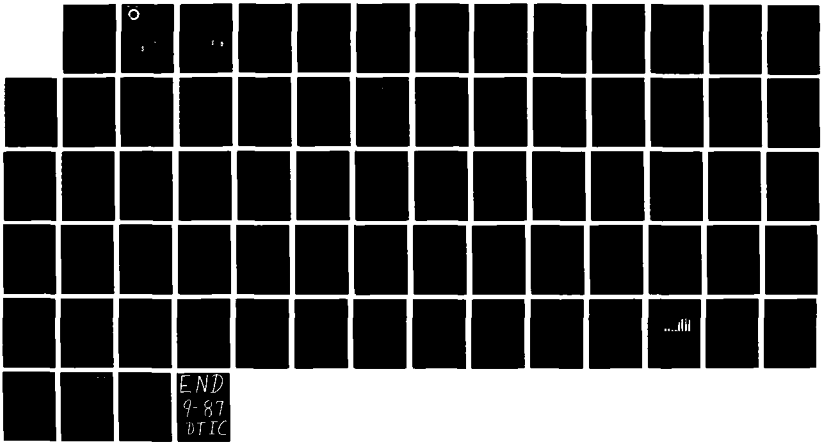
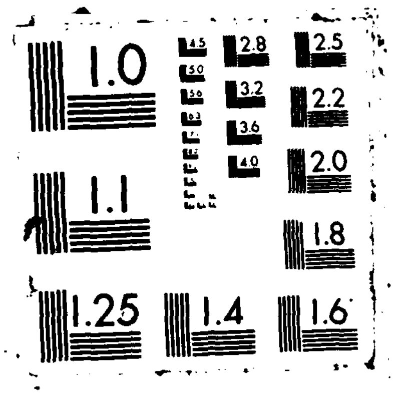

END
9-87
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

Systems Optimization Laboratory

(12)

COMPARISIONS OF COMPOSITE SIMPLEX ALGORITHMS

by
Irvin J. Lustig

TECHNICAL REPORT SOL 87-8

June 1987

Department of Operations Research
Stanford University
Stanford, CA 94305

SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF OPERATIONS RESEARCH
STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305-4022

COMPARISIONS OF COMPOSITE SIMPLEX ALGORITHMS

by
Irvin J. Lustig

TECHNICAL REPORT SOL 87-8

June 1987

Research and reproduction of this report were partially supported by the National Science Foundation Grants DMS-8420623, ECS-8312142, and SES-8518662; U.S. Department of Energy Grant DE-FG03-87-ER25028; and the Office of Naval Research Contract N00014-85-K-0343.

Comparisons of Composite Simplex Algorithms

Irvin J. Lustig
Department of Operations Research
Stanford University

ABSTRACT

For almost forty years, the simplex method has been the method of choice for solving linear programs. The method consists of first finding a feasible solution to the problem (Phase I), followed by finding the optimum (Phase II). Many algorithms have been proposed which try to combine the processes embedded in the two-phase process. This study will compare the merits of some of these composite algorithms.

Theoretical and computational aspects of the Weighted Objective, Self-Dual Parametric, and Markowitz Criteria algorithms are presented. Different variants of the Self-Dual methods are discussed.

A large amount of computational experience for each algorithm is presented. These results are used to compare the algorithms in various ways. The implementations of each algorithm are also discussed. One theme that is present throughout all of the computational experience is that there is no one algorithm which is the best algorithm for all problems.

Table of Contents

List of Tables and Figures

Section 1. Introduction

Since the simplex method was discovered by Dantzig in 1947, it has remained the method of choice for solving linear programs. It is used every day by people from many different fields and in many different applications. While other methods have been proposed in the past for solving linear programs, none were shown to perform consistently as well as the simplex method (see, for example, Hoffman, et.al., 1953). Recently, however, new methods have been proposed based on Karmarkar's (1984) discovery of a new polynomial-time algorithm for linear programming, and they are posing serious challenge. In comparing the performance of these algorithms with the simplex method, there is an implicit assumption that the simplex method is an explicit algorithm, whereas, in practice, there are many variants. The purpose of this study is to compare the variants to see if any one variant can be said to be the best in some sense and could be used as a standard for comparison with non-simplex algorithms.

While many variants of the simplex method have been proposed, systematic studies of their effectiveness have not appeared in the scientific literature. This study is limited to a certain class of variants of the simplex method, called *composite simplex algorithms*. This report contains computational comparisons of some composite simplex algorithms, and emphasis is placed on making the computational experiments replicable by other researchers.

Section 2 defines the class of composite simplex algorithms and describes a generic simplex algorithm and a framework whereby all the composite simplex algorithms are implemented. Section 3 discusses the various options that are available when one solves a linear program on a computer. The influence of these options must be understood when making computational comparisons of variants of the simplex method. The results for the standard two-phase algorithm are presented here.

Section 4 discusses the weighted objective algorithm. Section 5 analyzes the Markowitz Criterion. In Section 6, Dantzig's self-dual parametric algorithm is presented. Some variants of the self-dual algorithm are presented in Section 7. In Section 8, the computational results are analyzed and suggestions for further research are offered.

The following general inferences can be drawn from the very extensive systematic trials made in this study:

1. Almost every variant performed remarkably well on some problems and very poorly on others.
2. No variant stood out clearly as the best on all problems.

Section 2. The Linear Programming Problem

In this section, the general linear programming problem is discussed. A "generic" simplex method will be described in a way that serves as a framework for describing various variants. The mathematical programming system MINOS (Murtagh and Saunders, 1983), suitably modified, was used in the experiments on test examples in this study. A description of how MINOS solves linear programs is therefore given. Finally, the "MinSumInf" ratio test is discussed as an alternative ratio test which can be used in some methods.

2.1 The MINOS Standard Form

A linear program can be described as the minimization or maximization of a linear form $z = c^Tx$ over a linear constraint set. Each of the constraints in this set may be an equality or an inequality constraint. After a linear program is formulated mathematically and numerical values assigned to its coefficients and right-hand side, these values are placed in an input file in a prescribed format in order for a computer to solve it. Once the data is read into the computer, the next step is to translate it into an equivalent form for internal use by the computer program. An algorithm is then used to solve the linear program. In the case of MINOS, the form

$$\begin{aligned} \text{minimize} \quad & c^Tx \\ \text{subject to} \quad & [A\ I]x = 0, \\ & l \le x \le u. \end{aligned} \tag{2.1}$$

is termed the *The MINOS Standard Form.*

To maximize z, the user can minimize $-z$ and obtain the same optimal solution x. Henceforth, it is not restrictive to assume that the objective function is being minimized. The number of variables is denoted by n, and the solution is then given by some $x \in \Re^n$. Each variable x_j, $j = 1, \ldots, n$ is termed a *decision variable* and can have a lower bound l_j and an upper bound u_j. Some of these bounds may be $-\infty$ and/or $+\infty$, respectively.

The number of *constraints in the problem is denoted by* m. Each constraint is described by a vector $a_i \in \Re^n$ and two scalars, b_i and r_i. In the MINOS manual (Murtagh and Saunders, 1983), b_i is termed the *right-hand side* and r_i is termed the *range*. The i^{th} constraint is then

$$\begin{aligned} b_i + r_i \le a_i^Tx \le b_i & \quad \text{if} \quad r_i < 0, \\ b_i \le a_i^Tx \le b_i + r_i & \quad \text{if} \quad r_i > 0. \end{aligned}$$

Note that if $r_i = 0$, the constraint is of the form $a_i^Tx = b_i$; if $r_i = +\infty$, the constraint is of the form $a_i^Tx \ge b_i$; if $r_i = -\infty$, the constraint is of the form $a_i^Tx \le b_i$. Given these m constraints, m *slack variables*, x_j, $j = n+1, \ldots, n+m$, can be added to transform each of the constraints into equality constraints. This is done by setting $x_{n+i} = -a_i^Tx$, $i = 1, \ldots, m$, and placing lower and upper bounds on x_{n+i} that were previously on a_i^Tx. Hence the linear program has been converted to the form (2.1). Now $x \in \Re^{n+m}$ and there is no need to treat differently the slack variables and the decision variables of the problem, since each x_j is considered as a column of the large matrix $[A\ I]$ with a cost coefficient c_j and lower and upper bounds l_j and u_j.

2.2 A Generic Simplex Method

The simplex algorithm as described by Dantzig (1963) (and in earlier references) is applicable to linear programs that have an initial primal feasible basis. In order to find an initial basis, a "Phase I" procedure was devised. This procedure sets up a linear program which could be solved by the simplex algorithm, and which yields a primal feasible basis or a verification that the problem is infeasible. If the problem was feasible, the simplex algorithm could be used again to find an optimal solution ("Phase II"). The combination of first finding a primal feasible basis by the simplex algorithm and then finding the optimal solution by the same algorithm is collectively known as the simplex *method*. This fine distinction is made here to clarify the differences between the simplex algorithm and the simplex method.

Since 1947 when Dantzig proposed the simplex algorithm, many variants of the method and algorithm have been proposed. Some of these variants are listed in Dantzig's book (1963). These variants have a common framework, which can be described by the following proto-algorithm:

```
Comment Let $B$ be a basis for $[A\ I]$.
while not $optimal(B)$ do
    begin
    $\{s, q\} \leftarrow$ indices of $\{incoming\_column, exiting\_column\}$;
    Comment Column $q$ of $[A\ I]$ is the same as column $r$ of $B$.
    $pivot(B, s, q, r)$;
    end
```

The boolean function *optimal*(B) returns **true** if B is both a primal and dual feasible basis. The procedure *pivot*(B, s, q, r) assumes that x_q is the r^{th} basic variable. Column s of $[A\ I]$ replaces the r^{th} column in the basis, which is the same as column q of $[A\ I]$. This procedure usually updates a factorization of the basis B. The order of execution of the functions *incoming_column* and *exiting_column* determines which simplex variant is being executed. In the case of the primal simplex algorithm, the steps (called *pricing out*) for deciding which column is to be the *incoming_column* are executed before the steps (a *primal ratio test*) for deciding which is to be the *exiting_column*. In the case of the dual simplex algorithm, finding the *exiting_column* (determining the basic variable with the most infeasibility) is done before finding the *incoming_column* (a dual ratio test). As will be seen later, some algorithms may change the order of the two processes depending on the status of the current iteration. Furthermore, the actual algorithms that define the functions *incoming_column* and *exiting_column* may change as iterations progress.

