CHAPTER 4

Efficiency of the Simplex Method

In the previous chapter, we saw that the simplex method (with appropriate pivoting rules to guarantee no cycling) will solve any linear programming problem for which an optimal solution exists. In this chapter, we investigate just how fast it will solve a problem of a given size.

1. Performance Measures

Performance measures can be broadly divided into two types:

- Worst case
- Average case.

As its name implies, a worst-case analysis looks at all problems of a given "size" and asks how much effort is needed to solve the hardest of these problems. Similarly, an average-case analysis looks at the average amount of effort, averaging over all problems of a given size. Worst-case analyses are generally easier than average-case analyses. The reason is that, for worst-case analyses, one simply needs to give an upper bound on how much effort is required and then exhibit a specific example that attains this bound. However, for average-case analyses, one must have a stochastic model of the space of "random linear programming problems" and then be able to say something about the solution effort averaged over all the problems in the sample space. There are two serious difficulties here. The first is that it is not clear at all how one should model the space of random problems. Secondly, given such a model, one must be able to evaluate the amount of effort required to solve every problem in the sample space.

Therefore, worst-case analysis is more tractable than average-case analysis, but it is also less relevant to a person who needs to solve real problems. In this chapter, we will start by giving a detailed worst-case analysis of the simplex method using the largest-coefficient rule to select the entering variable. We will then present and discuss the results of some empirical studies in which millions of linear programming problems were generated randomly and solved by the simplex method. Such studies act as a surrogate for a true average-case analysis.

2. Measuring the Size of a Problem

Before looking at worst cases, we must discuss two issues. First, how do we specify the size of a problem? Two parameters come naturally to mind: m and n. Often, we simply use these two numbers to characterize the size a problem. However, we should mention some drawbacks associated with this choice. First of all,

R.J. Vanderbei, *Linear Programming*, International Series in Operations Research & Management Science 196, DOI 10.1007/978-1-4614-7630-6_4,
© Springer Science+Business Media New York 2014

it would be preferable to use only one number to indicate size. Since the data for a problem consist of the constraint coefficients together with the right-hand side and objective function coefficients, perhaps we should use the total number of data elements, which is roughly mn.

The product mn isn't bad, but what if many or even most of the data elements are zero? Wouldn't one expect such a problem to be easier to solve? Efficient implementations do indeed take advantage of the presence of lots of zeros, and so an analysis should also account for this. Hence, a good measure might be simply the number of nonzero data elements. This would definitely be an improvement, but one can go further. On a computer, floating-point numbers are all the same size and can be multiplied in the same amount of time. But if a person is to solve a problem by hand (or use unlimited precision computation on a computer), then certainly multiplying 23 by 7 is a lot easier than multiplying 23,453.2352 by 86,833.245643. So perhaps the best measure of a problem's size is not the number of data elements, but the actual number of bits needed to store all the data on a computer. This measure is popular among most computer scientists and is usually denoted by L.

However, with a little further abstraction, the size of the data, L, is seen to be ambiguous. As we saw in Chapter 1, real-world problems, while generally large and sparse, usually can be described quite simply and involve only a small amount of true input data that gets greatly expanded when setting the problem up with a constraint matrix, right-hand side, and objective function. So should L represent the number of bits needed to specify the nonzero constraint coefficients, objective coefficients, and right-hand sides, or should it be the number of bits in the original data set plus the number of bits in the description of how this data represents a linear programming problem? No one currently uses this last notion of problem size, but it seems fairly reasonable that they should (or at least that they should seriously consider it). Anyway, our purpose here is merely to mention that these important issues are lurking about, but, as stated above, we shall simply focus on m and n to characterize the size of a problem.

3. Measuring the Effort to Solve a Problem

The second issue to discuss is how one should measure the amount of work required to solve a problem. The best answer is the number of seconds of computer time required to solve the problem, using the computer sitting on one's desk. Unfortunately, there are (hopefully) many readers of this text, not all of whom use the exact same computer. Even if they did, computer technology changes rapidly, and a few years down the road everyone would be using something entirely different. It would be nice if the National Institute of Standards and Technology (the government organization in charge of setting standards, such as how many threads/inch a standard light bulb should have) would identify a standard computer for the purpose of benchmarking algorithms, but, needless to say, this is not very likely. So the time needed to solve a problem, while the most desirable measure, is not the most practical one here. Fortunately, there is a fairly reasonable substitute. Algorithms are generally iterative processes, and the time to solve a problem can be factored into the number of iterations required to solve the problem times the amount of time required to do each iteration. The first factor, the number of iterations, does not depend on the computer and so is a reasonable surrogate for the actual time. This surrogate is useful when comparing various algorithms within the same general class of algorithms, in which the time per iteration can be expected to be about the same among the algorithms; however, it becomes meaningless when one wishes to compare two entirely different algorithms. For now, we shall measure the amount of effort to solve a linear programming problem by counting the number of iterations, i.e. pivots, needed to solve it.

