Chapter VII: Linear Programming II

1 How Good is the Simplex Algorithm?

We have seen that the act of pivoting takes on the order of nm steps of addition and multiplication. The amount of effort necessary to solve an LP problem by the simplex method is therefore on the order of nm times the number of pivots needed, measured in single multiplications.

So the crucial issue, in evaluating the efficiency of the simplex algorithm, is: how many pivots can one expect to encounter in solving it?

There are some conflicting bits of evidence on this question, as follows:

1. In practice, the standard method of the field of operations research is to take whatever problem one has, approximate it as an LP, solve that by the simplex algorithm, and apply the result. Practitioners in this field claim that they generally need a number of pivots that is linear in the smaller of n and m.

2. One can find examples of problems in which it is possible to perform an exponential number of pivots and in fact to wander through every single vertex of the feasible region by pivoting. Though there are some versions of the algorithm which can do slightly better than this, there is none that anyone knows about that can't be forced into an exponential number of pivots by an appropriate problem.

3. We can construct a plausibility argument that implies (more or less) that the average performance of the version of the algorithm we have been discussing is roughly linear in the smaller of n and m.

We first consider the second of these points:

A Bad Kind of Problem There is an ancient puzzle called the Tower of Hanoi, which consists of three pegs, and a number n of disks of graduated radii. The disks initially sit on the first peg; they may be transferred one at a time from peg to peg; a smaller one may not sit under a bigger one, and one wants to transfer them all to another peg.

It is easy to see that transferring all n disks requires $2^n - 1$ individual transfers, by induction. For, in order to move the nth disk one must first move all $n-1$ others off it, and then must reassemble them on top of it. Thus $f(n) = 1 + 2f(n-1)$, where $f(n)$ is the indicated number of transfers, and this recursion has the solution $f(n) = 2^n - 1$.

If we can exhibit a situation in which pivoting can behave as the transfer steps in the Tower of Hanoi, we can have, similarly an exponential amount of pivoting.

With n constraints and n positive variables, one can construct a feasible region that looks like a cube in n-dimensions, which region will then have an exponential number of vertices. We will construct one that has the property that there is a path along its edges in which the objective function increases at each step, that goes through every vertex. This property implies that an unlucky sequence of pivots could wander through every single vertex of this region.

We can show this inductively; but instead of using n positive variables and n constraints, we use n unconstrained variables and $2n$ constraints. They are:

$$x_1[1, x_1Z - 1$$

for $j = 2$ to n:

$$x_j[x_{j-1} + 2^{j-1},$$
$$x_jZ - x_{j-1} - 2^{j-1}$$

and the LP problem is to maximize x_n.

Let us verify the following claims about this problem:

1. The variable x_n appears in only two constraints; on the faces of the feasible region in which either of these constraints is an equality, x_n is proportional to x_{n-1} or minus it.

2. If there is (as we assume by induction) a path in either face through all vertices using edges of the face only in which x_{n-1} always changes in the same direction, we can construct such a path in the entire region in which $x - n$ always in the same direction, by putting together the paths that work for x_{n-1} on each x_n-face, if we can jump from the vertex with largest value of x_n in the lower face to the one with smallest value of x_n in the upper face, along an edge of the region.

3. The vertex in the lower face with maximum x_n and the one in upper face with minimum x_n are each the (unique) vertex in its face having smallest value for x_{n-1}; they are therefore neighbors and an always-increasing path exists. (the vertex with x_n minimum is unique and has $x - j = 2^j - 1$ for all $j < n$.)

Though one could pivot through all the vertices in this example, one can easily avoid doing so. There are, however, variations of it that stymie most pivot rules, so that using almost any one of them one can make an exponential number of pivots.

This implies that in worst case, the simplex algorithm can be terrible. Now let's consider some good news about the simplex algorithm.

Average Behavior To prove results about average behavior of algorithms, it is necessary to have some sample space of problems on which we can imagine applying the algorithm and averaging over the results. There is some question of what makes a reasonable sample space.

Suppose we have a given set of m constraints and n variables. This means we have an n dimensional space with $m + n$ constraint hyperplanes through it. These hyperplanes cut the space into regions, one of which is the feasible region of the problem. The other regions are feasible for problems in which the directions (less than or greater than) of some of the constraints are different from what they are in the original problem. We will average over all of the possible choices of direction for the constraints that lead to non-vanishing feasible regions;

In other words, we will average over all the regions defined by our constraints. We will assume that the origin was not a solution so that we got our space from what was originally an $n-1$ dimensional problem, by our trick, and that our version of the simplex algorithm will be one in which we rotate the objective function always pivoting to stay at the maximum.

We assume that our problem has no degeneracy whatsoever.