2.3 An Outline of MINOS

In order to understand the implementations of the various algorithms studied in this study, it is necessary to understand the subroutines that MINOS uses to solve linear programs. For ease of discussion, the names that were given to subroutines in MINOS have been changed to a mnemonic that reminds the reader of the subroutine's purpose. After reading in the problem specification and converting the problem to MINOS standard form (subroutine *input*), a starting basis is determined. A basis is easily obtained from the matrix $[A\ I]$ by choosing the slack variables. As discussed in Section 3, it is usually more efficient to determine an initial "crash" basis to begin the simplex method (subroutine *crash*). Each nonbasic variable is initially set to be equal to one of its bounds. $\mathcal{B}$ denotes the index set $\{j_1, j_2, \ldots, j_m\}$ of the basis. The order of the basic columns is important and is indicated by an integer vector $\mathcal{B}_i$, $i = 1, \ldots, m$ where $\mathcal{B}_i = j$ indicates that x_j is the i^{th} basic variable. Given the initial basis and the initial states of the nonbasic variables, MINOS executes the following loop until termination:

```
procedure primal_simplex;
    finished ← false;
    while not finished do
        begin
        if needs_factorization(B) then factorize(B);
        if infeasible_basis(B) then $\hat{c} \leftarrow d$ else $\hat{c} \leftarrow c$;
        $\boxed{\textbf{FindPI}\text{: Solve } \pi^T B = \hat{c}_B \text{ for } \pi;}$
        $\boxed{\textbf{PriceOut}\text{: Find } s \text{ such that } x_s \text{ has minimum reduced cost } \bar{c}_s;}$
        optimal ← $\bar{c}_s \geq 0$;
        if not optimal then
            $\boxed{\textbf{Represent}\text{: Solve } Bp = [A\ I]_{\cdot s} \text{ for } p;}$
            $\boxed{\textbf{RatioTest}\text{: Find } r \text{ such that } x_{B_r} \text{ blocks the change in } x_s;}$
            unbounded ← $r = 0$
            if not unbounded then
                begin
                $q \leftarrow B_r$;
                $\boxed{\textbf{Update}\text{: Update value of } x;}$
                pivot(B, s, q, r);
                end;
            end;
        infeasible ← infeasible_basis(B) and optimal;
        finished ← optimal or unbounded or infeasible;
        end;
```

The boolean function *needs_factorization*(B) is **true** if the basis needs to be refactorized. A number of circumstances can contribute to this condition. MINOS has a `FACTORIZE FREQUENCY` option, which determines the maximum number of updates to B that can be done before refactorizing the basis. This number defaults to 50. MINOS will also refactorize the basis if the size of the current factorization has increased significantly, or if there is insufficient storage to update the factors. Because of the modular nature of the MINOS code, the conditions influencing refactorization are external to the variants of the simplex method that are being considered. Furthermore, the subroutine *factorize*(B) and the routines that solve the equations $\pi^T B = c$ and $Bx = b$ can be treated as a package; the results of using this package do not influence the paths generated by the different variants of the simplex method. MINOS uses a sparse LU factorization of the basis. The implementation is sufficiently modularized that the factorization and solve routines can be replaced by some other equivalent scheme of maintaining the factorization of the basis, if so desired.

The boolean function *infeasible_basis*(B) is **true** if the current basic solution indicated by B is primal infeasible. The vector $d \in \Re^{n+m}$ is defined element by element as

$$d_j = \begin{cases} 1 & \text{if } x_j > u_j; \\ -1 & \text{if } x_j < l_j; \\ 0 & \text{otherwise} \end{cases}$$

and represents the Phase I cost form used when the problem is infeasible. Note that if x_j is nonbasic, then $d_j = 0$, since each nonbasic variable is always equal to one of its bounds.

The Phase I objective vector d changes on each iteration, depending on which variables are infeasible. Once a variable is feasible, the ratio test prevents the variable from becoming infeasible. The objective in Phase I never increases since MINOS chooses incoming variables based on the cost form $w = d^T x$. However, because the sum of infeasibility changes as a function of the incoming variable, it is possible for this sum to increase. MINOS always insures that if the sum does increase, the number of infeasibilities will decrease, which is enough to prove convergence.

After determining the objective to minimize and solving for prices π, MINOS chooses an incoming variable x_s by computing the reduced costs of the nonbasic variables and picking the variable with the minimum reduced cost. The reduced cost $\bar{c}_j$ is defined for x_j nonbasic as

$$\bar{c}_j = \begin{cases} \hat{c}_j - \pi^T[A\ I]_{\cdot j} & \text{if } x_j = l_j; \\ -(\hat{c}_j - \pi^T[A\ I]_{\cdot j}) & \text{if } x_j = u_j. \end{cases}$$

It should be noted that MINOS does not save the values of the reduced costs as they are computed, since these values are only needed to determine the index s of the incoming variable. This facilitates the use of partial pricing schemes, which are discussed in Section 3.

The next step in the simplex method is a ratio test. In order to do this, a direction vector for modifying the current solution x is determined by solving the equation $Bp = [A\ I]_{\cdot s}$. The vector p is the representation of the incoming column in terms of the basis B. The components of x except for x_B and x_s are not changed. The dimension of the vector of basic variables x_B is expanded by one to include x_s, so that $\hat{x} = (x_B, x_s)$. The direction of change of $\hat{x}$ is $\hat{p} = (p, t)$, where $t = 1$ if $x_s = u_s$ and $t = -1$ if $x_s = l_s$. To make the ratio test easier to implement, let $\mathcal{B}_{m+1} = s$. The ratio test determines the maximum θ such that $\hat{x} - \theta\hat{p}$ is feasible. Specifically,

$$\theta = \min_{i=1,\ldots,m+1} \left\{ \begin{array}{ll} \dfrac{\hat{x}_i - l_{B_i}}{\hat{p}_i} & \text{if } \hat{p}_i > 0; \\ \dfrac{u_{B_i} - \hat{x}_i}{-\hat{p}_i} & \text{if } \hat{p}_i < 0; \end{array} \right\}.$$

Given θ, the value of the basic variables can be changed using the formula $x_B \leftarrow x_B - \theta\hat{p}$. If $\hat{x}_r$ is the blocking variable, then $\mathcal{B}_r$ and $\mathcal{B}_{m+1}$ are interchanged as well as $\hat{x}_r$ and $\hat{x}_{B_{m+1}}$. If $r = m + 1$, then the variable x_s has moved from one of its bounds to the opposite bound, and no update of the basis is necessary.

2.4 The "MinSumInf" Ratio Test

The standard ratio test described above has computational complexity $O(m)$. As an alternative to this ratio test in Phase I, Greenberg (1978) described a primal ratio test which has come to be known as the "Dennis Rarick" ratio test, since he implemented it in WHIZARD (see MPS-III (1975)). Wolfe (1961) described this ratio test as an "extended composite algorithm," but his use of the word "composite" has a different meaning than that defined in the next section. Because of the lack of suitable references, it is not clear from the literature whether Rarick or Greenberg were aware of Wolfe's earlier work.

It is easy to prove that the function measuring the sum of the infeasibilities of the current basic variables over the range of the incoming basic variable x_s is convex. The key idea behind the ratio test is to choose the outgoing variable associated with the value of x_s that minimizes this function. Without loss of generality, we may assume (by translating the variables and reversing sign, if necessary) that all the variables have lower bounds $l = 0$ and upper bounds $u = +\infty$. Unrestricted variables with infinite lower and upper bounds are ignored. After selecting the incoming variable x_s and finding the representation p of the incoming column, the value of x_s is increased from 0 to some value θ while the values of the basic variables x_{B_i}, $i = 1, \ldots, m$ are changed according to the formula $x_{B_i} \leftarrow x_{B_i} - \theta p$.

To simplify the notation, let $\beta_i = x_{B_i}$, and $\alpha_i = p_i$. For $\theta \geq 0$, the function which computes the sum of the infeasibility of the basic variables can be written as

$$S(\theta) = -\sum_{i=1}^{m} \min(0, \beta_i - \theta\alpha_i).$$

As a function of θ, this function is piecewise linear and convex. An example is shown in Figure 2.1. The function has breakpoints $t_i = \beta_i/\alpha_i$ where the slope $w_i(\theta) = S_i'(\theta)$ changes. The "MinSumInf" ratio test searches for the ratio t_i that minimizes $S(\theta)$. Let w_i^- be the left derivative and w_i^+ be the right derivative at t_i. Note that $w_i^+ = w_{i+1}^-$ for $i = 1, \ldots, m-1$. Since $S(\theta)$ is convex, $w_i^- \leq w_i^+$. The minimum of the function occurs at $\theta = t_i$ where $w_i^- \leq 0$ and $w_i^+ \geq 0$. If both t_i and t_{i+1} tie for the minimum, t_i is selected.

Greenberg's algorithm to minimize $S(\theta)$ involves *sorting* the sequence t_i. Under a simple computational model, it can be shown that at least $O(n \log n)$ comparisons are needed to sort this sequence (Aho, Hopcroft, and Ullman, 1974). However, the methodology of Megiddo (1983), who found a linear-time algorithm to solve linear programs when $n = 2$, can be used in the "MinSumInf" ratio test in order to develop a linear-time algorithm to find the t_i that minimizes $S(\theta)$. The latter, however is difficult to implement, and is only applicable to a two-phase method. For these reasons, this ratio test was not considered in this study.

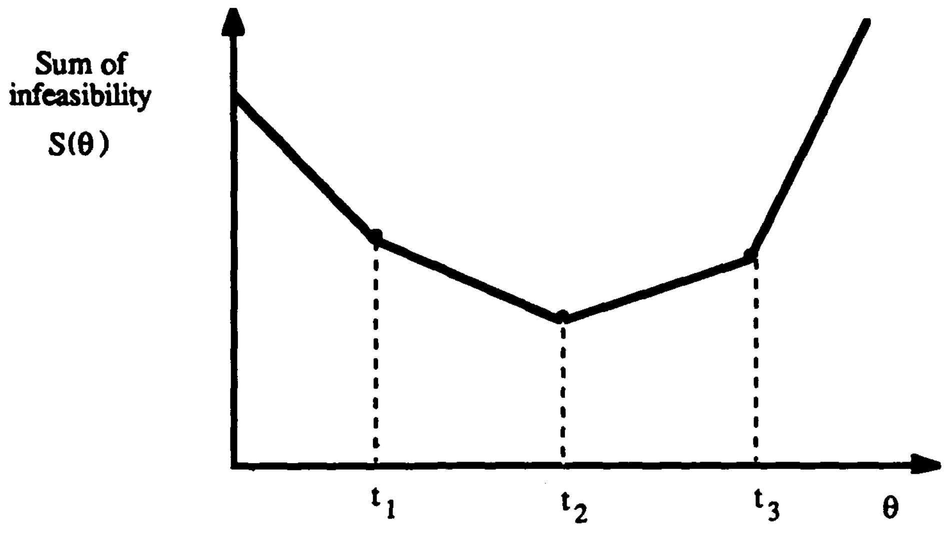


Figure 2.1 The function $S(\theta)$

2.5 Composite Simplex Algorithms

In most mathematical programming systems today, the simplex method is used with the Phase I/Phase II procedure described earlier. Ordinarily, the Phase I linear program has multiple optimal solutions, since every feasible solution for the original linear program is optimal in Phase I. This fact would lead one to believe that the Phase II procedure could begin at a vertex very far away from the optimal vertex. This was observed early on by many who worked on the development of the simplex method; these investigators invented numerous *composite* simplex algorithms which combine in various ways the reduction of primal and dual infeasibility, until zero is achieved for both. Essentially, these composite algorithms try to use information about the objective function $c^T x$ while finding an initial feasible point.