4. Worst-Case Analysis of the Simplex Method

How bad can the simplex method be in the worst case? Well, we have already seen that for some pivoting rules it can cycle, and hence the worst-case solution time for such variants is infinite. However, what about noncycling variants of the simplex method? Since the simplex method operates by moving from one basic feasible solution to another without ever returning to a previously visited solution, an upper bound on the number of iterations is simply the number of basic feasible solutions, of which there can be at most

$$\binom{n+m}{m}.$$

For a fixed value of the sum $n+m$, this expression is maximized when $m=n$. And how big is it? It is not hard to show that

$$\frac{1}{2n}2^{2n} \le \binom{2n}{n} \le 2^{2n}$$

(see Exercise 4.9). It should be noted that, even though typographically compact, the expression 2^{2n} is huge even when n is not very big. For example, for $n=25$, we have $2^{50} = 1.1259 \times 10^{15}$.

Our best chance for finding a bad example is to look at the case where $m=n$. In 1972, V. Klee and G.J. Minty were the first to discover an example in which the simplex method using the largest coefficient rule requires $2^n - 1$ iterations to solve. The example is quite simple to state:

$$(4.1)\qquad \begin{array}{lll} \text{maximize} & \displaystyle\sum_{j=1}^{n} 10^{n-j} x_j & \\ \text{subject to} & \displaystyle 2\sum_{j=1}^{i-1} 10^{i-j} x_j + x_i \le 100^{i-1} & i = 1,2,\ldots,n \\ & x_j \ge 0 & j = 1,2,\ldots,n. \end{array}$$

It is instructive to look closely at the constraints. The first three constraints are

$$\begin{array}{rcr} x_1 \qquad\qquad & \le & 1 \\ 20x_1 + \quad x_2 \qquad & \le & 100 \\ 200x_1 + 20x_2 + x_3 & \le & 10{,}000. \end{array}$$

The first constraint simply says that x_1 is no bigger than one. With this in mind, the second constraint says that x_2 has an upper bound of about 100, depending on how big x_1 is. Similarly, the third constraint says that x_3 is roughly no bigger than 10,000 (again, this statement needs some adjustment depending on the sizes of x_1 and x_2). Therefore, the constraints are approximately just a set of upper bounds, which means that the feasible region is virtually a stretched n-dimensional hypercube[1]:

$$\begin{aligned} 0 \leq &\quad x_1 &\leq 1 \\ 0 \leq &\quad x_2 &\leq 100 \\ &\quad \vdots & \\ 0 \leq &\quad x_n &\leq 100^{n-1}. \end{aligned}$$

For this reason, the feasible region for the Klee–Minty problem is often referred to as the Klee–Minty cube. An n-dimensional hypercube has 2^n vertices, and, as we shall see, the simplex method with the largest-coefficient rule will start at one of these vertices and visit every vertex before finally finding the optimal solution.

In order to gain a deeper understanding of the Klee–Minty problem, we first replace the specific right-hand sides, 100^{i-1}, with more generic values, b_i, having the property that

$$1 = b_1 \ll b_2 \ll \cdots \ll b_n.$$

As in the previous chapter, we use the expression $a \ll b$ to mean that a is so much smaller than b that no factors multiplying a and dividing b that arise in the course of applying the simplex method to the problem at hand can ever make the resulting a as large as the resulting b. Hence, we can think of the b_i's as independent variables for now (specific values can be chosen later). Next, it is convenient to change each right-hand side replacing b_i with

$$\sum_{j=1}^{i-1} 10^{i-j} b_j + b_i.$$

Since the numbers b_j, $j = 1, 2, \ldots, i-1$ are "small potatoes" compared with b_i, this modification to the right-hand sides amounts to a very small perturbation. The right-hand sides still grow by huge amounts as i increases. Finally, we wish to add a constant to the objective function so that our generalized Klee–Minty problem can finally be written as

$$\begin{array}{lllr} & \text{maximize} & \displaystyle\sum_{j=1}^{n} 10^{n-j} x_j - \frac{1}{2} \sum_{j=1}^{n} 10^{n-j} b_j & \\ (4.2) & \text{subject to} & \displaystyle 2 \sum_{j=1}^{i-1} 10^{i-j} x_j + x_i \leq \sum_{j=1}^{i-1} 10^{i-j} b_j + b_i & i = 1, 2, \ldots, n \\ & & \qquad\qquad x_j \geq 0 & j = 1, 2, \ldots, n. \end{array}$$

[1]More precisely, a hyperrectangle.

