e = \sum_{e=(u,v)} f_e$$

where the inequalities are defined for all edges e and the equations are defined all vertices u, except the source and sink. Figure 1 shows an example of this formulation of the max flow problem.



maximize $f_{sa} + f_{sb}$

subject to
$f_{sa} \le 3$ $f_{sb} \le 9$ $f_{ba} \le 2$ $f_{bc} \le 6$ $f_{at} \le 8$ $f_{ca} \le 3$ $f_{ct} \le 4$
$f_{sb} = f_{ba} + f_{bc}$ $f_{bc} = f_{ca} + f_{ct}$ $f_{sa} + f_{ba} + f_{ca} = f_{at}$

Figure 1: Example of a flow problem as a linear program

To put this in matrix form, let F be a column vector with an entry f_e for every edge e and let S be a column vector with a 1 for every edge leaving the source and a 0 for every other edge. With this definition, $S^T F$ denotes the total flow leaving the source. For the example in Figure 1,

$$\begin{aligned} F^T &= \begin{bmatrix} f_{sa} & f_{sb} & f_{at} & f_{ba} & f_{bc} & f_{ca} & f_{ct} \end{bmatrix} \\ S^T &= \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

The A matrix used to represent the coefficients in the linear constraints is constructed from two building blocks. The first is just an $m \times m$ identity matrix, where m is the number of edges in the flow graph. The second is an $(n-2) \times m$ *edge incidence matrix* $G = [g_{u,e}]$ where for all u except the source and sink, $g_{u,e} = 1$ if u is the tail of e, -1 if u is the head of e and 0

otherwise. For the example flow graph,

$$G = \begin{bmatrix} -1 & 0 & 1 & -1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 1 \end{bmatrix}$$

We construct the complete A matrix "stacking" the identity matrix above a copy of G and a copy of $-G$. To complete the matrix formulation, we define the column vector B representing the bounds in the inequalites, to be a column vector, with $m + 2(n-2)$ rows. The first m entries are the edge capacities, and the remaining entries are zero. For the example flow graph, the inequalities can be expressed as

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 1 & -1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 1 \\ 1 & 0 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 \end{bmatrix} \times \begin{bmatrix} f_{sa} \\ f_{sb} \\ f_{at} \\ f_{ba} \\ f_{bc} \\ f_{ca} \\ f_{ct} \end{bmatrix} \leq \begin{bmatrix} 3 \\ 9 \\ 8 \\ 2 \\ 6 \\ 3 \\ 4 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Note that the reason that we include both G and $-G$ in the A matrix is so that we can represent the flow conservation equalities with the equivalent pair of inequalities. This allows us to maintain the standard matrix form, which will become important when we discuss duality and the concept of complementary slackness.

With these definitions, the max flow problem becomes

$$\text{maximize } S^T F \qquad \text{subject to} \quad AF \leq B$$

The min cost max flow problem can also be expressed as a linear program. Here the natural formulation is

$$\begin{aligned} &\text{minimize} && \sum_e c_e f_e \\ &\text{subject to} && \sum_{e=(s,u)} f_e = f^* \end{aligned}$$

$$f_e \leq cap_e \quad \text{for all } e$$
$$\sum_{e=(w,u)} f_e = \sum_{e=(u,v)} f_e \quad \text{for all } u \text{ but the source and sink}$$

Here f^* denotes the maximum flow. We can put this into the standard matrix form by first converting it to a maximization problem by negating all the cost coefficients, then extending the coefficient matrix and bounds vector from the max flow problem to express the additional constraint on the total flow.

We've noted earlier that for the max flow problem, whenever the capacities are all integers, there is a maximum flow in which all flows are integers. This actually follows from a general property of the coefficient matrix. We say that a matrix is *totally unimodular* if every square submatrix has a determinant that is equal to 0, 1 or -1. In a linear program with an integer coefficient matrix and bounds, if the coefficient matrix is totally unimodular, there is an optimal solution in which all the variables have integer values. While it's tedious to verify the total unimodularity property, it's easy to see for our example flow problem that every non-singular 2×2 submatrix of the coefficient matrix has a determinant that is either $+1$ or -1.