Finding an optimal solution of the primal linear program involves among other things finding a feasible solution to the dual linear program. As most of the methods progress, the primal and dual systems play a game of "Tug-of-War." At any primal infeasible point, it is impossible to know whether the current objective value is below (superoptimal) or above (suboptimal) its optimal value. If the point is suboptimal, an improvement can often be made which will simultaneously reduce the amount of infeasibility and decrease the value of an objective function (perhaps going below the optimal value of the objective). At a superoptimal point, a reduction in the amount of infeasibility often can only be made by increasing the value of the objective function. However, some superoptimal points have adjacent vertices which can reduce infeasibility and improve the value of the objective function. At such a point, movement to such an adjacent vertex looks promising, but one is deceived by the local improvement criterion of the simplex method. In this case, the adjacent vertex is improving the value of the primal objective function while increasing the amount of dual infeasibility. It is in this sense that we say that the primal and dual feasibility conditions are "tugging" at each other. The various composite algorithms attempt to act as the referee between trying to satisfy the competing primal and dual feasibility conditions.

Section 3. Computational Testing Methodology

In the course of making computational comparisons of various variants of the simplex method, it is important to recognize that many factors can contribute to the overall performance. MINOS allows the user to select from a number of options, and the ones used must be carefully chosen. In this section, these options and their effects on the computational testing are discussed. In order to reduce the amount of variability in performance due to the options chosen, the options used were fixed in such a way as not to influence the performance of any specific algorithm at the expense of another competing algorithm. In this section, previous studies on computational testing of the simplex algorithm are also discussed.

3.1 Practical Experience with the Simplex Method

The question of just how efficient the simplex method is has existed since its proposal in 1947. Interestingly enough, in spite of the long interest in its efficiency, the amount of serious scientific testing of the algorithm appears to be quite small. Shamir (1986) provides an excellent summary of results that have appeared in the literature pertaining to the performance of the simplex method; the interested reader should refer to his work for references. Shamir states:

> "The experience accumulated during the last three decades on the behavior of the Simplex Method is undoubtedly vast. However, there is surprisingly little documented evidence on this experience in the scientific literature. One prosaic explanation may be that practitioners usually do not keep record of parameters of the solution process for the problems they solve (since they are interested mainly in the result), and if they do they seldom report on this record in the scientific literature."

With regard to many composite simplex algorithms that have been proposed, no systematic study seems to exist in the scientific literature of their efficiencies.

The earliest experiments with the simplex method were performed by Hoffman, Mannos, Sokolowsky, and Wigmann in 1953. At the time, they were comparing the simplex method to other non-simplex methods for solving linear programs; they found the simplex method to be superior. Dantzig has remarked that their work provided the impetus to perfect the simplex method. If their results had not been favorable, it is possible that other methods would have been perfected and the simplex method would not be in use today!

In 1963, Wolfe and Cutler experimented with nine "real" linear programming problems, which were submitted to them by practitioners of linear programming at the time. They compared different pivoting rules for the simplex method; their conclusion was that the original 'most negative reduced cost' rule, in spite of the fact that it is dependent on the scaling of the variable, did as well as the other alternatives. It may be one of the reasons why it is used in most codes today. The most-negative reduced cost rule has its origins in problems with a convexity constraint. In such problems, an argument can be made for choosing this rule. Why it appears to work well in practice without such a constraint is a mystery and may be due to practitioners choosing units for competing activities consistently (See Dantzig (1987) for further discussion). Since the Wolfe-Cutler study, no serious computational comparisons of simplex variants on *real-world problems* have appeared in the scientific literature.

In contrast to using "real" problems for testing, there has been much work done using randomly generated problems. In the early sixties, Kuhn and Quandt (1962) were the first to conduct a series of experiments comparing the performance of many different pivoting rules on randomly generated problems. Other authors followed with different experiments, usually changing some aspect of the probabilistic model. Because of the stochastic nature of these experiments, the statistics computed can give some insight on the average performance of different variants. But, as discussed later, it is not clear that the performance of the simplex method on such problems is representative of the performance of the simplex method on the variety of real-world problems that are solved every day by linear programming practitioners.

3.2 The Crash Basis

Like most linear programming systems, MINOS provides the option of finding a *crash* basis as the initial basis to start up the simplex method. The key idea is to rearrange the order of the rows and then select a lower triangular basis using columns from the original input matrix A. It should be noted that columns from the slack variables can always be used to keep the initial basis lower triangular. In fact, if A were totally dense, then only one column would be chosen from A, with the rest of the columns being chosen from the slack variable matrix I. MINOS also selects columns so that each diagonal element of the crash basis is reasonably large relative to the other elements in the same column, in order to prevent the initial basis from being ill-conditioned.

There seem to be some good reasons to begin with a crash basis, as opposed to the basis consisting entirely of slack variables. To understand these reasons, the concept of *Hamming distance* is useful. Let J_1 and J_2 be two sets of indices of basic variables. Then the Hamming distance is defined as

$$h(J_1, J_2) = |J_1 \cup J_2| - |J_1 \cap J_2|.$$

In other words, $h(J_1, J_2)$ is the minimal number of pivot steps (ignoring feasibility) required to move from the basis corresponding to J_1 to the basis corresponding to J_2. It provides a useful lower bound on how close a specific basis is to another basis, in terms of number of pivots of the simplex method.

In most linear programs, the number n of original columns is larger than the number m of rows. Hence, if all variables (decision and slack) have equal probability of being in a specific basis $\mathcal{B}$, the probability of the i^{th} basic variable in the basis $\mathcal{B}$ being a decision variable (as opposed to a slack variable) is $\frac{n}{m+n} > \frac{1}{2}$. Hence, the expected number of decision variables in the optimal basis should be larger than the expected number of slack variables in the optimal basis. Therefore, choosing a crash basis from the set of original decision variables should keep the initial Hamming distance between the crash basis and the optimal basis lower.

On the other hand, if we consider the probability of a certain variable x_j being in the optimal basis, with all bases being equally likely, a different analysis is required. Since there are $\binom{n+m}{m}$ possible bases, and $\binom{n+m-1}{m}$ bases which do not include x_j, the probability of x_j not being in a random basis is $\frac{n}{n+m}$. Hence, the likelihood of x_j being in a randomly chosen basis is $\frac{m}{n+m} < \frac{1}{2}$. So choosing a specific x_j to be in the crash basis may be disadvantageous.

However, many slack variables correspond to equality rows, and therefore cannot be in the optimal basis unless redundancies among the constraints exist, or in certain degenerate situations. Furthermore, in formulating linear programs, there is a tendency to write inequality constraints, a good proportion of which are expected to be tight at the optimal solution so that their corresponding slack variable will most likely not be in the optimal basis. On the other hand, if the problem has many constraints which are not binding in the optimal solution, then the optimal basis may contain more slack variables than decision variables. In most problems, the former seems to be the case. With these considerations, all the experiments in this dissertation were done using the CRASH BASIS option of MINOS to initiate the various algorithms.

3.3 Scaling

MINOS offers a SCALE option when solving a linear program. This option will take the original input of the problem and attempt to make the coefficients of each column and row as close to 1 as possible. This is done by choosing two nonsingular diagonal matrices R and S to respectively multiply the rows and columns of the problem. The values of the elements of A, c, l, and u are modified. The algorithm used in MINOS is described by Fourer (1982). The most immediate advantage of using the SCALE option is in controlling any numerical difficulties that may occur in a specific problem. Scaling is theoretically attractive because it makes certain selection rules scale free. Appealing as it is on theoretical grounds to be scale free, nevertheless, it is not clear how scaling affects the performance of the simplex algorithm or its variants. This is apparent from the computational results in this study.

Badly scaled linear programs seem to be an evil that those who develop software to solve practical problems must contend with. Quite often practical models contain constraints that implicitly must convert the units used in one constraint to the units used in another. For example, if x_1 is a variable whose unit is millions of dollars, and x_2 is a variable whose unit is dollars, then a constraint that adds x_1 and x_2 would be similar to

$$x_1 + 10^6 x_2 \leq 10.$$

It is clear that such constraints can cause bad problems with regard to numerical accuracy. Scaling often helps remove some of these effects.

It is easy to see that scaling a problem can change the path of the simplex algorithm. If a column j is multiplied by ρ, then the reduced cost of that column will be $\rho\bar{c}_j$. Therefore, the choice of incoming column used by the 'most negative reduced cost rule' could change if the units of a variable were changed by the user at the time of formulating his linear program.

The scaling algorithm is executed before a crash basis is determined. Scaling, of course, does not affect the sparsity pattern of a linear program. Because the numerical values in the constraint matrix A influence the choice of columns for the crash basis, scaling can change the starting basis and therefore comparing the results of solving with the SCALE YES option with the results of solving with the SCALE NO option will not isolate the effects of scaling alone. If the two crash bases generated are labeled as the "unscaled basis" and the "scaled basis," these bases can be used as either the starting basis for either the scaled or unscaled, problem, respectively. Hence, for a specific problem and a specific algorithm, four variations can be considered, and the experience gained from running each algorithm under all of the variations can be used to compare the effects of scaling by removing any variability in the different starting bases. For notational convenience, the unscaled crash basis will be notated as UB and the scaled crash basis will be notated as SB. In all problems except AFIRO, these bases were different.

In order to generate the scaled basis for the unscaled problem and the unscaled basis for the scaled problem, the two initial bases for each problem were saved using the PUNCH option of MINOS. When each problem was solved, the scale option was set in the specifications file, and the appropriate crash basis was read in using the INSERT option. When the UB basis was used with the unscaled problem, and the SB basis with the scaled problem, the bases were not read in from their files, but were generated by the code at runtime. As explained later, the extra CPU time for these extra computations was recorded and adjustments made to correct the running time statistics generated by the experiments.