In Exercise 4.7, you are asked to prove that this problem takes $2^n - 1$ iterations. To start to get a handle on the proof, here are the seven iterations that one gets with $n = 3$. The initial dictionary is

$$
\begin{array}{rl}
\zeta = & -\frac{100}{2}b_1 - \frac{10}{2}b_2 - \frac{1}{2}b_3 + 100x_1 + 10x_2 + x_3 \\ \hline
w_1 = & b_1 - x_1 \\
w_2 = & 10b_1 + b_2 - 20x_1 - x_2 \\
w_3 = & 100b_1 + 10b_2 + b_3 - 200x_1 - 20x_2 - x_3,
\end{array}
$$

which is feasible. Using the largest coefficient rule, the entering variable is x_1. From the fact that each subsequent b_i is huge compared with its predecessor it follows that w_1 is the leaving variable. After the first iteration, the dictionary reads

$$
\begin{array}{rl}
\zeta = & \frac{100}{2}b_1 - \frac{10}{2}b_2 - \frac{1}{2}b_3 - 100w_1 + 10x_2 + x_3 \\ \hline
x_1 = & b_1 - w_1 \\
w_2 = & -10b_1 + b_2 + 20w_1 - x_2 \\
w_3 = & -100b_1 + 10b_2 + b_3 + 200w_1 - 20x_2 - x_3.
\end{array}
$$

Now, x_2 enters and w_2 leaves, so after the second iteration we get:

$$
\begin{array}{rl}
\zeta = & -\frac{100}{2}b_1 + \frac{10}{2}b_2 - \frac{1}{2}b_3 + 100w_1 - 10w_2 + x_3 \\ \hline
x_1 = & b_1 - w_1 \\
x_2 = & -10b_1 + b_2 + 20w_1 - w_2 \\
w_3 = & 100b_1 - 10b_2 + b_3 - 200w_1 + 20w_2 - x_3.
\end{array}
$$

After the third iteration

$$
\begin{array}{rl}
\zeta = & \frac{100}{2}b_1 + \frac{10}{2}b_2 - \frac{1}{2}b_3 - 100x_1 - 10w_2 + x_3 \\ \hline
w_1 = & b_1 - x_1 \\
x_2 = & 10b_1 + b_2 - 20x_1 - w_2 \\
w_3 = & -100b_1 - 10b_2 + b_3 + 200x_1 + 20w_2 - x_3.
\end{array}
$$

After the fourth iteration

$$
\begin{array}{rl}
\zeta = & -\frac{100}{2}b_1 - \frac{10}{2}b_2 + \frac{1}{2}b_3 + 100x_1 + 10w_2 - w_3 \\ \hline
w_1 = & b_1 - x_1 \\
x_2 = & 10b_1 + b_2 - 20x_1 - w_2 \\
x_3 = & -100b_1 - 10b_2 + b_3 + 200x_1 + 20w_2 - w_3.
\end{array}
$$

After the fifth iteration

$$
\begin{array}{rl}
\zeta = & \frac{100}{2}b_1 - \frac{10}{2}b_2 + \frac{1}{2}b_3 - 100w_1 + 10w_2 - w_3 \\ \hline
x_1 = & b_1 - w_1 \\
x_2 = & -10b_1 + b_2 + 20w_1 - w_2 \\
x_3 = & 100b_1 - 10b_2 + b_3 - 200w_1 + 20w_2 - w_3.
\end{array}
$$

After the sixth iteration

$$
\begin{array}{rl}
\zeta = & -\frac{100}{2}b_1 + \frac{10}{2}b_2 + \frac{1}{2}b_3 + 100w_1 - 10x_2 - w_3 \\ \hline
x_1 = & b_1 - w_1 \\
w_2 = & -10b_1 + b_2 + 20w_1 - x_2 \\
x_3 = & -100b_1 + 10b_2 + b_3 + 200w_1 - 20x_2 - w_3.
\end{array}
$$

And, finally, after the seventh iteration, we get

$$\begin{array}{rrrrrrr}
\zeta = & \frac{100}{2}b_1 + & \frac{10}{2}b_2 + & \frac{1}{2}b_3 & - 100x_1 & - 10x_2 & - w_3 \\ \hline
w_1 = & b_1 & & & - x_1 & & \\
w_2 = & 10b_1 + & b_2 & & - 20x_1 & - x_2 & \\
x_3 = & 100b_1 + & 10b_2 + & b_3 & - 200x_1 & - 20x_2 & - w_3,
\end{array}$$

which is, of course, optimal.