The quantity we are interested in is the number of edges that get pivoted over; we assume that for every region we are able to start at a minimum for the first objective function, and seek to rotate from its opposite, through the second objective function to the first, always staying at maxima, (now in practice we would not know where to begin except for our special problem, in which we happen to know that the origin is a maximum point for $-x_{n+1}$; but so what? If the average number of pivots per region is small for the given algorithm in this sample space, it is certainly *plausible* that there will be this same order of magnitude of pivots in its use on our specific problem.)

Since we are using a method that changes the objective function, we really ought to define a problem even when the region we deal with is unbounded and the objective function can go to infinity in it. We define the problem in that case to be: find the direction whose dot product with the objective function is the largest, among directions in which one can go to infinity while staying in the feasible region. With this definition, a solution point for a feasible region will be a vertex if the region is bounded, and may be a vertex or a direction if the feasible region is unbounded.

We shall show that *no edge of the entire system of regions gets used as pivot edge for more than one of its adjacent regions.* if the same algorithm with same two objective functions is used for all of them. It immediately follows that the total number of pivots for all regions is no more than the total number of edges; so that the average number of pivots needed is no more than the number of edges divided by the number of regions.

Suppose we have E edges, L lines, (which are each made up of of edges), R regions, and V vertices.

Let us first note that (1) $L = C(n+m, n-1)$, (2) $E = (M+1)C(n+m, n-1)$, (3) $V = C(n+m, n)$ and (4) $RZV + L$.

These are proven as follows: (1) any $n-1$ of the $n+m$ constraints defines a line; (2) the remaining $m+1$ constraints cut the line into $m+1$ segments or edges where we count opposite edges that go to infinity as one edge; (3) every n constraints determine a vertex;

and (4) if you choose a single arbitrary objective function, any vertex will be the solution point of one of its neighboring regions. Similarly, one side of the infinite edge of each line will be the best line for some adjacent region. The number of regions is therefore at least $V + L$.

(We can actually compute the number of regions precisely; If we denote the number of regions determined in n dimensions with $n+m$ constraints with no degeneracy by $r(n+m, n)$, we notice that the effect of the last hyperplane is to cut each region that intersects it into two parts, and the number that intersects it is $r(n + m - 1, n - 1)$. Thus, $r(n + m, n)$ is the number of regions that would exist without this last hyperplane, namely $r(n + m - 1, n)$ plus the $r(n + m - 1, n - 1)$. It creates this recursion:

$$r(n + m, n) = r(n + m - 1, n) + r(n + m - 1, n - 1)$$

may be solved here to yield the result:

$$r(n + m, n) = C(n + m, n) + C(n + m, n - 1) + \ldots C(n + m, 0).$$

The first two terms here are V and L. Putting (1) to (4) together we get that

$$E/R = (m + 1)n/(n + m - 1),$$

which tells us that if each edge can be pivoted on for only one region, there will be on the average of at most $(m + 1)n/(n + m + 1)$ pivots per region, which number is no bigger than the smaller of n and m.

To prove this all we have to do is show that no edge can be pivoted on twice; here is a proof of that:

(1) As we noted on page 11, an edge will be used in a pivot only when the objective function is normal to it, which happens only once in rotating the objective function 180 degrees.

(2) Suppose we pivot from vertex v to vertex w; just before the pivot only one region can have the vertex v as maximum point for the objective function; therefore the solution can move from v to w at the instant that the objective function becomes normal to the edge (w, v) only for that one same region.

We have therefore proven that the simplex algorithm, in some *average* sense, in this rotating implementation, has no more than a linear number of pivots.

This argument, which is still somewhat controversial, is due to Haimovits.

2 The Ellipsoid Algorithm for Linear Programming

About fifteen years ago, two Russians, Yudin and Nemerovsky, devised a simple method for solving not only linear programming problems, but more general problems of the same kind; namely those having a convex feasible region – one in which any point lying between two points in it is also in it.

The method is actually a method for finding a feasible point rather than an optimum seeking algorithm, and in it the constraints are assumed to be strict constraints instead of the "less than or equal" constraints we have considered so far. Actually, if we have a method allows us to find a feasible point when there is one, and to prove that there isn't one if there isn't then it alone is enough to allow us to find the optimum. Let us see this explicitly:

- We assume first that we know that our solution, if it exists, lies within a certain given n-sphere in our n dimensional space, of radius R and the objective function, $c * x$ lies between a lower bound of L and an upper bound of U.

- If we have a test of feasibility, we can add the constraint $c * x \geq (L + U)/G$ and test the original problem with this additional constraint with this test. If the answer is no, so that this problem is not feasible, then we have reduced the upper bound on the objective function form U to $(U + L)/2$. If we find a feasible point we have increased to this value, and have a feasible point besides.

- If we repeat this test with the new U and L a total of k times, the distance between U and L will diminish to $(U - L)2^{-k}$; if we get any feasible points along the way, they should converge to the optimum solution. Thus with a number of repetitions of the feasibility test that is roughly a logarithm of the desired accuracy we want for our answer, we can, with little luck, find a more or less optimal solution.