3.4 Tolerances

If a linear program is solved using rational arithmetic, the solution will satisfy the equations $[A\ I]x = 0$ and $l \leq x \leq u$ exactly. Furthermore, the conditions for optimality of the nonbasic variables $\bar{c}_j \geq 0$ will also hold. However, MINOS uses floating-point arithmetic, and it is not practical to obtain exact solutions to the primal and dual systems. It is necessary, therefore, to specify tolerances for infeasibility of the problem.

MINOS provides a `FEASIBILITY TOLERANCE` option. If t_f is this tolerance, then a solution x is said to be *feasible* if

$$l_j - t_f \leq x_j \leq u_j + t_f$$

for each $j = 1, \ldots, m+n$. Since these equations include the slack variables, the linear constraints are also satisfied to this tolerance t_f. For MINOS, the default feasibility tolerance is $t_f = 10^{-6}$. For all of the experiments done, the default feasibility tolerance was used.

A second tolerance is the `OPTIMALITY TOLERANCE`. This is related to the dual feasibility of the problem. If t_d is this tolerance, then the solution is declared optimal if

$$\frac{\bar{c}_j}{\|\pi\|} \geq -t_d.$$

(To avoid division by zero, the value in the denominator is actually $\max(1, \frac{\|\pi\|}{\sqrt{m}})$.) Dividing by the norm of π makes the test independent of any scaling of the objective function. The default value of the optimality tolerance is $t_d = 10^{-6}$. This value was used in all the computational tests.

The third tolerance that is important when solving linear programs is the `PIVOT TOLERANCE`. This tolerance is used to prevent columns from entering the basis if they would cause the basis to become almost singular. If this tolerance is denoted t_p, then a row i in the ratio test will be rejected if

$$|p_i| \leq t_p.$$

The default pivot tolerance for MINOS is $t_p = \epsilon^{2/3}$ and was used throughout all the testing. Here ϵ is the measure of machine precision; t_p was approximately 10^{-11} on the machine used in the experiments.

In order to maintain the "equality" of the linear constraints $[A\ I]x = 0$, a numerical test is done on every k^{th} iteration, where k is the `CHECK FREQUENCY`. If the largest component of the residual vector $r = [A\ I]x$ is determined to be too large (as determined by t_f and $\|x\|$; a larger error is allowed if $\|x\|$ is large), then the basis is refactorized and the basic variables are recomputed by a special algorithm to satisfy the linear constraints more accurately. The default value of $k = 30$ was used in all tests in this dissertation. In each test, the check never caused a refactorization.

3.5 Pricing

In determining a variable x_s to enter the basis, the simplex algorithm chooses s such that $\bar{c}_s$ is minimal. Since $\bar{c}$ is not needed elsewhere in the execution of the algorithm, it is efficient to compute $\bar{c}_j$ for each nonbasic x_j for $j = 1, \ldots, n+m$, while keeping track of the minimum $\bar{c}_j$ found so far. This combined procedure is known as *pricing out* the nonbasic variables. The fundamental calculation is to compute

$$g_j = c_j - \pi^T [A \ I]_{\cdot j}$$

for each column. If $x_j = l_j$, then $\bar{c}_j = g_j$; otherwise, $x_j = u_j$ and $\bar{c}_j = -g_j$. If $\bar{c}_j \geq 0$ for all j, then the current solution to the linear program is optimal.

The expense of pricing a specific column is directly related to the density of that column. If x_j is a slack variable, then $g_j = -\pi_j$. If the j^{th} column in A has k_j nonzero elements, then only k_j multiplications are needed to compute g_j. To actually compute s such that $\bar{c}_s$ is minimal requires one to price out *all* the columns of A. For convergence of the simplex algorithm, any s will do provided $\bar{c}_s < 0$ on each iteration. It has been observed empirically that sweeping through the $j = (1, \ldots, n)$ in batches and optimizing over each batch can reduce CPU time dramatically in many applications. Hence, this scheme, called *partial pricing*, is used in most commercial linear programming systems. In particular, the **PARTIAL PRICE** option of MINOS allows the user to specify a value p so that the $m+n$ columns for the decision and slack variables are divided equally into p batches, A_k and I_k for $k = 0, \ldots, p-1$. A tolerance parameter, $\bar{t}_d \geq t_d$, is set for each sweep through each group, and it is reduced dynamically on each iteration. If the previous iteration found an x_j such that $\bar{c}_j < 0$ for some j in the k^{th} group, then the next group searched is A_{k+1}, I_{k+1} (When $k+1 = p$, then the next group searched is A_0, I_0). If no j is found in this group such that $(\bar{c}_j/\|\pi\|) < -\bar{t}_d$, then the next group $(k+2)$ (with provisions for recycling back to zero) is searched. If all groups are searched and no candidate is found, $\bar{t}_d \geq t_d$ is reduced, unless $\bar{t}_d$ is already as small as t_d, in which case the current solution to the linear program is declared optimal.

The partial pricing scheme described above will most likely reduce the amount of pricing out and hence the average amount of computation done per iteration, but may increase the number of iterations necessary to find the optimal solution, because the "best candidate" (in terms of minimal $\bar{c}_j$) may not always be chosen on each iteration. There is, therefore, a tradeoff that the user of a linear programming system must make, and it is not clear what rule one should use to choose the value of p. In the MINOS manual, it is recommended that for time-stage models with t time periods, the user should choose $p = t$. The example below shows that this may not always be optimal.

The problem HITACHI was described as a 48-period model by Nishiya and Funabashi (1984). Brady (1986) wrote a program and used it to generate an MPS deck for this model. MINOS 5.0 was then used to solve this problem on an IBM 3081 using different partial price options. The size of the problem was quite large, with $m = 1008$, $n = 1632$ and 3741 nonzero elements in A. All the options of MINOS were set to their default mode except the **PARTIAL PRICE** option, which was varied from $p = 1$ to $p = 48$. The results are shown in Table 3.1.

It is interesting to note that as p increased from 1 to 8, the number of iterations and the total CPU time decreased. For $p > 8$, both iterations and CPU time increased with p except for $p = 16$. The outlyer at $p = 16$ indicates that it is difficult to predict the results of varying p. It is also interesting to note that while HITACHI has 48 time periods, $p = 48$ was not the best choice.

Partial Price Option	Number of Iterations	CPU Time (Seconds)
1	790	24.07
2	791	21.10
4	781	19.72
6	781	19.32
8	770	18.77
12	807	19.17
16	799	18.93
24	822	19.45
48	879	20.47

Table 3.1 Results for HITACHI problem

This example demonstrates (and this is true for many others) that how to choose p to minimize the running time of solving a linear program is not obvious. Hence, for all other results in this study, the **PARTIAL PRICE** option of MINOS was not used, and the default choice of $p = 1$ was set by the program. In later sections, there will be discussion on how partial pricing could possibly be used to reduce the running time of different algorithms implemented herein.

3.6 Random versus Real-World Problems

When comparing algorithms for linear programming, it is necessary to determine whether randomized or real-world problems will be used. Up to now, no one has been able to create a random model that generates problems that are representative and as varied as those encountered in practice. A number of characteristics seem to be inherent in real-world problems, but are difficult to capture and model stochastically. We can, however, comment about some of their properties.

1. *Sparsity.* Most real-world problems are sparse. Typically, they are sparse because if they were not, they never would be formulated: it would require too much effort on the part of the model formulator to collect the data for a dense matrix. For example, specifying 100 percent of a 1000 by 2000 dense matrix would require the input of 2 million nonzeros. Thus sheer data collection effort on the part of the user seems to be the main reason that linear programs that we encounter in practice are sparse. Another reason is that even if dense models were formulated, they probably could not be solved. Sparsity is usually measured by the *density* $d = \frac{\epsilon}{m \times n}$ of the $m \times n$ matrix A, which is assumed to have ϵ nonzero entries.
2. *Structure and near triangularity of the basis.* In problems where submatrices correspond to interactions between two different sets of technologies, many of the technologies are independent and hence the corresponding elements are zero, yielding large sparse blocks in A. Indeed, there appear to be a great variety of structures encountered in the sparsity patterns in real world problems. This is apparent in time staged and multi staged linear programs, which are often described as staircase in structure. There may be special structures within the blocks of a staircase structure. Networks are another general class. These also may have sparsity patterns within the network

which are specially structured. One approach to solving large-scale linear programs is to exploit special structures, as has been done by Fourer (1982). A difficulty with this approach is that there appear to be enough slight differences in structure from problem to problem that no characteristic pattern among them is readily apparent. In the course of solving a linear program by the simplex method, one can usually rearrange the rows and columns of the various bases encountered so that they are nearly triangular. Near triangularity makes it a relatively inexpensive operation to represent the basis as a product of lower and upper triangular matrices and preserves much of the original sparsity from iteration to iteration.

3. *Degeneracy*. Random models, with probability one, do not have degenerate vertices, while most real-world problems typically have many degenerate vertices. Recent work by Krueger (1986) has analyzed the effects of degeneracy in random polyhedra, when such degeneracy can be included among the randomly generated models. He found that as n increases, the limiting behavior of the average performance of the simplex method does not change when the number of degeneracies and m are fixed. Degeneracy occurs in practice because modellers tend to overspecify a problem, thereby generating redundant constraints. Furthermore, models tend to have constraints that convert one variable into a sum of other variables; this can cause degeneracy. For example, degeneracy can occur in practice because modellers include technologies which have a number of activities wholly dependent on whether a particular technology is used or not. These can have constraints linking them. If the technology is not used, each of these constraints will generate a basic variable with value zero.
4. *Unit Elements*. Many of the nonzero elements in the constraint matrix A of most real-world problems have values of $+1$ or -1. This often happens because many of the constraints are "bookkeeping" constraints that a modeller puts in to find the sums of certain variables. If x_1 represents the amount invested in some activity, then some constraint will add x_1 to its sum while another constraint will subtract x_1 from its sum. The column will then have two nonzeros, 1.0 and -1.0, which occur in the rows corresponding to inputs and outputs that x_1 affects. Such relations are fairly common in models and are used to summarize the results of models in various ways. Spreadsheets are very popular because they express the same kind of relations in a convenient format of two-way tables that allow easy horizontal and vertical addition.
5. *Feasibility*. Most models for randomly generating problems are deliberately primal feasible (e.g., Kuhn and Quandt (1962)). If a problem is given in primal feasible canonical form, then the Phase II simplex algorithm applies, while if a problem is initially given in dual-feasible canonical form, the dual simplex algorithm applies. Composite simplex algorithms are most applicable to the case where the problem is both primal and dual infeasible in its given canonical form.