A few observations should be made. First, every pivot is the swap of an x_j with the corresponding w_j. Second, every dictionary looks just like the first one with the exception that the w_i's and the x_i's have become intertwined and various signs have changed (see Exercise 4.6).

Also note that the final dictionary could have been reached from the initial dictionary in just one pivot if we had selected x_3 to be the entering variable. But the largest-coefficient rule dictated selecting x_1. It is natural to wonder whether the largest-coefficient rule could be replaced by some other pivot rule for which the worst-case behavior would be much better than the 2^n behavior of the largest-coefficient rule. So far no one has found such a pivot rule. However, no one has proved that such a rule does not exist either.

Finally, we mention that one desirable property of an algorithm is that it be scale invariant. This means that should the units in which one measures the decision variables in a problem be changed, the algorithm would still behave in exactly the same manner. The simplex method with the largest-coefficient rule is not scale invariant. To see this, consider changing variables in the Klee–Minty problem by putting

$$\bar{x}_j = 100^{j-1}x_j.$$

In the new variables, the initial dictionary for the $n = 3$ Klee–Minty problem becomes

$$\begin{array}{rrrrrrr}
\zeta = & -\frac{100}{2}b_1 - & \frac{10}{2}b_2 - & \frac{1}{2}b_3 & + 100\bar{x}_1 & + 1000\bar{x}_2 & + 10000\bar{x}_3 \\ \hline
w_1 = & b_1 & & & - \bar{x}_1 & & \\
w_2 = & 10b_1 + & b_2 & & - 20\bar{x}_1 & - \bar{x}_2 & \\
w_3 = & 100b_1 + & 1000b_2 + & b_3 & - 200\bar{x}_1 & - 2000\bar{x}_2 & - 10000\bar{x}_3.
\end{array}$$

Now, the largest-coefficient rule picks variable x_3 to enter. Variable w_3 leaves, and the method steps to the optimal solution in just one iteration. There exist pivot rules for the simplex method that are scale invariant. But Klee–Minty-like examples have been found for most proposed alternative pivot rules (whether scale invariant or not). In fact, it is an open question whether there exist pivot rules for which one can prove that no problem instance requires an exponential number of iterations (as a function of m or n).

5. Empirical Average Performance of the Simplex Method

To investigate the empirical average case performance of the simplex method, we generated a large number of random linear programming problems and solved each of them using the simplex method. There are many ways to generate random problems. In this section, we consider one such method and analyze the results.

As we have so dramatically seen, the number of iterations can depend on the specific choice of pivot rule. In the code discussed below, we choose the entering variable to be the one with the largest coefficient. The way we generate random problems makes it unlikely that there will be ties in the choice of largest coefficient (at least after the first pivot). So, even though the program makes a choice, it is not terribly important to articulate what that choice is. Similarly, there is little chance for a tie when choosing a leaving variable, so we do not dwell on this matter either.

We list below the source code. The program is written in a language called MATLAB—a widely used language whose source code is fairly easy to read, even for those not familiar with the language. So, to get started, here's how we initialize the data describing a linear programming problem:

```
m = round(10*exp(log(100)*rand()));
n = round(10*exp(log(100)*rand()));

sigma = 10;
A = round(sigma*(randn(m,n)));
b = round(sigma*abs(randn(m,1)));
c = round(sigma*randn(1,n));
```

Here, `rand()` generates a (pseudo) random number uniformly distributed on the interval $[0, 1]$ and `round()` simply rounds a number to its nearest integer value. The formulas for m and n produce numbers between 10 and 1,000. The formula may seem more complicated than one would expect. For example, one might suggest this simpler formula: `m = round(10 + 990*rand())`. There is a good reason for the more complicated version. We would like about half of the problems to be between 10 and 100 and the other half to be between 100 and 1,000. Using the simple scheme suggested above, only about 10 % of the numbers would be between 10 and 100. The vast majority would be between 100 and 1,000. So, what we want is to have our numbers uniformly distributed when viewed on a logarithmic scale. Our formula for m and n achieves this logarithmically-uniform distribution.