Now we give a high-level description of how we can go about testing for feasibility. We start with our n-sphere of radius R which we know contains all possible feasible points, if there are any.

We notice that this n-sphere becomes a simple kind of what is called an n-ellipsoid, if we make a scale change in one direction. Under such a change, hyperplanes remain hyperplanes, but they get tilted. Notice that this fact also means that such an n-ellipsoid (or any n-ellipsoid) can be made into an n-sphere by an appropriate scale change.

Our algorithm is based upon a simple and elementary geometric fact: that **there is an n-ellipsoid that contains half of an n-sphere, whose n-volume is less**

than that of the n-sphere itself by a factor of at least $r(n)$ with $r(n) < (1 - 1/(2n + 2))$.

In two dimensions, this says that we can put an ellipse around a semi-circle; which ellipse will have area that is less than that of the full circle by a factor of $r(2)$. We can therefore construct an algorithm as follows:

1. Test the center, c, of our present n-sphere to see if c is feasible. We do this by seeing if c obeys all the constraints. If it does, we are done since we have found a feasible point. If it does not, then we have found a constraint that is violated at c, and hence *is violated in at least half of the original n-sphere.*

2. Find the center of a new (reduced n-volume) n-ellipsoid that contains the part of the previous n-sphere that obeys this constraint, and make a change of scale so that this ellipsoid becomes an n-sphere again. Go back an repeat step 1 with this n-sphere, by testing its center.

3. Now either, as we repeat these steps, a center of one of our n-spheres will obey all constraints and be feasible, or the n-volume of the n-sphere produced, measured on the original length scale in all directions, will go to zero at least exponentially; after q iterations, it will have area that is the original times $r(n)^q$.

We will show that for strict constraints, after a polynomially bounded number of iterations either (1) we get to a solution, or (2) the original problem was infeasible. Let us examine what is really involved in this algorithm.

- Step one is quite easy; it simply involves evaluating the constraints at the current n-sphere center. If we find a violated constraint, we can from it the normal direction $\underline{d}$ to the constraint, and w, the distance from the current n-center to the violated constraint.

- Suppose we have determined a function $f(r, w)$, where r is the radius of the current n-sphere; that represents the distance from the center of the present n-sphere to the center of the next one. The new center will be located a distance $f(r, w)$ in the direction $\underline{d}$ from the old center. We will find such a function by use of elementary geometry.

- Also from elementary geometry we can determine the radius $r_d(r, w)$, of the new ellipsoid in the d direction, and its radius $r_0(r, w)$ in all other directions. We then make a scale change by a factor r_d/r_0, in the d-direction, in order to make our ellipsoid into a sphere again. We will have to apply this scale change to the constraints, to put them in a form appropriate to the new n-ellipsoid being an n-sphere.

This is all that is involved in the algorithm. The functions $f(r,w), r_d(r,w)$, and $r_0(r,w)$ are known and in fact we will write them down shortly. The steps of the algorithm involve evaluating constraints at our given center, which is a straightforward task; computing the location of the new center, which is done by adding a determined multiple of the violated constraint's normal to the old center location; and rescaling the constraints to remove the difference between the $\underline{d}$ direction and all the others in our ellipsoid. This last step is easy to do: if one has a constraint normal $\underline{q}$, one only has to replace it by $\underline{q} + t(\underline{q} \cdot \underline{d})\underline{d}$ where t is a known quantity. (This must be done for all the $n - m$ equality constraints, including those that say some x_j positive.)

In practice, the steps of the algorithm are very easy to perform; if anything they are easier to do than are our pivots in the simplex algorithm. What is surprising, is one can prove that this algorithm can take at most a number of steps that is polynomial in n and m and the number of places of accuracy one wants in the answer. The proof comes almost at once from the elementary geometric fact that one can put an n-ellipsoid around half of an n-sphere whose n-volume is less than that of the original n-sphere by a factor of $(1 - 1/(2n + 2)$ or less.

This implies that repetition of this step a number of times given by $2n^2 \log(A)$ will reduce the volume by a factor of at least A^n. This implies that in at least some direction, the radius has gone down to at most R/A. If we make A large enough, this will be smaller than the accuracy we care about, so that the component of our possible solution in this direction can be considered fixed, to this accuracy, at its value at the current center. Normally we would expect most components to be fixed similarly, but at least one must be, and that is enough for our proof.

We can show in theory, anyway, go back to the original sphere, fix the component in this direction and make everything into an $n - 1$ dimensional problem, and do everything again. Again we can determine another component. Eventually, after at most n such actions, we will determine all components of our potential solution to our desired accuracy, and either it is a solution or it isn't,

What is the worst possible performance of this algorithm?