With these considerations as backround, it was decided to do all testing on 12 real-world problems. These problems were collected from various sources by the Systems Optimizations Laboratory in the Department of Operations Research at Stanford University, and have been available for many years. The problems range in size from small to medium, and are believed to be representative of linear programs that are solved by most practitioners. All the problems have crash bases generated by MINOS which are primal and dual infeasible. The statistics as to size and density for the twelve problems are shown in Table 3.2.

Problem Name	Size			Density Percent
	Rows	Cols	NonZeros	
AFIRO	28	32	88	9.82
ADLITTLE	56	97	465	8.56
SHARE2B	99	79	802	10.25
SHARE1B	118	225	1182	4.45
BEACONFD	174	262	3476	7.62
ISRAEL	175	142	2358	9.49
BRANDY	221	249	2150	3.91
E226	226	282	3038	4.77
CAPRI	272	353	1786	1.86
BANDM	306	472	2659	1.84
STAIR	357	467	3857	2.31
ETAMACRO	401	688	2489	0.90

Table 3.2 Problem Statistics

3.7 Testing Methodology

Except for the earlier reported tests on partial pricing, all computational tests in this study were done on a Digital Equipment Corporation VaxStation II with 2 megabytes of main memory. The operating system was MicroVMS version 4.3, and the VAX Fortran Compiler, version 4.1, was used with the default options (which includes an optimizer). All tests on a particular problem were run as a batch job to eliminate the effect of differences in time of terminal input/output on the results.

The algorithms were implemented using MINOS version 5.0 as a framework, using as modules for the various algorithms the same MINOS subroutines whenever possible, in order to minimize any distortion of results due to special routines written specially for each algorithm. All the options of MINOS were set at their defaults. For each specific algorithm, four runs were made. These were classified as to whether the problem was unscaled or scaled, and whether the initial crash basis was the **UB** crash basis or the **SB** crash basis.

A special routine was written to act as a timer, which used a VMS routine that returns the CPU time used in centiseconds. Hence, all timing results are accurate to at most a hundredth of a second. Mainly timed were the execution of the entire code, including input of the data and output of the results, and the execution of the MINOS subroutine M5SOLV. This subroutine takes as input a linear program in memory, and outputs in memory the indices of the optimal basis. All algorithms were implemented as subroutines of this main solving routine. The CPU time of this central routine is used to compare the speeds of each algorithm. Since each algorithm was a variant of the simplex method, the number of simplex iterations was also recorded.

Average timing statistics were also collected for the following operations: refactorizing the basis, ratio tests, pricing out, updating the factorization of the basis, solving for π, solving for the representation of the incoming column, and updating the current solution x. The CPU time required to collect these statistics was negligible compared to the total execution time of the routine M5SOLV, and did not influence the results.

It should be noted that a re-run of the same algorithm on the same problem can produce slightly different CPU times, but never a difference in iteration counts. This can happen because after every iteration, MINOS prints one line to an output file on a disk to record information about that iteration. The time to perform this operation can vary slightly, depending upon the position of the read/write heads of the disk drive. Because of the nature of the VMS operating system, it is impossible to recreate exactly the conditions of the disk drive prior to each run of the solver. I assumed that this source of error was small and could be ignored in comparing the CPU times of various algorithms.

For each algorithm, four tables that correspond to one of two crash bases and one of two scaling options will be presented in the sections that follow. The vertical ordering of the results will be by the increasing number of rows in each problem. Corresponding to the four tables, the effects of scaling for the algorithm will be analyzed. The four tables for each composite algorithm will be compared to the four tables in the next section. Section 8 contains a comparitive study of the entire set of algorithms.

3.8 Results for the Primal Simplex Algorithm

For the twelve problems, the standard two-phase procedure described in Section 2 was run. The results for this algorithm are presented in Tables 3.4–3.7. Table 3.3 below summarizes these results for the effects of scaling and the crash basis on the two-phase algorithm.

Starting Basis		UB		SB	
	Scaling Status	Unscaled	Scaled	Unscaled	Scaled
UB	Unscaled		4W-7L	4W-4L	4W-7L
	Scaled	7W-4L		6W-5L	6W-4L
SB	Unscaled	4W-4L	5W-6L		5W-5L
	Scaled	7W-4L	4W-6L	5W-5L	

Table 3.3 Two-Phase Algorithm—Comparison of Scaling versus Nonscaling

For any pair of tables, the twelve problems were compared and the number of "wins" and "losses" were counted. The entry W/L in Table 3.3 corresponds to the number of wins and losses, respectively. A win W is scored for each problem when the combination shown on the left margin of the table had less CPU time and less iterations than the one shown on the top margin of the table. A loss L is similarly scored. If either the CPU time was equal or the number of iterations was equal, or the number of iterations was higher, but the CPU time was less, neither a win or a loss was credited. Hence, $W + L \neq 12$ in all of the cases. There are three possible explanations why the latter inconsistency occurred:

1. There are slight inaccuracies in the timing mechanism (e.g., AFIRO);
2. The number of iterations is close, but the simplex paths taken are slightly different, with one path using somewhat sparser LU factorizations of the basis (e.g., BRANDY); and,
3. The number of iterations is close, but more time is spent in Phase II, where the computation of π is more expensive, since $\hat{c}_B$ is more dense in Phase II than in Phase I (e.g., ETAMACRO).

It seems that there is no clear cut advantage to choosing scaling or one of the two different starting crash bases. There is a slight advantage to using the SCALE option of MINOS (scaled with the **SB** basis) as opposed to not using that option (unscaled with the **UB** basis).

Problem Name	Phase I Iterations	Phase II Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	2	4	6	0.52
ADLITTLE	28	96	124	12.04
SHARE2B	82	42	124	15.20
SHARE1B	143	144	287	45.91
BEACONFD	8	30	38	8.36
ISRAEL	109	243	352	73.88
BRANDY	176	116	292	68.15
E226	111	460	571	147.74
CAPRI	140	143	283	62.02
BANDM	186	247	433	134.10
STAIR	198	319	517	273.25
ETAMACRO	251	422	673	200.59

Table 3.4 Two-Phase Algorithm—Unscaled with UB Basis

Problem Name	Phase I Iterations	Phase II Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	2	4	6	0.52
ADLITTLE	22	64	86	8.40
SHARE2B	84	52	136	16.17
SHARE1B	135	116	251	36.92
BEACONFD	7	32	39	8.45
ISRAEL	93	211	304	62.38
BRANDY	253	139	392	94.59
E226	69	417	486	118.55
CAPRI	153	63	216	46.62
BANDM	188	291	379	112.67
STAIR	180	126	306	167.99
ETAMACRO	282	477	759	230.12

Table 3.5 Two-Phase Algorithm—Scaled with UB Basis

Problem Name	Phase I Iterations	Phase II Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	2	4	6	0.51
ADLITTLE	31	79	110	10.16
SHARE2B	62	26	88	10.42
SHARE1B	135	169	304	47.53
BEACONFD	7	31	38	8.23
ISRAEL	65	225	290	59.23
BRANDY	178	114	292	70.65
E226	101	302	403	99.22
CAPRI	130	161	291	64.43
BANDM	188	253	441	133.45
STAIR	214	349	563	278.14
ETAMACRO	484	464	948	274.01

Table 3.6 Two-Phase Algorithm—Unscaled with SB Basis

Problem Name	Phase I Iterations	Phase II Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	2	4	6	0.50
ADLITTLE	30	69	99	9.37
SHARE2B	74	47	121	14.31
SHARE1B	144	116	260	39.12
BEACONFD	6	33	39	8.45
ISRAEL	41	186	227	48.68
BRANDY	216	161	377	93.71
E226	101	368	469	117.97
CAPRI	178	57	235	48.46
BANDM	280	254	534	164.76
STAIR	193	196	389	211.16
ETAMACRO	219	669	888	275.66

Table 3.7 Two-Phase Algorithm—Scaled with SB Basis

Section 4. The Weighted Objective Method

In this section, the weighted objective method is discussed. The algorithm is similar to the classical big-M method and absolute value penalty methods of nonlinear programming. These similarities are noted and the implementation of the algorithm is discussed as well.

4.1 The Weighted Objective Method: Algorithm

The main idea behind the weighted objective method is to create a new linear program combining the objectives used in Phase I and Phase II in the standard procedure. The Phase I objective d^Tx is combined with the Phase II objective c^Tx to form a weighted combination $z = d^Tx + \sigma c^Tx$, which is then minimized. If the new objective has an optimal solution that is infeasible for the original problem, then σ is decreased by a factor of 10 and the method is continued. If σ is reduced by a factor of 10 five times (a MINOS default) and a feasible solution has not been reached, then σ is set to 0, and the usual two-phase approach is resumed. If the weighted linear program remains infeasible after a 10^5 reduction, it is most likely the original linear program is infeasible or σ was initially chosen much too large. The usual 'most negative reduced cost rule' is used to choose the incoming variable. The algorithm that follows looks very similar to the two-phase procedure, with the major differences being the way $\hat{c}$ is defined and in the scheme for reducing σ.

```
procedure weighted_objective;
    finished ← false;
    count ← 5;
    while not finished do
        begin
        if needs_factorization(B) then factorize(B);
        if infeasible_basis(B) then $\hat{c} \leftarrow d + \sigma c$ else $\hat{c} \leftarrow c$;
        $\boxed{\textbf{FindPI}\text{: Solve } \pi^T B = \hat{c}_B \text{ for } \pi;}$
        $\boxed{\textbf{PriceOut}\text{: Find } s \text{ such that } x_s \text{ has minimum reduced cost } \bar{c}_s;}$
        optimal ← $\bar{c}_s > 0$;
        if not optimal then
            begin
            $\boxed{\textbf{Represent}\text{: Solve } Bp = [A\ I]_{\cdot s} \text{ for } p;}$
            $\boxed{\textbf{RatioTest}\text{: Find } r \text{ such that } x_{B_r} \text{ blocks the change in } x_s;}$
            unbounded ← $r = 0$
            if not unbounded then
                begin
                $q \leftarrow \mathcal{B}_r$;
                $\boxed{\textbf{Update}\text{: Update value of } x;}$
                pivot(B, s, q, r);
                end;
            end;
        infeasible ← infeasible_basis(B) and optimal;
        if infeasible and $\sigma > 0$ then
        begin
        infeasible ← false;
        if count > 1 then
        begin
            count ← count − 1;
            $\sigma \leftarrow \sigma/10$;
            end
         else $\sigma \leftarrow 0$;
        end;
    finished ← optimal or unbounded or infeasible;
    end;
```

MINOS provides a `WEIGHT ON LINEAR OBJECTIVE` option that allows the user to choose the initial value of σ. It is not clear what heuristics a user should use when choosing σ. In their research on the barrier method, Gill *et al.* (1986) found that a choice of $\sigma = .1/\|c\|$ was promising for that method. The same value was used for the computational tests contained herein.