The vector c of objective function coefficients is generated using the function `randn()`. This function is like `rand()` but, instead of generating numbers with a uniform distribution, it generates numbers with a Gaussian (aka "normal") distribution with mean 0 and standard deviation 1. Multiplying such a variable by `sigma`, which is set to 10, increases the standard deviation to 10. The arguments `(1,n)` passed to `randn()` tells the random number generator not to produce just one such number but rather to produce a $1 \times n$ matrix, i.e. a row vector, of independent instances of these random variables. There is no particular reason to round the coefficients in the row vector c to be integers. The only reason this was done was to make the random problems seem slightly more realistic since many/most real-world problems involve data that is mostly integer-valued. The matrix A and the right-hand side vector b are generated in a manner similar to how c is generated. But, note that the formula for b involves the `abs()` which returns absolute values so that all elements of b are non-negative. This minor, but important, twist is done to ensure that the starting dictionary is feasible. Here's the main pivot loop:

```
iter = 0;
while max(c) > eps,
   % pick largest coefficient
   [cj, col] = max(c);
   Acol = A(:,col);

   % select leaving variable
   if sum(Acol<-eps) == 0,
         opt = -1;  % unbounded
         'unbounded'
         break;
   end
   nums = b.*(Acol<-eps);
   dens = -Acol.*(Acol<-eps);
   [t, row] = min(nums./dens);
   Arow = A(row,:);

   a = A(row,col); % pivot element

   A = A - Acol*Arow/a;
   A(row,:) = -Arow/a;
   A(:,col) = Acol/a;
   A(row,col) = 1/a;

   brow = b(row);
   b = b - brow*Acol/a;
   b(row) = -brow/a;

   ccol = c(col);
   c = c - ccol*Arow/a;
   c(col) = ccol/a;

   iter = iter+1;
end
```

In this code, the expression `max(c)` computes the maximum value of the elements of the vector `c`. It returns both the maximum value and the index at which this value was attained (the first such index if there are more than one). So, `cj` is the maximal coefficient and `col` is the index at which this maximum is attained. Hence `col` is the entering column. Given the matrix `A`, the expression `A(:,col)` denotes the column vector consisting of the elements from the `col` column of `A`. Hence, `Acol` denotes the column of the dictionary associated with the entering variable. The next lines of this short code selects the leaving variable, which is in the row called `row`. Given the entering column and the leaving row, all that remains is to update the coefficients in the objective function, the right-hand side, and the array of coefficients `A`. The last few lines encode exactly what one needs to do to carry out a pivot.

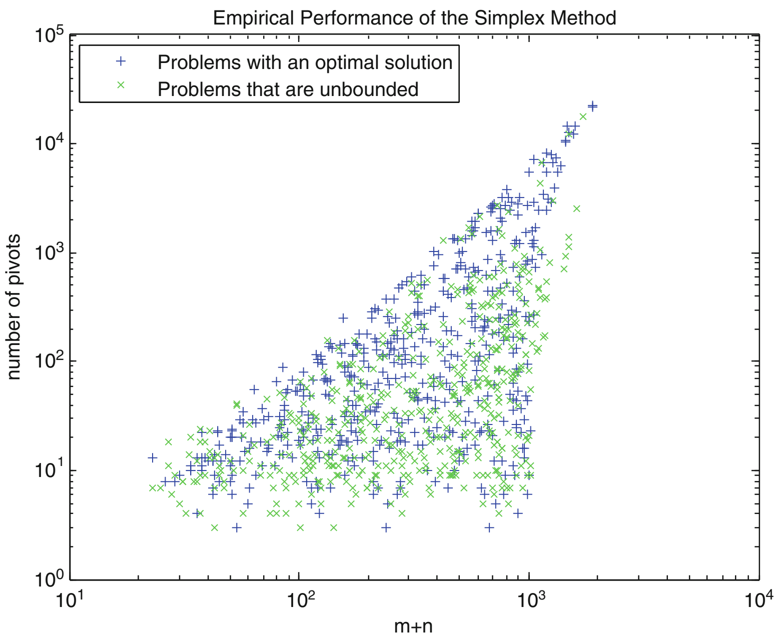

FIGURE 4.1. Starting from a primal feasible solution and using the primal simplex method to pivot to an optimal solution, shown here is a log-log plot showing the number of pivots required to reach optimality (or discover that the problem is unbounded) plotted against $m + n$. Points plotted in *blue* correspond to problems having an optimal solution whereas points plotted in *green* correspond to unbounded problems.

The code was run 1,000 times and for each randomly generated problem the values of `m`, `n`, and the number of iterations, `iter`, were saved. Figure 4.1 shows a plot of the number of pivots plotted against the sum `m+n`. Note that this is a log-log plot. That is, both the horizontal and vertical axes are stretched logarithmically.