Updating the constraints could take $(m + n)n$ steps, and we could have to repeat this updating, at worst, something like $2n^2 \log(A)$ times for each component, for, at most, something like $(m + n)n^4 \log(A)$ individual steps to test feasibility. One might have to repeat this whole mess another $\log(A)$ times to use it iteratively to solve for an optimum.

In practice, one hopes one can do much better than this, in the same way that the simplex algorithm requires many fewer pivots than the best upper bound one can prove for the number of pivots.

If, when doing the algorithm, you find a constraint that is badly violated at the current center, so that w/r is bigger than some constant c then *the radius* of your new n-sphere is down from r by a factor of $(1 - c^2)^{1/2}$ even if one merely moves the center onto the hyperplane of the constraint, without bothering with any scale change. This move requires no updating of the constraints at all so that it could be done in a linear number of steps, perhaps, for sparse constraints; and it replaces on the order of $n^2/\log(c^{-1})$ of the worst case iterations by only one, faster iteration. With any kind of luck, many of the variables settle down rather than only one, after getting the volume down, so that the last factor of n would not be expected to be needed most of the time. Putting all these possibilities together, the best one could hope for it would be $n(\log(A))^2$ iterations, if one could always find highly violated constraints, and used the method to find an optimal.

There is a trick to making the algorithm faster that may be worth performing here. What we want is to find a constraint that is badly violated; this constraint need not be one of the original constraints. By use of a little plane geometry, given two constraints that are both violated, find another, that contains their intersection, that is further away from the old center, and also violated. By iterating this trick, one keeps finding constraints violated further and further form our center, and can speed performance of the algorithm considerably.

To this time, nobody claims to be able to make this algorithm competitive with the simplex algorithm in practice. The optimistic estimate given above for the number of steps has not been realized in practice, at least so far. One is in trouble if one gets very near constraints; yet the first thing the algorithm does is drag you near to constraints; it seems as if it seeks out points at which it is close to many constraints but still far from the center.

3 Karmakar's Algorithm for Linear Programming

This section is based on notes written by Mark Haiman for 18.310

Although the simplex method, which we have discussed in detail, seems to work fairly quickly on practical linear programming problems, it can take an extremely long time on bad problems, as we have seen. In 1982, N. Karmakar of Bell Telephone Laboratories published a new linear programming algorithm. Karmakar's algorithm has a worst-case running time which is a polynomial function of the number $n+m$ of variables and constraints in the problem, in fact, $O((n+m)^{3.5})$.

More importantly, unlike the earlier polynomial-time "ellipsoid" algorithm, the running time is a *small enough* polynomial function to make Karmakar's algorithm a serious practical competitor to the simplex algorithm. How serious remains a matter of hot debate at the present moment, but for at least some types of problems, Karmakar's algorithm seems to do better. In particular, it can be set up to take advantage of *sparseness* in the problem: the situation when many entries of the matrix of constraint coefficients are zero. This is a common feature of practical problems, in which each constraint typically involves only a few of the variables.

Karmakar's algorithm is an *iterative* or *successive approximation* method. Unlike simplex, which tests potential solutions until it finds the exact optimum, Karmakar's approaches the optimum gradually, and never actually gets there. This may sound like a drawback, but actually it is not one. We know that the actual optimum is a vertex - a point defined by the vanishing of some of the variables and slack variables. As Karmakar's approaches the optimum vertex, therefore, certain of these variables will get close to zero. When we find that enough variables are near to zero, we can guess that setting them to zero gives exactly the true optimum. It is easy enough to set them to zero and check; if the resulting vertex happens not to be optimum, we can iterate some more and try again. In any case, we generally only want the answer to some fixed precision (10 digits, or whatever), and since (as we shall see), Karmakar's algorithm approaches the optimum exponentially, the extra running time is proportional to the extra precision. We couldn't really hope for more than that with any algorithm.

Before describing Karmakar's algorithm, we must explain how to rewrite our linear program in *homogeneous* form. (*Homogeneous* is a general-purpose mathematical word for polynomials and related expressions in which all terms are of the same degree. You may have seen it in differential equations.)

3.1 Homogeneous form of a linear program.

A linear program is in *homogeneous form* if it obeys the following three rules:

1. the program should be given as follows:

$$\text{minimize} \sum_{j=1}^{N} f_j u_j$$

   subject to the conditions

$$u_j \geq 0 (j = 1, ..., N)$$

$$\sum_{j=1}^{N} u_j = 1$$

$$\sum_{j=1}^{N} B_{i,j} u_j = 0 (i = 1, ..., M)$$

   .

2. the actual minimum value of the objective function $\sum_{j=1}^{n} f_j uj$ should be known ahead of time to be zero. (The problem, then, being not to find out what the minimum is, but for what values of the variables u_j it occurs.)

3. there should be some given point $(u_1^*, ..., u_N^*)$ which satisfies all the conditions, that is, a point of the feasible region.