4.2 The Big-M Method

The big-M method is very similar to the weighted objective method. In its simplest form, a linear program with equality constraints is given:

$$\begin{array}{rl} \text{minimize} & c^T x \\ \text{subject to} & Ax = b, \\ & x \geq 0. \end{array} \tag{4.1}$$

By multiplying equation i by -1 if $b_i < 0$, it transforms this linear program so that $b_i \geq 0$ for all i. A set of artificial variables u is added and a new objective is formed:

$$\begin{array}{rl} \text{minimize} & c^T x + M e^T u \\ \text{subject to} & Ax + Iu = b, \\ & x, u \geq 0, \end{array} \tag{4.2}$$

where $e^T = (1, 1, \ldots, 1)$. If M is a very large number, then minimization of this form will be equivalent to the textbook form of Phase I of the simplex method. (A slightly different form is used in MINOS). If M is given a specific value, then this method is equivalent to the weighted objective method.

The origins of this method are not clear. Hadley (1962) refers to the method as "Charnes' $-M$ method." The earliest work by Charnes using the notation of M appears in Charnes, Cooper, and Henderson (1953). There, Charnes credits Dantzig's first paper on the simplex method (1951) with the idea that became the big-M method. There, Dantzig referred to assigning "small" weights to the infeasibility objective. (Since his linear program was a *maximization* problem, the "small" must have been in the sense of being initially very negative.) There also seems to be an independent development which used the notation w for M as used above. This appears in the works of Orden (1951) and Mayberry (1951), who credit Dantzig with the idea. Orden worked closely with Dantzig in the Pentagon around 1950 and the idea may have been informally suggested to Orden by Dantzig. In the implementation of the method, it is not clear whether M is to be used symbolically, or whether M was actually a preassigned a fixed value. The big-M method has been reported to have been quite successful when a model with minor changes is solved repeatedly and the value of M (experimentally arrived at) is not too large.

4.3 Absolute Value Penalty Methods

Gill *et al.* (1981) describe the use of absolute value penalty functions when applied to nonlinear programs with nonlinear constraints. When applied to the linear program (4.1), this method becomes equivalent to the weighted objective method.

Given the nonlinear program

$$\begin{array}{ll} \text{minimize} & F(x) \\ \text{subject to} & \hat{c}_i(x) = 0, \quad i = 1, \ldots, t, \end{array} \tag{4.3}$$

the absolute value penalty function is defined as

$$P_A(x, \rho) = F(x) + \rho \sum_{i=1}^{t} |\hat{c}_i(x)|, \tag{4.4}$$

for some scalar $\rho \geq 0$. If one lets

$$\hat{c}_i(x) = b_i - a_i^T x = u_i, \tag{4.5}$$

where u represents the artificial variables of the big-M method, then

$$P_A(x, \rho) = c^T x + \rho \sum_{i=1}^{t} u_i. \tag{4.6}$$

This is the same function used by the big-M method with $\rho = M$. Hence, the absolute value penalty function, when applied to linear programs, yields the weighted objective method.

4.4 Implementation and Computational Results

The weighted objective method is quite simple to implement. Since the objective vector c is stored as part of the matrix A in MINOS, a pass must be made through A in order to compute $\|c\|$. MINOS provides the mechanism for reducing σ if the current basis is optimal for the weighted objective but still infeasible. Hence, there is a small amount of extra work necessary for this method that occurs at the beginning of execution and during the case when σ must be reduced. It should also be noted that partial pricing is available with this algorithm, if the user so desires.

The computational results are presented in Tables 4.3–4.6. The column for "Infeasibility" iterations refers to iterations while the solution was infeasible. The column for "Post-Infeasibility" iterations refers to iterations that occurred after a feasible solution was found, which were iterations identical to Phase II of the two-phase algorithm. It is interesting to note that for the problem BRANDY *without scaling, the first feasible solution* found for both starting bases was the optimal solution. This phenomena did not occur for this problem when the problem was scaled.

Table 4.5 summarizes the effects of scaling for the weighted objective algorithm. The meaning of each entry is the same as in Section 3. The choice of starting basis or scaling seems to have little effect on this algorithm.

Table 4.6 compares the weighted objective algorithm to the two-phase algorithm. For each scaling/starting-basis pair, the two tables were compared separately for CPU time and iterations. The W/L entry indicates the number of wins and losses for the weighted objective algorithm versus the two-phase algorithm. In comparing the results, there were many instances where the number of iterations was unchanged, indicating that the choice of initial weight may be too small. From this table, we can see that the weighted objective algorithm tends to decrease the number of pivots necessary to reach the optimal solution, but does not always reduce the CPU time. This is because in the case where the number of iterations on a particular problem is equal, the CPU time for the weighted objective algorithm will be higher due to the initial computation of $\|c\|$.

Problem Name	Infeasibility Iterations	Post-Infeas. Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	2	4	6	0.53
ADLITTLE	31	110	141	13.52
SHARE2B	77	41	118	14.94
SHARE1B	143	144	287	47.80
BEACONFD	8	29	37	8.46
ISRAEL	148	230	378	83.47
BRANDY	311	0	311	79.44
E226	115	373	488	129.47
CAPRI	194	35	229	52.05
BANDM	184	177	361	115.27
STAIR	198	319	517	263.17
ETAMACRO	217	389	606	181.98

Table 4.1 Weighted Objective Algorithm—Unscaled with UB Basis

Problem Name	Infeasibility Iterations	Post-Infeas. Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	2	4	6	0.53
ADLITTLE	22	63	85	8.35
SHARE2B	94	36	130	15.85
SHARE1B	128	136	264	40.74
BEACONFD	7	32	39	8.83
ISRAEL	90	209	299	64.16
BRANDY	228	138	366	88.30
E226	70	411	481	121.27
CAPRI	171	45	216	48.87
BANDM	186	207	393	123.99
STAIR	180	126	306	174.74
ETAMACRO	194	584	778	256.10

Table 4.2 Weighted Objective Algorithm—Scaled with UB Basis

Problem Name	Infeasibility Iterations	Post-Infeas. Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	2	4	6	0.51
ADLITTLE	26	73	99	9.43
SHARE2B	60	25	85	9.82
SHARE1B	149	157	306	52.61
BEACONFD	7	31	38	8.58
ISRAEL	92	215	307	65.25
BRANDY	237	0	237	61.09
E226	101	390	491	127.13
CAPRI	195	51	246	54.91
BANDM	235	212	447	141.83
STAIR	215	275	490	243.72
ETAMACRO	471	462	933	286.92

Table 4.3 Weighted Objective Algorithm—Unscaled with SB Basis

Problem Name	Infeasibility Iterations	Post-Infeas. Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	2	4	6	0.52
ADLITTLE	26	67	93	8.99
SHARE2B	90	55	145	18.02
SHARE1B	123	75	198	31.52
BEACONFD	6	33	39	8.95
ISRAEL	43	184	227	49.52
BRANDY	208	143	351	85.14
E226	106	348	454	112.87
CAPRI	204	53	257	57.71
BANDM	280	254	534	169.87
STAIR	193	196	389	204.45
ETAMACRO	219	513	732	235.86

Table 4.4 Weighted Objective Algorithm—Scaled with SB Basis

Starting Basis		UB		SB	
	Scaling Status	Unscaled	Scaled	Unscaled	Scaled
UB	Unscaled		5W-6L	5W-5L	6W-5L
	Scaled	6W-5L		8W-3L	5W-5L
SB	Unscaled	5W-5L	3W-8L		5W-6L
	Scaled	5W-6L	5W-5L	6W-5L	

Table 4.5 Weighted Objective Algorithm - Comparison of Scaling versus Nonscaling

Starting Basis		UB		SB	
Scaling Status		Unscaled	Scaled	Unscaled	Scaled
Weighted Objective	CPU	6W-6L	3W-9L	5W-6L	6W-6L
versus Two-Phase	Iterations	6W-3L	5W-3L	6W-4L	5W-2L

Table 4.6 Weighted Objective Algorithm Comparison to Two-Phase Algorithm

Section 5. The Markowitz Criterion

The Markowitz criterion is an alternative rule for selecting the incoming variable in Phase I of the simplex method. In this section, the algorithm and its implementation are discussed and computational results are presented.

5.1 The Markowitz Criterion: Algorithm

The criterion informally suggested to Dantzig by Harry Markowitz and discussed in Dantzig's book (1963) is a different rule for choosing the incoming variable x_s in Phase I. After a feasible point is found, the usual Phase II procedure is resumed. The reduced cost $\bar{d}_j$ represents the decrease in the infeasibility form $w = d^T x$ per unit change in x_j. Similarly, the reduced cost $\bar{c}_j$ represents the decrease in the objective $z = c^T x$ per unit change in x_j. The idea behind the Markowitz criterion is to choose x_s in such a way that there is a maximum decrease of the objective form z per unit decrease of the infeasibility form w. Mathematically, s is chosen so that

$$\frac{\bar{c}_s}{-\bar{d}_s} = \min_{\bar{d}_j < 0} \frac{\bar{c}_s}{-\bar{d}_s} \qquad (5.1.1)$$

The algorithm is guaranteed to converge, since $\bar{d}_s < 0$, and w will decrease at every iteration, assuming some lexicographic device for resolving degeneracy. The procedure for this algorithm in terms of the simplex method is as follows:

```
procedure Markowitz_Criterion;
    finished ← false;
    while not finished do
        begin
        if needs_factorization(B) then factorize(B);
        FindPI: Solve $\pi^T B = c_s$ for $\pi$;
        if infeasible_basis(B) then begin
            FindPI: Solve $\bar{\pi}^T B = d_s$ for $\bar{\pi}$;
            PriceOut: Find $s$ such that $x_s$ has min ratio $\frac{c_s}{-\bar{d}_s}$ for $d_s < 0$;
            optimal ← $\bar{d}_s \geq 0$;
            end
        else begin
            PriceOut: Find $s$ such that $x_s$ has minimum reduced cost $c_s$;
            optimal ← $c_s \geq 0$;
            end
        if not optimal then
            Represent: Solve $Bp = [A\ I]_s$ for $p$;
            RatioTest: Find $r$ such that $x_{s_r}$ blocks the change in $x_s$;
            unbounded ← $r = 0$
        if not unbounded then
            begin
            $q \leftarrow B_r$;
            Update: Update value of $x$;
            pivot($B, s, q, r$);
            end;
        end;
    infeasible ← infeasible_basis($B$) and optimal;
    finished ← optimal or unbounded or infeasible;
    end;
```

There are some interesting properties of this algorithm. If $\bar{c}_s > 0$, then a change in any nonbasic variable that decreases w will increase z. At such a point, the current solution to the linear program is most likely superoptimal. If $\bar{c}_s < 0$, then changing x_s will decrease simultaneously both the infeasibility form w and the objective form z. This would appear to be the most desired choice of s. However, it appears that any such s are found for only the first few iterations of the method. Thereafter, $\bar{c}_s > 0$. This observation leads one to believe that the Markowitz criterion has the feature that the primal and dual feasibility conditions are at war with each other. In the next section, the computational results will verify this statement.