The data points shown in blue correspond to problems where an optimal solution was obtained whereas those shown in green correspond to unbounded problems. Clearly, unboundedness is a common occurance for problems generated randomly. In fact, of the 1,000 problems, 501 had optimal solutions and the remaining 499 were unbounded. These numbers suggest that the probability of encountering an unbounded problem is exactly one half. This is likely true but it has not been proven. Further investigation reveals that instances where $m > n$ are almost never unbounded whereas the preponderance of $m < n$ instances are unbounded. Of course, the way m and n were generated, it is true that $m < n$ and $m > n$ are equally likely.

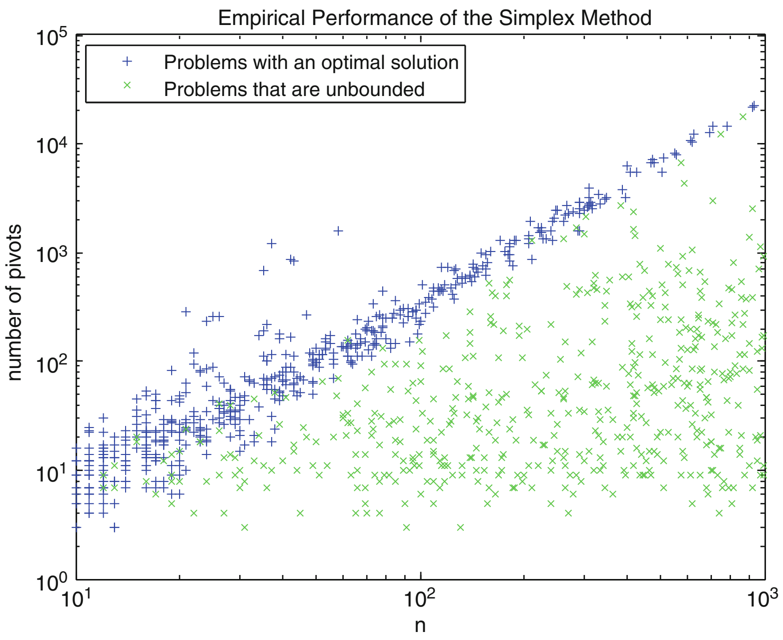

FIGURE 4.2. The same data as shown in Figure 4.1 but plotted against n instead of $m + n$.

One might be tempted to add code that rounds to zero any right-hand side value that is within a small tolerance of zero. For example, we could add

```
    b = b.*(abs(b)>eps);
```

just before the line defining `row`. Doing so forces dictionaries to be degenerate and this can lead to cycling. Experiments show that, with this extra line of code, about 5 out of 1,000 instances will cycle due to degeneracy. Without the extra line of code, no instances cycled.

A second observation is that, for a given value of $m + n$, there seems to be an effective upper limit on the number of pivots required. Some problems, especially the unbounded ones but also many having an optimal solution, solve in many fewer iterations.

The fact that some problems solved quickly even when $m+n$ was large suggests that perhaps $m + n$ is not the best measure of problem size. Figure 4.2 shows the same data plotted using just n as the measure of problem size. Interestingly, this change dramatically improves the correlation between size and number of iterations for those problems that arrived at an optimal solution. But, the unbounded problems are still spread out quite a bit.

Upon reflection, it seems that perhaps a problem is "easy" if either m or n is small relative to the other. To test this idea, we plot the number of iterations against the minimum of m and n. This plot is shown in Figure 4.3. It appears like we

