CMPSCI611: **Linear Programming** Lecture 23

For our last three lectures we turn to a general optimization problem called **linear programming**, which involves maximizing or minimizing a linear function subject to linear constraints. Linear programming was identified as important in the 1950's, and a general solution called the **simplex method** was developed. It turns out that the simplex method is not guaranteed to run in polynomial time, but usually does so in the problems that occur in practice. Since the development of **NP**-completeness, polynomial-time general solutions have been developed, some of which are competitive with simplex in practice.

Here we'll present the simplex method (in Lecture #24), but not the more complicated poly-time methods. Our main focus will be on understanding what a linear programming problem is, and thus when a general linear programming algorithm might be useful.

Let's first see an example. We have a set of foods, each of which provides a certain amount (per unit of food) of each of a set of nutrients. We need to choose a diet, which consists of a non-negative real-number amount of each food. A **feasible solution** will be a diet that provides at least the required daily allowance of each nutrient. Each food also has a **cost** per unit, and the **optimal solution** will be a diet that minimizes this cost while still being a valid solution.

Let f be the number of foods and n be the number of nutrients. A diet is thus a vector d of f real numbers. Our input to the problem consists of:

- A vector c of length f, where c_j is the cost of one unit of food i,
- An n by f matrix M, where m_{ij} is the amount of nutrient i provided by food j, and
- A vector r of length n, where r_i is the required daily amount of nutrient j.

We write d and c as column vectors. Thus for a diet d, Ad is a vector of length n, whose i'th component gives the amount of nutrient i provided by d. To be a valid solution, each entry of Ad must be greater than or equal to the corresponding entry of r. We write this last property as "$Ad \geq r$", recognizing that we are overloading the operator "$\geq$". (For one thing, this $\geq$ is not a total order on vectors, because two vectors might be incomparable.)

Our goal is thus to find a diet d that minimizes the dot product $c \cdot d$ (a **scalar**, or single real number) while meeting the **linear constraint** $Ad \geq r$. Also note that we must have the vector inequality $d \geq \mathbf{0}$, where $\mathbf{0}$ is the vector of all zeroes, for a diet to make sense, because we must have a non-negative amount of each food.

It is crucial to the definition of the problem that we are allowed to choose a *real* number of units of each food in our diets, rather than an integer. If we were forced to choose integer values we would have an instance of **integer programming**, and it is not hard to see that integer programming in general is **NP**-hard. In fact, when we proved the SUBSET-SUM decision problem to be **NP**-complete, we really reduced 3-SAT to SUBSET-SUM through an intermediate decision problem:

- We chose an integer vector x, giving a setting of each variable x_i
- We bounded x by the vector inequalities $\mathbf{0} \leq x$ and $x \leq \mathbf{1}$
- Each clause gave us a constraint: for example, we could write $x_1 \vee \neg x_2 \vee x_3$ as $x_1 + (1 - x_2) + x_3 \geq 1$ or $x_1 - x_2 + x_3 \geq 0$. The set of constraints could be written "$Cx \geq b$", where C is a matrix specifying which variable is in which clause and b is a vector that indirectly tells how many negative literals are in each clause.

The SUBSET-SUM problem had a solution iff there is an integer vector x satisfying all these constraints. You'll show on HW#5 that detecting whether there is a feasible solution in a *linear* program is an equivalent to solving a linear programming problem, so there are practical methods to solve it. Why can't we use these methods to solve SUBSET-SUM this way?

The problem is that the linear program might well have a feasible solution where the values of x_i are not integers – we could "satisfy a clause" by setting part of one variable true and part of another true. The existence of such a solution says nothing about whether an integer solution, and thus a satisfying assignment, might exist.

It's useful to have a **standard form** for linear programming problems. We say that a problem is in standard form if it is given in terms of an m by n matrix A, an m-vector b, and an n-vector c so that:

- We want to choose a n-vector x to maximize the real number $c\dot{x}$, but
- The vector x must satisfy two constraints: the m-vector inequality $x \geq \mathbf{0}$ and the n vector equation $Ax = b$.

Is our diet problem in this standard form? No, we have d with the constraint $d \geq \mathbf{0}$, but the remaining constraints are inequalities, not equalities. But we can convert the problem into standard form without too much trouble. Note that in a valid solution, where $Ad \geq r$, we can write Ad as $r + e$ (using vector addition) where $e \geq \mathbf{0}$. Here e is the *excess* vector of nutrients our dieter eats beyond the daily requirement.