The Markowitz criterion relates in an interesting way to the weighted objective method. The weighted objective is

$$z = d^T x + \sigma c^T x. \tag{5.1.2}$$

This objective can be multiplied by $1/\sigma$ so that the weight is now on the infeasibility form. The reduced costs for this new objective are equal to

$$\left(\frac{1}{\sigma}\right) z = \left(\frac{1}{\sigma}\right) d + c. \tag{5.1.3}$$

For any nonbasic x_j with $d_j < 0$, if

$$\frac{1}{\sigma} = \frac{c_j}{-d_j}, \tag{5.1.4}$$

then

$$\left(\frac{1}{\sigma}\right) z_j = 0. \tag{5.1.5}$$

By comparing equations (5.1.1) and (5.1.4), one can see that the Markowitz criterion is dynamically balancing the weight on the infeasibility form $d^T x$ so that $z_j = 0$. It chooses the minimal weight so that this condition is true.

5.2 Implementation and Computational Results

The Markowitz criterion requires some extra work per iteration when the basis is primal infeasible. In Phase I of the two-phase method, only the prices and reduced costs for the infeasibility form $d^T x$ are computed. For the Markowitz criterion, the prices and reduced costs for the objective form $c^T x$ may need to be computed as well. There are two possible methods of implementation (where t_d is the optimality tolerance as defined in Section 3):

1. In the main loop that computes the reduced costs d_j for each nonbasic x_j, c_j is computed as well. The values of the elements of the sparse matrix A are each retrieved once from memory.
2. For each nonbasic x_j, c_j is computed only when $d_j < -t_d \pi$. Hence c_j is not always computed. However, when c_j is computed, a similar pricing loop is used as when computing d_j. Therefore, each element of the sparse matrix A is retrieved *twice* from memory.

In either case, an extra call to the procedure **FindPI** must be made in order to compute π. It would seem that the second of the two options is more efficient. Experimental evidence indicates that the second option was better in over 90 percent of the computational tests. Hence, the results reported below use that option.

In order to use equivalent tolerance criteria, only those j for which $d_j < -t_d \pi$ are used when evaluating the expression (5.1.1). If all $d_j \geq -t_d \pi$, then the current solution is considered optimal for the Phase I linear program and the usual Phase II procedure is begun.

The computational results for the Markowitz criterion are presented in Tables 5.1–5.4. The column for "Infeasibility iterations" refers to the number of iterations performed when the basis was primal infeasible. The column for "Post Infeasibility iterations" refers to the number of iterations performed after a feasible solution was found. It is interesting to note that for many of the problems, the first feasible solution found was the optimal solution, independent of the starting crash basis or of scaling. This indicates that the Markowitz criterion usually finds a basis that is superoptimal and maintains superoptimality until the optimal basis is found. Most of the problems require few Phase II iterations. Hence, the initial feasible point found by the Markowitz criterion is close to the optimal solution in terms of Hamming distance.

Table 5.5 summarizes the effects of scaling for the Markowitz criterion. The meaning of each entry is the same as in Section 3. The choice of starting basis or scaling seems to have no overall effect on this algorithm. It is interesting to note that the **SB** basis was better than the **UB** basis when scaling was used, but that the results were *reversed* when scaling was not used.

Table 5.6 compares the Markowitz criterion to the two-phase algorithm. For each scaling/starting basis pair, the two tables were compared separately for CPU time and iterations. The $W-L$ entry indicates the number of wins and losses for the Markowitz criterion versus the two-phase algorithm. From this table, we can see that there seems to be no advantage to using the Markowitz criterion, except for a few problems.

Problem Name	Infeasibility Iterations	Post-Infeas. Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	7	3	10	0.78
ADLITTLE	117	0	117	12.77
SHARE2B	150	2	152	22.73
SHARE1B	196	13	209	36.99
BEACONFD	56	11	67	15.85
ISRAEL	315	0	315	80.35
BRANDY	359	0	359	103.32
E226	632	0	632	208.88
CAPRI	203	33	236	60.19
BANDM	599	0	599	225.55
STAIR	1021	0	1021	593.92
ETAMACRO	673	191	864	321.73

Table 5.1 Markowitz Criterion Unscaled with UB Basis

Problem Name	Infeasibility Iterations	Post-Infeas. Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	7	3	10	0.79
ADLITTLE	100	0	100	11.34
SHARE2B	83	10	93	12.44
SHARE1B	153	2	155	27.25
BEACONFD	59	21	80	19.50
ISRAEL	260	25	285	69.34
BRANDY	465	0	465	133.88
E226	649	27	676	214.83
CAPRI	190	39	229	58.19
BANDM	700	0	700	265.61
STAIR	1766	0	1766	984.42
ETAMACRO	568	157	725	267.54

Table 5.2 Markowitz Criterion Scaled with UB Basis

Problem Name	Infeasibility Iterations	Post-Infeas. Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	7	3	10	0.78
ADLITTLE	131	0	131	14.14
SHARE2B	138	8	146	20.72
SHARE1B	268	12	280	50.14
BEACONFD	75	4	79	21.30
ISRAEL	271	0	271	70.31
BRANDY	479	0	479	137.23
E226	566	14	580	188.23
CAPRI	280	14	294	74.87
BANDM	684	0	684	260.22
STAIR	1238	14	1252	637.92
ETAMACRO	883	67	950	358.07

Table 5.3 Markowitz Criterion—Unscaled with SB Basis

Problem Name	Infeasibility Iterations	Post-Infeas. Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	7	3	10	0.77
ADLITTLE	90	1	91	10.09
SHARE2B	74	10	84	11.11
SHARE1B	247	44	291	50.16
BEACONFD	61	12	73	19.46
ISRAEL	262	0	262	68.14
BRANDY	448	0	448	128.26
E226	640	0	640	204.22
CAPRI	308	13	321	83.62
BANDM	575	0	575	211.62
STAIR	1849	0	1849	1088.95
ETAMACRO	524	195	719	259.51

Table 5.4 Markowitz Criterion—Scaled with SB Basis

Starting Basis		UB		SB	
	Scaling Status	Unscaled	Scaled	Unscaled	Scaled
UB	Unscaled		5W-6L	8W-3L	5W-5L
	Scaled	6W-5L		6W-3L	3W-8L
SB	Unscaled	3W-8L	3W-6L		4W-7L
	Scaled	5W-5L	8W-3L	7W-4L	

Table 5.5 Markowitz Criterion—Comparisons of Scaling versus Nonscaling

Starting Basis		UB		SB	
Scaling Status		Unscaled	Scaled	Unscaled	Scaled
Markowitz Criterion	CPU	2W-10L	2W-10L	0W-12L	2W-10L
versus Two-Phase	Iterations	4W-8L	4W-8L	2W-10L	3W-9L

Table 5.6 Markowitz Criterion—Comparison to Two-Phase Algorithm

Section 6. The Self-Dual Parametric Algorithm

In this section, Dantzig's self-dual parametric algorithm is discussed. This algorithm has been shown (Lustig, 1987) to be equivalent to Lemke's algorithm for solving linear complementarity problems when applied to linear programs as a special case. The implementation of the self-dual algorithm is described and computational results are presented.

6.1 Dantzig's Self-Dual Parametric Algorithm

Dantzig (1963) describes the self-dual parametric algorithm as a method which does not require an initial primal or dual feasible solution, but which uses the concepts of the primal and dual simplex algorithms extensively. Both the weighted objective method and the Markowitz criterion are alternative rules for choosing the *incoming* variable s in Phase I of the simplex method. After s is chosen, the *outgoing* variable r is chosen by the normal ratio test. In contrast, the self-dual algorithm changes the *order* of choosing the incoming variable s and the outgoing variable r on each iteration depending on the current basic solution to the problem. Ravindran (1970) gave an equivalent statement of the algorithm by modifying Lemke's method to modify only the primal simplex tableau.

It is easier to understand the method using some of his notation. The discussion of the algorithm is simpler when applied to the linear program

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \geq b, \\ & x \geq 0. \end{array} \tag{6.1.1}$$

A parameter θ is used in this method and the linear program (6.1.1) is transformed to

$$\begin{array}{ll} \text{minimize} & (c+\theta d)^T x \\ \text{subject to} & -Ax + Iu = -b + \theta f, \\ & x, u \geq 0 \end{array} \tag{6.1.2}$$

where $u \in \Re^m$ is a vector of slack variables which form an initial basis,

$$f_i = \begin{cases} 1, & \text{if } b_i > 0; \\ 0, & \text{if } b_i \leq 0 \end{cases}, \qquad \text{for} \quad i = 1, \ldots, m, \tag{6.1.3}$$

and

$$d_j = \begin{cases} 1, & \text{if } c_j < 0; \\ 0, & \text{if } c_j \geq 0 \end{cases}, \qquad \text{for} \quad j = 1, \ldots, n. \tag{6.1.4}$$

If $\theta = +\infty$, then the basis $u_1, u_2, \ldots, u_m$ is both primal and dual feasible for the parametric linear program (6.1.2). There exists a value

$$\theta_0 = -\min\left\{ \min_{i=1,\ldots,m} \left\{ \frac{-b_i}{f_i}, f_i > 0 \right\}, \min_{j=1,\ldots,n} \left\{ \frac{c_j}{d_j}, d_j > 0 \right\} \right\} \tag{6.1.5}$$

and an $\epsilon > 0$ such that for $\theta = \theta_0 - \epsilon$, the current basic solution to the parametric linear program (6.1.2) is infeasible for only one primal variable or one dual variable (assuming no ties exist when equation (6.1.5) is evaluated—this occurs when degeneracy exists in the parametric linear program). If $\theta = \theta_0$, then the parametric linear program has either multiple primal or multiple dual solutions. If a reduced cost $c_j + \theta_0 d_j$ for a nonbasic x_j was zero, one could set $s = j$ and attempt to make x_s basic, using the primal simplex ratio test

$$r = \arg\min_{i=1,\ldots,m} \left\{ \frac{-b_i + f_i\theta_0}{-a_{is}}, -a_{is} > 0 \right\} \tag{6.1.6}$$

to determine r. If a primal basic variable $u_i = -b_i + \theta_0 f_i$ was zero, then one could set $r = i$ and use the dual simplex ratio test

$$s = \arg\min_{j=1,\ldots,n} \left\{ \frac{c_j + d_j\theta_0}{a_{rj}}, -a_{rj} < 0 \right\} \tag{6.1.7}$$

to determine s. After exchanging x_s and u_r in the basis, the new basic solution is optimal for $\theta = \theta_0$. Both f and d must be updated to values $\bar{f}$ and $\bar{d}$ in the same way as $-b$ and c are updated to values $\bar{b}$ and $\bar{c}$. Dantzig (1963) and Ravindran (1970) give different proofs showing that θ can now be reduced to a value below θ_0 while maintaining primal and dual feasibility. The system is now in the same form as the parametric linear program (6.1.2) by a relabeling of the variables. The process can be continued until $\theta = 0$. At this point, a primal and dual feasible solution to the original linear program (6.1.1) has been obtained. If the ratio test performed in (6.1.6) has no $-a_{is} > 0$ or the ratio test performed in (6.1.7) has no $-a_{rj} < 0$, then the original linear program (6.1.1) is primal or dual infeasible (or both).