So why does this matter? Suppose we are trying to solve a linear program, in which some variables are required to be integers. For example, we might require that certain variables take on only values 0 or 1. Suppose we replace these integrality constraints with bounds on the values of the variables (so the 0-1 variables are now just real-valued variables in the interval [0,1]). This ordinary linear program is called the *LP relaxation* of the ILP. If the coefficient matrix for the LP relaxation is totally unimodular, then it has optimal solutions in which all variables are integers (so those variables that we wanted to be 0 or 1, will automatically be integral in the solution to the LP relaxation). This property can be used to solve some integer linear programs, using their LP relaxation. We'll see an example of this when we study the weighted matching problem in general graphs.

The usual max flow and min cost flow problems model situations in which we have one type of "commodity" that we want to move from a source to a sink. In some applications, we're interested in moving multiple types of commodities among multiple sources and sinks. These applications can be modeled as a *multicommodity flow problem*. One formulation of the multicommodity flow problem includes a source/sink pair for each *commodity* we want to move. Edges have total capacities as before, but also have separate capacities for individual commodities. We seek a flow function for each commodity that respects the per-commodity constraints and for which the total

flow on each edge (from all commodities) respects the total capacity. The multicommodity flow problem can also be formulated as a linear program, but unlike the normal max flow and min cost flow problems, there are no known algorithms for this problem that are substantially more efficient than solving the linear program directly. The LP formulation also allows for the introduction of per-commodity costs and more complex capacity constraints (for example, limiting the flow involving a given pair of commodities).

Next, let's look at how we can formulate the shortest path problem in graphs as a linear program. To keep things simple, we'll focus on the single-pair version of the problem, where we want a shortest path from a vertex s to a vertex t. We'll also assume strictly positive costs. One way to solve this problem is to turn it into a min cost flow problem, but we'll take a more direct approach. Let's define an ILP using a 0-1 selection variable for each edge e. In the optimal solution, the non-zero variables will define a shortest path from s to t. Our objective is to minimize the sum of the edge costs for those edges e with $x_e = 1$. So, if C is a vector of edge lengths and X is a vector of variables, we seek to minimize $C^T X$.

To ensure that the non-zero variables define a path, we introduce a constraint that requires that the sum of the edges incident to t be at least 1. This ensures that there will be at least one selected edge entering t. Of course we only want one such edge, but since the edge costs are positive, we can be sure that an optimal solution will not contain any edges not needed to define a path. We also include a constraint for every other vertex u except the source. These constraints require that the sum of the variables for the edges entering u be at least as large as the sum of the variables for the edges leaving u. This means that if one of u's outgoing edges is selected, then at least one of its incoming edges is also. Notice that any shortest path from s to t satisfies these conditions, and no other set of edges can satisfy the conditions at a lower cost. An example of this formulation is shown in Figure 2. Notice that the coefficient matrix A is the edge incidence matrix

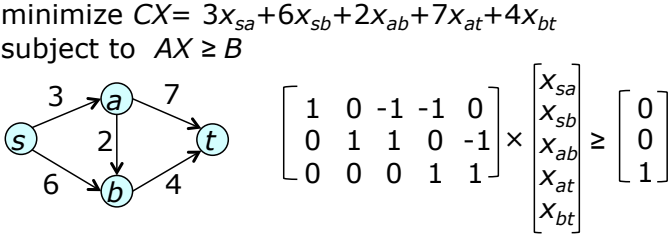


Figure 2: Shortest path problem as a linear program

for the graph, with the row for the source vertex omitted. Also, notice that we have not included the 0-1 constraints on the variables. It turns out that these can be replaced with simple inequalities requiring that each x_e be no larger than 1. Indeed, even these can be omitted, since the minimization in the objective function will prevent the assignment of larger values in an optimal solution.

There is actually an alternate way to formulate the shortest path problem as a linear program. The easiest way to understand this version is to imagine the graph as a collection of balls connected by strings of different lengths. If we pull the balls for s and t as far apart as we can, then the distance between them will be the shortest path distance, since the strings for the edges on the shortest path won't allow us to pull them any further apart than this.