We *transform* the problem so that x is a vector of length $f+n$, the concatenation of d and e. Now forcing $x \geq \mathbf{0}$ forces both $d \geq \mathbf{0}$ and $e \geq \mathbf{0}$, as desired. We need to define equality constraints that will force the correct relationship $Ad = r + e$. All we need is to have $Ad - Ie = r$, which we can write as $Bx = r$ where B is a matrix that is a concatenation of A and $-I$.

In general a linear programming problem has some vector of n variables, x each of which may be constrained to be non-negative, constrained to be positive, or not constrained at all. Then we have m linear constraints, each of the form $a_i \cdot x = b_i$, $a_i \cdot x \geq b_i$, or $a_i \cdot x \leq b_i$. We could make these constraints into a matrix equation *if* they were all equalities.

But we can *make* them all equalities, by adding new variables just as we did with the diet problem. For every constraint of the form $a_i \cdot x \geq b_i$, we make a new *scalar* variable y_i to represent the extent by which the constraint is oversatisfied. So we have $a_i \cdot x = b_i + y_i$ and $y_i \geq 0$. The first equation can be written $a_i \cdot x - y_i = b_i$, which is a linear constraint on what are now $n+1$ variables, the n variables in x together with y_i.

We deal with $a_i \cdot x \leq b_i$ just the same way, letting $a_i \cdot x + y = b$ with $y \geq 0$. The only remaining problem is any unconstrained variables x_i. But for these we can just let $x_i = x_i' - x_i''$ with $x_i' \geq 0$ and $x_i'' \geq 0$.

So now we have a (perhaps longer) vector of variables z, with $z \geq \mathbf{0}$, and a set of linear equality constraints that we can write $Az = b$. We can make two more general simplifications:

- Because we can multiply both a row of A and the corresponding entry of b by -1 without changing the meaning of the equations, we may assume the vector inequality $b \geq \mathbf{0}$.
- If there are linear dependences among the rows of A, we can delete any redundant rows (or just give up if the constraints are contradictory, *e.g.*, $x_i = 2$ and $x_i = 3$).
- If removing linearly independent rows gives us $Ax = b$ where A is a *square* matrix we just invert A and report the answer as $x = A^{-1}b$ – if this fails to satisfy $x \geq \mathbf{0}$ then there is no solution. Remember that inverting a matrix is of similar complexity to matrix multiplication.

Let's look at one more example, a **network flow** problem with n vertices and e directed edges. Remember that we have a flow f_{uv} on each edge (u, v), satisfying the **skew symmetry** constraints $f_{uv} + f_{vu} = 0$ for every pair of vertices and the **flow conservation** constraints $\Sigma_v f_{uv} = 0$ for every vertex except s and t. We can write these last constraints as $Bf + Fd = \mathbf{0}$, where:

- B is an n by e matrix with entry $+1$ when an edge flows out of a vertex and -1 when it flows into a vertex,
- f is an e-vector giving the flow on each edge (with an entry for both f_{uv} and f_{vu}),
- F is a scalar, the size of the flow (the net amount out of s or into t)
- d is an n-vector with entry -1 for s, 1 for t, and 0 elsewhere

We want to maximize the scalar F while keeping all the equality constraints and keeping the vector inequality $f \leq c$, where c is an e-vector giving the maximum flow on each edge in its given direction.

To convert this problem to standard linear programming form, we let g be an e-vector giving the **surplus capacity** $c_{uv} - f_{uv}$ for each edge (u, v). Now we can just say that $g \geq \mathbf{0}$, and the remaining constraints are all equality constraints. For every u and v, we have $g_{uv} + g_{vu} = c_{uv} + c_{vu}$ – these replace the skew symmetry constraints. Once we recast B in terms of g as a new matrix B', we get $B'g = z$ for some constant vector z, and we want to *minimize* the excess capacity out of vertex s.

Remember that in the standard problem we want to minimize some linear function of the variables. When we create new variables, we give them zero weight in the function to be minimized. In the diet example the cost is a given linear function of the food items, and in the max flow example it is the excess capacity out of the start vertex.