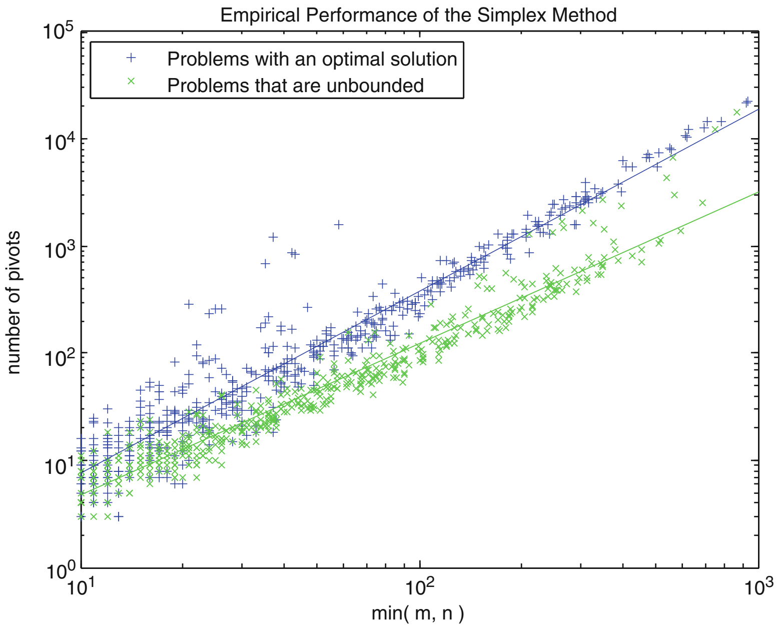

FIGURE 4.3. The same data as before but plotted against the minimum of m and n.

have finally arrived at the best measure of size. Performing a statistical analysis (see Chapter 12), we can empirically derive the straight line through this log-log plot that best matches the data. Separate lines for the blue (optimal) and green (unbounded) points are shown on the graph. For the blue points, the equation is given by

$$\log T \approx -1.90 + 1.70 \log(\min(m, n))$$

where T denotes the number of pivots required to solve a problem. Taking the exponential of both sides, we get

$$T \approx e^{-1.90} e^{1.70 \log(\min(m,n))} = 0.150 \min(m, n)^{1.70}.$$

For the green (unbounded) points, the equation is

$$T \approx 0.180 \min(m, n)^{1.42}.$$

In both cases, the rate of growth of T with respect to $\min(m, n)$ is “superlinear” as the exponents are both larger than 1. Figure 4.4 is a regular, that is not log-log, plot of the number of simplex pivots versus the minimum of m and n. This plot makes the superlinearity easy to spot.

Finally, a careful comparison of the blue (optimal solution) data points in Figures 4.2 and 4.3 reveals that they are almost all in exactly the same position in the two plots. The reason is that the vast majority of the problems that had an optimal solution were problems in which n was smaller than m.

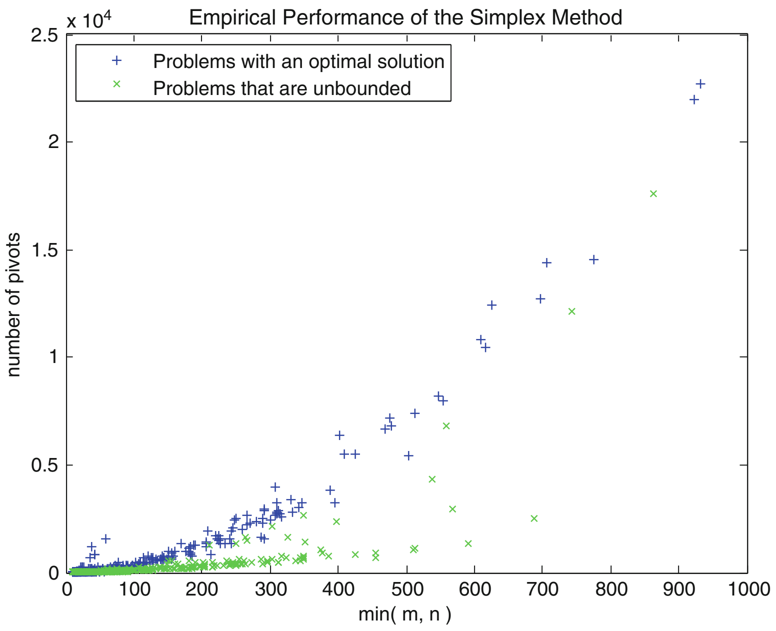

FIGURE 4.4. The same comparison as in Figure 4.3 but plot linearly rather than log-log. This version makes clear that the number of pivots grows faster than linearly.

Exercises

In solving the following problems, the simple pivot tool can be used to check your arithmetic:

www.princeton.edu/∼rvdb/JAVA/pivot/simple.html

4.1 Compare the performance of the largest-coefficient and the smallest-index pivoting rules on the following linear program:

$$\begin{array}{lrcl} \text{maximize} & 4x_1 + 5x_2 & & \\ \text{subject to} & 2x_1 + 2x_2 & \leq & 9 \\ & x_1 \qquad & \leq & 4 \\ & x_2 & \leq & 3 \\ & x_1,\ x_2 & \geq & 0\,. \end{array}$$

4.2 Compare the performance of the largest-coefficient and the smallest-index pivoting rules on the following linear program:

$$\begin{array}{lrcl} \text{maximize} & 2x_1 + x_2 & & \\ \text{subject to} & 3x_1 + x_2 & \leq & 3 \\ & x_1,\ x_2 & \geq & 0\,. \end{array}$$