Now this may seem a lot to ask for, but we shall now see that for any linear program given in standard form, we can concoct one in homogeneous form such that solving the homogeneous one gives the solution to the original one.

Let's consider requirement (3), that we have a feasible starting point. We already encountered that problem for the simplex method, and solved it by introducing an extra variable that makes any point you like feasible when it is large enough, and gives you back your original problem when you minimize it. We can do that here, too, as follows.

Suppose you are given a linear program in the form above (first rule). Add an extra variable u_{N+1} and let $B_{i,N+1} = -\sum_{j=1}^{N} B_{i,j}$. Now if we let $u_i^* = 1/(N+1)$ for every variable u_1 through u_N, we clearly have a feasible starting point for the new L. P. with this extra variable. For this L. P., we can take the objective function to be simply u_{N+1}. Finding a point where this takes its minimum of zero (by running Karmakar's on the new problem) gives a feasible starting point for the original problem, since the constraints of the new problem reduce to those of the old when $u_{N+1} = 0$. Then we run Karmakar's starting with this feasible point to solve the original problem In practice, we may already know a feasible starting point, and not have to run the problem twice like this.

Now we address requirements (1) and (2). Suppose the original problem is

$$\text{maximize } \sum_{j=1}^{n} c_j x_j$$

subject to

$$x_j \geq 0 (j = 1, ..., n)$$

$$\sum_{j=1}^{n} A_{i,j} \geq b_j (i = 1, ..., m).$$

As usual we introduce slack variables $s_i = b_i - \sum_j A_{i,j} x_j$ so that the program becomes (using bold-face letters for vectors and matrices whose entries are the corresponding quantities).

$$\text{maximize } \mathbf{c} \cdot \mathbf{x}$$

subject to

$$\begin{pmatrix} s \\ x \end{pmatrix} \geq \mathbf{0}$$

$$\begin{pmatrix} I & A \end{pmatrix} \begin{pmatrix} s \\ x \end{pmatrix} = \mathbf{b}.$$

Remember that this has a *dual* linear program:

$$\text{maximize } \mathbf{b} \cdot \mathbf{y}$$

subject to

$$\begin{pmatrix} t \\ y \end{pmatrix} \geq \mathbf{0}$$

$$\begin{pmatrix} -I & A^T \end{pmatrix} \begin{pmatrix} t \\ y \end{pmatrix} = \mathbf{c}.$$

Recall that the minimum of $\mathbf{b} \cdot \mathbf{y}$ is equal to the maximum of $\mathbf{c} \cdot \mathbf{x}$ and thus $\mathbf{b} \cdot \mathbf{y} - \mathbf{c} \cdot \mathbf{x} = \begin{pmatrix} b & -c \end{pmatrix} \cdot \begin{pmatrix} y \\ x \end{pmatrix}$ is always non-negative, and has a minimum of zero which occurs exactly at the simultaneous solution of the primal and dual problems (if these problems have solutions).

Therefore let us combine the primal and dual into the program

$$\text{minimize } \begin{pmatrix} 0 & b & 0 & -c \end{pmatrix} \cdot \begin{pmatrix} t \\ y \\ s \\ z \end{pmatrix}$$

subject to

$$\begin{pmatrix} t \\ y \\ s \\ x \end{pmatrix} \geq \mathbf{0}$$

$$\begin{pmatrix} -I & A^T & 0 & 0 \\ 0 & 0 & I & A \end{pmatrix} \begin{pmatrix} t \\ y \\ s \\ x \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix}.$$

Now we have satisfied the second equation for homogeneous form, that the actual minimum of the objective function is zero. We still don't quite have the equations as in the first condition, though. For that, let's first collect all the t, y, s, and x variables into one long list, say $v_1, v_2, ..., v_{2m+n}$ (there are n x and t variables; m y and s variables). Now, in order to get them to sum to one, and to get a zero vector on the right side of the matrix equation, let's let $u_j = v_j/(1 + \sum_j v_j)$ for

$j = 1, ..., 2(m+n)$, and $u_{2(m+n)+1} = 1/(1+\sum_j v_j)$. Thus the total number of u variables will be $N = 2(m+n)+1$. Obviously, $\sum_{j=1}^{N} u_j = 1$, and the equations

$$\begin{pmatrix} -I & A^T & 0 & 0 \\ 0 & 0 & I & A \end{pmatrix} \begin{pmatrix} t \\ y \\ s \\ x \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix}.$$

become

$$\begin{pmatrix} -I & A^T & 0 & 0 \\ 0 & 0 & I & A \end{pmatrix} \mathbf{u} = \mathbf{0}$$

These equations are in the form

$$\mathbf{B}\mathbf{u} = \mathbf{0}$$

$$\sum_{j=1}^{N} u_j = 1$$

as required by the first condition. As for the objective function, since the vector $(u_1, u_2, ..., u_{N-1})$ is a positive scalar multiple (by the factor $1/(1+\sum_j v_j)$) of the original vector of t, y, s, and x variables, the condition

$$\text{minimize} \begin{pmatrix} 0 & b & 0 & -c \end{pmatrix} \cdot \begin{pmatrix} t \\ y \\ s \\ z \end{pmatrix}$$

can be replaced by the condition

$$\text{minimize} \begin{pmatrix} 0 & b & 0 & -c & 0 \end{pmatrix} \cdot \mathbf{u}$$

which will attain its minimum of zero when the original t, y, s, x objective function does. We now have a program which satisfies the first and second requirements, and if we solve it we can get back the solution of our original problem (and its dual).