Why should we expect this minimization problem to have a nice solution in general? This should become clearer when we look at a **geometric** interpretation of linear programming.

Feasible Solutions

A linear programming problem in standard form gives us a space of values for a vector x – the segment of the Euclidean space $\mathbf{R}^n$ where all values are positive. (With $n = 2$ this is the northeast quadrant, when $n = 3$ it is an octant, and so forth.) We then have a set of $m < n$ constraints. What do they look like geometrically?

A single linear constraint of the form $a \cdot x = b$, where a is a constant vector, x a variable vector, and b a scalar constant, represents a **plane** or **hyperplane** in $\mathbf{R}^n$. For example, think of $\mathbf{R}^3$ and the equation $x_1 + x_2 + x_3 = 1$. The solution set of the equation is a plane passing through $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, perpendicular to the vector $a = (1, 1, 1)$.

So our solution set is the part of the positive octant of $\mathbf{R}^n$ that is the **intersection** of the m planes corresponding to the constraints. In general, the intersection of m planes will be a space of dimension $n - m$, if the planes are in **general position**. This means that no two of them are parallel and no three intersect in a space of dimension more than $n - 3$.

The situation is easier to visualize if we transform the problem once again, from standard form into **slack form**. Remember that A is an m by n matrix and b is an m-vector with $b \geq \mathbf{0}$. By Gaussian operations, we can change the equation $Ax = b$ to $A'x = b'$, where the first m columns of A' form an identity matrix. (This is the same as multiplying both A and b on the left by the inverse of the matrix formed by the first m columns – if these columns are not linearly independent we can re-order the variables so that they are.)

Now we have the first m variables each expressed as a function of the last $n - m$ variables. So for i from 1 through m, we have the (scalar) inequality $x_i \geq 0$ and an equation $x_i + \Sigma_j c_j x_j = b_i$, which gives us $\Sigma_j c_j x_j \leq b_i$, a linear inequality in the last $n - m$ variables.

We have m constraints on these $n - m$ variables, each of which has a solution space called a **half-space**, a plane together with all the points on one side of it in $\mathbf{R}^{n-m}$. The intersection of one or more half-spaces gives us a set called a **polyhedron**, or a **polytope** if it happens to be bounded. For example, in $\mathbf{R}^3$ if we have $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$, and $x_1 + x_2 + x_3 \leq 1$, we define a polytope that is a tetrahedron, with four vertices $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$.

Page 143 of the Adler notes (to be displayed in class on a slide) has a picture of a more complicated polytope in $\mathbf{R}^3$, arising from the following set of inequalities:

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 4 \\ x_1 \qquad\qquad &\leq 2 \\ x_3 &\leq 3 \\ 3x_2 + x_3 &\leq 6 \end{aligned}$$

These four inequalities are derived from a set of four equations in seven variables. But of those seven variables, only three are independent – the standard-form constraints define the other four in terms of them.

Now that we can see the polytope (or polyhedron) geometrically, we can interpret the minimization of the objective function. It is a linear function of the variables in the standard-form problem, which we may rewrite as a function of the $n - m$ variables in the slack-form version (since each of the other variables may be written in terms of those).

A function of the form $c \cdot x$, where x is now an $(n - m)$-vector, has constant planes that are each perpendicular to the vector c. If we imagine one of these planes moving from one constant value of $c \cdot x$ to another, we see that it will sometimes intersect the feasible region's polytope and sometimes not. If and when it changes from intersecting to not intersecting, it does so at a **vertex** of the polytope – a point where at least $n - m$ of the planes meet. It is possible that the plane will meet more than one vertex (and the lines or faces between them) at once, but it will necessarily hit a vertex.

If the feasible region is an unbounded polyhedron, the objective function on it may be unbounded in one or both directions. If it takes on all numbers $y \leq c$ for some constant c, for example, then the minimization problem has no solution – there is no minimum value because you can always go lower.

But if the feasible region is bounded, or at least if the value of the objective function is bounded below in the case of a minimization problem, there will be a solution to the problem and it will occur at a vertex or vertices.

This suggests a solution to the general linear programming problem! Compute the location of all the vertices, then calculate the objective function at each of them and find the smallest value.

This is a valid method, but in general not an efficient one because there might be an exponential number of vertices. In the next lecture we'll consider a more systematic method for searching the vertices to find one with a minimum value.