4.3 Compare the performance of the largest-coefficient and the smallest-index pivoting rules on the following linear program:

$$\begin{array}{ll} \text{maximize} & 3x_1 + 5x_2 \\ \text{subject to} & x_1 + 2x_2 \le 5 \\ & x_1 \qquad\;\; \le 3 \\ & \qquad\;\; x_2 \le 2 \\ & x_1,\; x_2 \ge 0 . \end{array}$$

4.4 Solve the Klee–Minty problem (4.1) for $n = 3$.

4.5 Solve the four variable Klee-Minty problem using the online pivot tool:

www.princeton.edu/∼rvdb/JAVA/pivot/kleeminty.html

4.6 Consider the dictionary

$$\begin{aligned} \zeta &= -\sum_{j=1}^{n} \epsilon_j 10^{n-j} \left(\frac{1}{2} b_j - x_j \right) \\ w_i &= \sum_{j=1}^{i-1} \epsilon_i \epsilon_j 10^{i-j} (b_j - 2x_j) + (b_i - x_i) \qquad i = 1, 2, \ldots, n, \end{aligned}$$

where the b_i's are as in the Klee–Minty problem (4.2) and where each ϵ_i is ± 1. Fix k and consider the pivot in which x_k enters the basis and w_k leaves the basis. Show that the resulting dictionary is of the same form as before. How are the new ϵ_i's related to the old ϵ_i's?

4.7 Use the result of the previous problem to show that the Klee–Minty problem (4.2) requires $2^n - 1$ iterations.

4.8 Consider the Klee–Minty problem (4.2). Suppose that $b_i = \beta^{i-1}$ for some $\beta > 1$. Find the greatest lower bound on the set of β's for which the this problem requires $2^n - 1$ iterations.

4.9 Show that, for any integer n,

$$\frac{1}{2n} 2^{2n} \le \binom{2n}{n} \le 2^{2n}.$$

4.10 Consider a linear programming problem that has an optimal dictionary in which exactly k of the original slack variables are nonbasic. Show that by ignoring feasibility preservation of intermediate dictionaries this dictionary can be arrived at in exactly k pivots. Don't forget to allow for the fact that some pivot elements might be zero. *Hint: see Exercise 2.15.*

4.11 (MATLAB required.) Modify the MATLAB code posted at

www.princeton.edu/∼rvdb/LPbook/complexity/primalsimplex.m

so that data elements in A, b, and c are not rounded off to integers. Run the code and compare the results to those shown in Figure 4.3.

4.12 (MATLAB required.) Modify the MATLAB code posted at

www.princeton.edu/∼rvdb/LPbook/complexity/primalsimplex.m

so that the output is a log-log plot of the number of pivots versus the product m times n. Run the code and compare the results to those shown in Figure 4.3.

Notes

The first example of a linear programming problem in n variables and n constraints taking $2^n - 1$ iterations to solve was published by Klee and Minty (1972). Several researchers, including Smale (1983), Borgwardt (1982, 1987a), Adler and Megiddo (1985), and Todd (1986), have studied the average number of iterations. For a survey of probabilistic methods, the reader should consult Borgwardt (1987b).

Roughly speaking, a class of problems is said to have *polynomial complexity* if there is a polynomial p for which every problem of "size" n in the class can be solved by some algorithm in at most $p(n)$ operations. For many years it was unknown whether linear programming had polynomial complexity. The Klee–Minty examples show that, if linear programming is polynomial, then the simplex method is not the algorithm that gives the polynomial bound, since 2^n is not dominated by any polynomial. In 1979, Khachian gave a new algorithm for linear programming, called the *ellipsoid method*, which *is* polynomial and therefore established once and for all that linear programming has polynomial complexity. The collection of all problem classes having polynomial complexity is usually denoted by $\mathcal{P}$. A class of problems is said to belong to the class $\mathcal{NP}$ if, given a (proposed) solution, one can verify its optimality in a number of operations that is bounded by some polynomial in the "size" of the problem. Clearly, $\mathcal{P} \subset \mathcal{NP}$ (since, if we can solve from scratch in a polynomial amount of time, surely we can verify optimality at least that fast). An important problem in theoretical computer science is to determine whether or not $\mathcal{P}$ is a strict subset of $\mathcal{NP}$.

The study of how difficult it is to solve a class of problems is called *complexity theory*. Readers interested in pursuing this subject further should consult Garey and Johnson (1977).