Notice that in all of this that if the original coefficient matrix $\mathbf{A}$ is *sparse*, that is, has mostly zeroes in it, this will be the case for the matrix $\mathbf{B}$ of the homogeneous linear program as well.

3.2 The Potential Function

Rather than directly optimizing the objective function, Karmakar's algorithm is based on trying to minimize the *potential function*

$$g(\mathbf{u}) = \frac{(\sum_j f_j u_j)^N}{u_1 u_2 \cdots u_N}.$$

The potential function has certain characteristics that should be pointed out:

1. $g(\mathbf{u})$ is unchanged if all the u_j are multiplied by the same thing, c, since then both numerator and denominator contain a factor of c^N.

2. As you approach the optimum, the numerator goes to zero as an Nth power. The denominator also goes to zero, since some of the u_j will be zero at the optimum. However, not all of the u_j can be zero at once (since their sum is 1), so the denominator will go to zero as a power less than N. Therefore the whole potential function approaches zero at the optimum.

3. As you near any constraint boundary $u_j = 0$, without also approaching the optimum, $g(\mathbf{u})$ goes to ∞. Thus proceeding in the direction of decreasing potential function tends to steer you away from irrelevant constraints and towards the optimum.

4. In the next section we will describe the iterative step of Karmakar's algorithm, which starts with the current point $\mathbf{u}^*$ and finds a new point $\mathbf{u}^+$ for which $g(\mathbf{u}^+) \leq \frac{2}{e} g(\mathbf{u}^*)$. Thus the potential function is reduced by a constant factor at each step, and the optimum is approached exponentially, with the number of steps required to reach any given precision proportional to N, since the objective function goes as the Nth root of the potential function. Since each step, as we shall see, requires solving a system of linear equations, which can be done in $O(N^{2.5})$ steps, we get a guaranteed maximum overall running time of $O(N^{3.5})$.

Now the theoretical $N^{2.5}$ time to solve a system of linear equations comes from the theoretical $N^{2.5}$ time for matrix multiplication, which as we have seen is highly theoretical indeed! However, one can easily solve systems in N^3 time by Gaussian elimination, and a bit faster by iterative methods. This is not much worse than the theoretical $N^{2.5}$. Moreover, for sparse systems, the iterative methods can be very fast, and Karmakar's algorithm has the advantage that unlike simplex, it does not destroy sparseness in the problem.

Since simplex seems to take about N^3 time, you can see that it is a close call between the two. Indeed, since one iteration of Karmakar generally takes not much less time than a full run of simplex, simplex would probably be a clear winner were it not for the fact (or, at least, *claim* by some people) that Karmakar's algorithm actually needs very few iterations - far less than the maximum guaranteed by the theory. Probably it will be several years before experience indicates one method or the other as better, or indicates circumstances in which each is more appropriate.

3.3 The Iterative Step

We now show how to go from $\mathbf{u}^*$ to a new point $\mathbf{u}^+$ with a reduced value of the potential function.

First, we *rescale* the problem to put our current point $\mathbf{u}^*$ at the "center", where all N coordinates are equal to $1/N$. This is done by defining

$$v_j = \frac{(u_j/u_j^*)}{\sum_j (u_j u_j^*)}$$

We have taken the u's, scaled them relative to their values at the current point $\mathbf{u}^*$, and normalized the whole thing so that it sums to 1. Thus we have $v_j^* = 1/N$ and the v_j satisfy the conditions

$$\sum_j (B_{i,j} u_j^*) v_j = 0$$

$$\sum_j v_j = 1,$$

and the potential function becomes

$$g(\mathbf{v}) = \frac{(\sum_j (f_j u_j^*) v_j)^N}{v_1 v_2 \cdots v_N}.$$

In short, this change of variable modifies the original L.P. by scaling the jth column by the factor u_j^*. These new conditions are equivalent to the old because the vector $\mathbf{v}$ is a multiple of the vector $(u_1/u_1^*, ..., u_N/u_N^*)$ and the equations (except sum = 1) and potential function are unaffected by scalar multiplication.