Using this idea, we define a variable d_u for each vertex u, that represents a possible distance between s and u. If we require that every edge $e = (u, v)$ satisfy the constraint $d_v \leq d_u + c_e$ (where c_e is the length of e), then d_u can be no larger than the shortest path distance from s (in the ball and string analogy, the ball for u can be pulled no further from the ball for s than the lengths of the strings allow). So now, we can define an LP maximization problem where we seek to maximize d_t subject to the constraints defined by the above inequalities. Since $d_s = 0$ we don't need to include it as an explicit variable. For the previous example, this formulation of the problem is given in Figure 3.

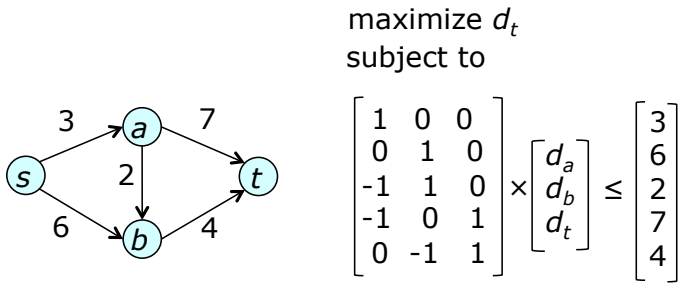


Figure 3: Alternate formulation of shortest path problem

These two versions of the shortest path problem are actually closely related to one another. Specifically, they are *duals*. Dual problems have a number of useful properties (for example the values of the optimal objective functions are equal). Sometimes, the dual problem is easier to solve than the primal problem. In other cases, we can arrive at a solution most efficiently

by solving the primal and dual versions of the problem in tandem. Indeed, the usual labeling method for the shortest path problem can be viewed as such *primal-dual algorithm*.

To define the dual problem in general, suppose we start with a maximization problem in standard matrix form: maximize $C^T X$ subject to $AX \leq B$. The dual problem is then to minimize $B^T Z$ subject to $A^T Z \geq C$, where Z is a vector of *dual variables*. Observe how the role of vectors B and C are interchanged.

Note that if we are given an LP that is not in standard form we can always convert it to standard form. Specifically, a minimization problem can be converted to a maximization problem by negating the cost vector. Also, any lower-bound constraint can be converted to an upper-bound constraint by multiplying both sides by -1. When constructing the dual for a problem, it is helpful to go through this process in a methodical way, even though the standard form may not be the most natural way to express the problem.

One of the useful properties of the dual problem is the *complementary slackness* property. For the problem maximize $C^T X$ subject to $AX \leq B$, the slackness vector for a given value of X is just $B - AX$. In an optimal solution, some of the constraints become equalities (that is, the slackness of those constraints is zero), so the general notion of slackness tells us something about how close a given solution is to an optimal one. Consider the dual of the above problem: minimize $B^T Z$ subject to $A^T Z \geq C$. In the dual problem, the slackness vector is $A^T Z - C$. The complementary slackness property states that X^* and Z^* are both optimal solutions to the primal and dual problems if and only if the following two conditions are satisfied.

$$\begin{aligned} (B - AX^*) = [s_i] \quad &\Rightarrow \quad s_i z_i = 0 \quad \text{for all } i \\ (A^T Z^* - C) = [t_j] \quad &\Rightarrow \quad t_j x_j = 0 \quad \text{for all } j \end{aligned}$$

In other words, for each slackness value s_i in the primal, either s_i or the corresponding dual variable z_i must be zero. Similarly, for each slackness value t_j in the dual, either t_j or the corresponding primal variable x_j must be zero. These conditions can be written as a pair of equations

$$(B - AX^*) \circ Z^* = [0] \quad \text{and} \quad (A^T Z^* - C) \circ X^* = [0]$$

where the symbol $\circ$ denotes the vector formed from pairwise products of the vectors' elements. This is known as the Hadamard product of the vectors.

So-called *primal-dual algorithms* adjust values of the primal and dual variables with the objective of making the complementary slackness conditions true, thus arriving at an optimal solution.