Now we ask, in what direction from $\mathbf{v}^*$ is the potential function decreasing most rapidly? Because of the symmetry (all $v_j^* = 1/N$), the gradient of the denominator at $\mathbf{v}^*$ is in the direction of (1, 1,..., 1), or perpendicular to the hyperplane $\sum_j v_j = 1$. So the denominator is locally constant in this plane, and the direction of most rapid decrease is controlled entirely by the numerator: it is minus the gradient of the numerator, *i.e.*, in the direction $\mathbf{d} = (-f_1 u_1^*, ..., -f_N u_N^*)$. We would like to move from $\mathbf{v}^*$ in that direction - but if we did we would probably violate some of the equations of our linear program. Therefore, we must *project* the vector $\mathbf{d}$ that gives the *best* direction into the space of *legal* directions $\mathbf{\Delta v}$ which satisfy the conditions

$$\sum_j (B_{i,j} u_j^*) \Delta v_j = 0$$

$$\sum_j \Delta v_j = 0$$

(the second equation is because $\sum_j v_j = 1$ is not allowed to change). We can write these equations as a single matrix equation

$$\hat{\mathbf{B}}\boldsymbol{\Delta}\mathbf{v} = \mathbf{0},$$

where $\hat{B}$ is the rescaled $\mathbf{B}$ with a row of ones added.

For $\boldsymbol{\Delta}\mathbf{v}$ to be the projection of $\mathbf{d}$, $\boldsymbol{\Delta}\mathbf{v}$ should be perpendicular to the difference between itself and $\mathbf{d}$: $(\mathbf{d} - \boldsymbol{\Delta}\mathbf{v}) \cdot \boldsymbol{\Delta}\mathbf{v} = \mathbf{0}$. The equation $\hat{\mathbf{B}}\boldsymbol{\Delta}\mathbf{v} = \mathbf{0}$ says that $\boldsymbol{\Delta}\mathbf{v}$ is perpendicular to all the rows of $\hat{\mathbf{b}}$. Therefore we take $(\mathbf{d} - \boldsymbol{\Delta}\mathbf{v})$ to be a linear combination of these rows: $(\mathbf{d} - \boldsymbol{\Delta}\mathbf{v}) = \hat{\mathbf{B}}^T\mathbf{y}$ for some vector $\mathbf{y}$. Then, multiplying by $\hat{\mathbf{B}}$, we have $\hat{\mathbf{B}}\mathbf{d} = \hat{\mathbf{B}}\hat{\mathbf{B}}^T\mathbf{y}$, which is a system of $M + 1$ equations and $M + 1$ unknowns that can be solved for $\mathbf{y}$, and thus we find $\boldsymbol{\Delta}\mathbf{v}$.

Karmakar's iteration step is to adjust $\mathbf{v}$ by a certain amount in the direction Δv. Converting back into u variables (by multiplying each new v_j by u_j^*, then scaling so $\sum_j u_j = 1$) gives the new point $\mathbf{u}^+$.

It remains to decide what that "certain amount" should be, and to show that it has the desired effect on the potential function.

Let's summarize the situation. We have our problem expressed in variables v_j, with the current point given by $v_j^* = 1/N$. We have found a vector $\boldsymbol{\Delta}\mathbf{v}$ such that moving in that direction from $\mathbf{v}^*$ keeps the equations satisfied, and causes the potential function $(\sum_j (f_j u_j^*) v_j)^N / (v_1 v_2 \cdots v_N)$ to decrease as rapidly as possible, subject to those equations. This direction is also the direction of most rapid decrease of the linear function $\sum_j (f_j u_j^*) v_j$. Also, we know that at the true optimum, wherever it is, the potential function, and hence the linear function $\sum_j (f_j u_j^*) v_j$, is zero.

Now the farthest away you can be from the current point $(1/N, 1/N, ..., 1/N)$ and still have all coordinates positive and sum to one, is to be at the point $(1, 0, 0, ..., 0)$ or a symmetric point, and the distance to such a point is

$$\sqrt{(1 - 1/N)^2 + (N - 1)(1/N)^2} = \sqrt{(N - 1)/N} < 1.$$

On the other hand, the *nearest* you can be and have some coordinate not positive, and all sum to one, is to be at $(0^-, 1/(N-1), 1/(N-1), ..., 1/(N-1))$ or a symmetric point, and the distance to such a point is

$$\sqrt{(1/N)^2 + (N-1)(1/(N-1) - 1/N)^2} = \sqrt{1/[N(N-1)]} > 1.$$

Therefore it is definitely safe to move a distance, say, of $\frac{1}{2N}$ in the $\mathbf{\Delta v}$ direction: this cannot make any v_j negative.

Such a move must also reduce the linear function $\sum_j (f_j u_j^*) v_j$ by at least a fraction $\frac{1}{2N}$ of its original value, since this is the distance actually moved, whereas the distance required to move to reduce it to zero is at most 1. So this move decreases the numerator of the potential function by a factor at least as small as $(1 - \frac{1}{2N})^N$. note that for large N, this factor is about $e^{-1/2}$.

Now we check that the denominator of the potential function is *not* reduced very much. To find out how much, observe that we are only allowed to move distance $\frac{1}{2N}$ from $(1/N, 1/N, ..., 1/N)$, that is, to a point $(1/N + \epsilon_1, ..., 1/N + \epsilon_N)$ where $\sum_j \epsilon_j^2 = \frac{1}{4N^2}$, and $\sum_j \epsilon_j = 0$ (to preserve the sum = 1 condition).

We want to know the minimum of $\prod_j (1/N + \epsilon_j) / \prod_j (1/N) = \prod_j (1 + N\epsilon_j)$, under these conditions. By the method of Lagrange multipliers, for such a minimum, the gradient of the quantity to be minimized must be a linear combination of the gradients of the contrasting conditions. The gradient of the function $\prod_j (1 + N\epsilon_j)$ is a vector whose jth component is $N \prod_{i \neq j} (1 + N\epsilon_i)$; this is proportional to

$$\mathbf{h} = (\frac{1}{1 + N\epsilon_1}, ..., \frac{1}{1 + N\epsilon_N}).$$

The gradient of the condition on $\sum_j \epsilon_j^2$ is $\sum_j 2\epsilon_j$, proportional to

$$\mathbf{p} = (\epsilon_1, ..., \epsilon_N)$$

The gradient of the condition on $\sum_j \epsilon_j$ is

$$\mathbf{q} = (1, ..., 1).$$

At the minimum, we must have

$$\mathbf{h} = \alpha \mathbf{p} + \beta \mathbf{q}$$

or

$$\frac{1}{1 + N\epsilon_j} = \alpha \epsilon_j + \beta$$

$$N\alpha\epsilon_j^2 + (\alpha + N\beta)\epsilon_j + \beta - 1 = 0$$

together with the constraining conditions

$$\sum_j \epsilon_j = 0$$

$$\sum_j \epsilon_j^2 = \frac{1}{4N^2}.$$

Since all the ϵ_j satisfy the same quadratic equation, they must fall into two groups, with, say K of them being negative and equal, and the other $N-K$ being positive and equal. You can verify that the minimizing case occurs when one is negative, say $-\rho$, and the rest are positive, equal to $\rho/(N-1)$. We must have $\rho^2+(N-1)\rho^2/(N-1)^2 = \frac{1}{4N^2}$ so $\rho = \frac{\sqrt{N-1}}{2N^{3/2}} \approx \frac{1}{2N}$, and $\prod_j(1+N\epsilon_j) = (1-N\rho)(1+N\rho/(N-1))^{N-1} \approx (1-N\rho)e^{N\rho} \approx \frac{1}{2}e^{1/2}$ for reasonably large N.

So the numerator in the potential function decreases to *at most* a factor of $e^{-1/2}$ times its former value, while the denominator decreases to *no less than* $\frac{1}{2}e^{1/2}$ of its former value. Therefore the whole potential function decreases to *at most* $e^{-1/2}/(\frac{1}{2}e^{1/2}) = 2/e$ of its previous value at every iteration, independent of the number of variables. Since the objective function goes as the Nth root of the potential function, you can see that the number of iterations to reach the optimum, to any given precision, is proportional to N.

3.4 Summary of Karmakar's algorithm

First, use duality to write your linear program in homogeneous form. If you don't have a feasible starting point, add an extra variable and pre-run the algorithm on the modified L.P. to find one.

Now, repeat the following steps until the objective function is near zero, to within whatever precision you require. u_j^* are the coordinates of your current point.

1. Form the desired-direction vector $\mathbf{d}$ whose coordinates are $d_j = -f_j u_j^*$. (f_j are the coordinates of the objective function).

2. Form the rescaled matrix $\hat{\mathbf{B}}$ by multiplying the jth column of the original constraint coefficient matrix $\mathbf{B}$ by u_j, for each j, and then adding a row of ones.

3. Project $\mathbf{d}$ to get the vector $\mathbf{\Delta v}$ as follows: solve the system of linear equations

$$\hat{\mathbf{B}}\hat{\mathbf{B}}^T\mathbf{y} = \hat{\mathbf{B}}\mathbf{d}$$

for y. Then set

$$\mathbf{\Delta v} = \mathbf{d} - \hat{\mathbf{B}}\mathbf{y}$$

4. Normalize $\boldsymbol{\Delta}\mathbf{v}$ to a length of $\frac{1}{2N}$ and add the result to the vector $(1/N, 1/N, ..., 1/N)$. This is your new $\mathbf{v}$.

5. Convert back to $\mathbf{u}$ variables by multiplying each v_j by u_j, then scaling the resulting vector so that the coordinates sum to 1. The result is your next current point.