In the framework of the revised simplex method, the algorithm is as follows:

```
procedure Self_Dual($\bar{f}, \bar{d}$);
    finished ← false;
    $\theta \leftarrow +\infty$;
    while not finished do begin
        optimal ← false;
        if needs_factorization($B$) then factorize($B$);
        | RatioTest: Find $s$ or $r$ such that $x_{B_r}$ or $\bar{c}_s$ blocks the change in $\theta$ |
        if blocks($\bar{c}_s$) then begin
            | Represent: Solve $Bp = [A\ I]_{\cdot s}$ for $p$; |
            | RatioTest: Find $r$ such that $x_{B_r} + \theta\bar{f}_r$ blocks the change in $x_s$; |
            unbounded_infeasible ← $r = 0$;
            if not unbounded_infeasible then begin
                | FindPI: Solve $\gamma^T B = e_r$ for $\gamma$; |
                | PriceOut: Compute $\bar{\gamma} = A_{r\cdot} - \gamma[A\ I]$; |
            end
        end
    else if blocks($x_{B_r}$) then begin
```

```
        $\boxed{\textbf{FindPI}\text{: Solve } \gamma^T B = e_r \text{ for } \gamma;}$
        $\boxed{\textbf{PriceOut}\text{: Compute } \bar{\gamma} = A_{r\cdot} - \gamma[A\ I];}$
        $\boxed{\textbf{RatioTest}\text{: Find } s \text{ such that } \bar{c}_s + \theta\bar{d}_s \text{ blocks the change in } \pi_r;}$
        *unbounded_infeasible* ← $s = 0$;
        if not *unbounded_infeasible* then
            $\boxed{\textbf{Represent}\text{: Solve } Bp = [A\ I]_{\cdot s} \text{ for } p;}$
        end
    *finished* ← *optimal* or *unbounded_infeasible*;
    if not *finished* then begin
        $q \leftarrow \mathcal{B}_r$;
        $\boxed{\textbf{Update}\text{: Update values of } x, \bar{c}, \bar{d}, \bar{f};}$
        *pivot*$(B, s, q, r)$;
        end;
    end;
```

There are some things to note about this form of the algorithm. The variable *unbounded_infeasible* indicates that the original linear program has either an unbounded objective function or no feasible solution. If this variable is set to **true**, then one can use the normal two-phase simplex method to determine the infeasibility or unboundedness of the problem. The vector $\bar{\gamma}$ is needed to update $\bar{c}$ and $\bar{d}$. In the normal revised simplex method, $\bar{c}$ is recomputed each iteration by pricing out. In the revised simplex form of the self-dual method, it is more efficient to compute $\bar{\gamma}$ by a pricing operation and then update $\bar{c}$ and $\bar{d}$ using the formulas

$$\bar{c} \leftarrow \bar{c} - \delta_1\bar{\gamma} \quad \text{and} \quad \bar{d} \leftarrow \bar{d} - \delta_2\bar{\gamma}, \tag{6.1.8}$$

where

$$\delta_1 = \frac{\bar{c}_s}{\bar{\gamma}_s} \quad \text{and} \quad \delta_2 = \frac{\bar{d}_s}{\bar{\gamma}_s}. \tag{6.1.9}$$

This method eliminates one full call to the routine **PriceOut**.

It should be noted that the routine name *Self_Dual*$(\bar{f}, \bar{d})$ has $\bar{f}$ and $\bar{d}$ as parameters. This is because the initial choice of f and d must satisfy $-b + \theta f \geq 0$ and $c + \theta d \geq 0$ for some $\theta \geq 0$. Hence the initial values of f and d defined in equations (6.1.3) and (6.1.4) are specific instances of the parameters $\bar{f}$ and $\bar{d}$, respectively. In Section 7, a variant of the self-dual method that uses a different choice for the initial values of $\bar{f}$ and $\bar{d}$ is discussed.

6.2 Implementation and Computational Results

As compared to the two-phase simplex method, the self-dual algorithm must perform some extra computational work per iteration. This extra work can be separated into two categories—extra ratio tests and extra work in the procedure **Update**. In the two-phase method, there is one ratio test for the m primal basic variables on every iteration. In the self-dual algorithm, a ratio test is used to compute the new value of θ, by testing the m primal basic variables and the n reduced costs for the nonbasic variables. If a dual variable blocks the decrease of θ, the primal simplex algorithm is used and another ratio test for the m primal variables is used. If a primal variable blocks the decrease of θ, then the dual simplex algorithm is used and another ratio test for the n reduced costs is used. Therefore, on each iteration, either $2m+n$ or $m+2n$ variables are tested in various ratio tests.

On each iteration in the two-phase simplex method, only the values of the m primal basic variables must be updated. In the self-dual method, the values of $\bar{c}$, $\bar{d}$ and $\bar{f}$ must be updated as well. Hence $2n+m$ extra variables must have their values updated besides the m components of x.

For theoretical convergence of the self-dual method, θ should decrease on each iteration. However, due to degeneracies in the parametric linear program (6.1.2), θ may remain unchanged. Because of round-off errors, θ may actually *increase* by an extremely small amount. Hence, on each iteration, a tentative value of $\theta = \hat{\theta}$ is computed via the ratio test. Then θ is computed by taking the minimum of the previous value of θ and $\hat{\theta}$, i.e.,

$$\theta \leftarrow \min(\theta, \hat{\theta}). \tag{6.2.1}$$

Other round-off errors may cause the values $x+\theta\bar{f}$ and $\bar{c}+\theta\bar{d}$ to violate their feasibility constraints by more than the primal and dual feasibility tolerances, respectively. When MINOS checks to see if the current solution is too infeasible (in accordance with the CHECK FREQUENCY option), the current parametric solution is checked as well. If a component of the parametric solution violates its respective feasibility tolerance, then the respective component of $\bar{f}$ or $\bar{d}$ is reset so that the current parametric solution is feasible. This allows the algorithm to continue from the current solution without increasing the value of θ.

In the above discussion of the self-dual method, it was assumed that the linear program had every variable bounded below by zero and that all of the linear constraints were of the same type. The method was started from the basic variable set consisting of the m slack variables. In the context of MINOS, the self-dual method is started from a crash basis B, and each variable may have a finite lower or upper bound, or both. This affects the sign of the nonzeros of $\bar{f}$. Given this initial basis, the nonbasic variables are represented by set $\mathcal{N}$ such that $N = A_{\mathcal{N}}$. Each of the nonbasic variables $x_{\mathcal{N}}$ is set equal to one of its bounds. The equations for the linear program can be rewritten as

$$Bx_{B} + Nx_{\mathcal{N}} = 0, \tag{6.2.2a}$$
$$l \le x \le u. \tag{6.2.2b}$$

Since B is nonsingular, x_B can be computed by solving equation (6.2.2a). Now f_i is chosen according to whether x_{B_i} violates its upper or lower bound, i.e.

$$f_i = \begin{cases} 1 & \text{if } x_j < l_j,\ j = B_i; \\ -1 & \text{if } x_j > u_j,\ j = B_i; \\ 0 & \text{otherwise.} \end{cases} \tag{6.2.3}$$

Then $l_B \le x_B + \theta f \le u_B$ for all sufficiently large θ if each of the variables in the initial crash basis has exactly one finite bound.

If a variable has finite lower and upper bounds, then the parametric value $x_B + \theta f$ will satisfy one bound for large values of θ, but not the other. It turns out that the most difficult computational issue in the self-dual method is the manipulation of primal basic variables that have finite lower and upper bounds. Suppose x_1 is such a variable in the initial crash basis and that $B_1 = 1$. If $x_1 < l_1$ for the initial basic solution, then the parameterized value $x_1 + \theta f_1 > u_1$ for large θ. After a pivot in the self-dual method, the value of x_1 may exceed l_1 in such a way that $x_1 + \theta \bar{f}_1 > l_1$ for all $\theta \ge 0$. Furthermore, $x_1 + \theta \bar{f}_1 > u_1$ for the current value of θ. In order for the self-dual method to converge, it is necessary to treat such conditions specially.

In the theoretical version of the self-dual method, each component of x has a lower bound of 0 and no upper bound. If a variable x_1 has an upper bound u_1, then the constraint

$$x_1 + s = u_1 \tag{6.2.4}$$

is added to the problem, where s is the slack variable for the upper bound constraint. In the context of MINOS, it is necessary to compute the value of s implicitly. If such a row (6.2.4) were added to the MINOS constraint set, then the variable s would be in the initial basis. Furthermore, the initial basic solution would satisfy

$$\begin{pmatrix} & B & & 0 \\ 1 & 0 \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} x_B \\ s \end{pmatrix} = \begin{pmatrix} -Nx_N \\ u_1 \end{pmatrix}. \tag{6.2.5}$$

Similarly, the initial value of f satisfies

$$\begin{pmatrix} & B & & 0 \\ 1 & 0 \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} f \\ 0 \end{pmatrix} = \begin{pmatrix} \hat{f} \\ f_1 \end{pmatrix} \tag{6.2.6}$$

for some $\hat{f}$ (where f_1 is the initial value of f corresponding to x_1). If $\bar{f}_s$ represents the value of $\bar{f}$ corresponding to s on later iterations and x_1 remains basic, then

$$\begin{pmatrix} & B & & 0 \\ 1 & 0 \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} \bar{f} \\ \bar{f}_s \end{pmatrix} = \begin{pmatrix} \hat{f} \\ f_1 \end{pmatrix}. \tag{6.2.7}$$

Hence, the value of $\bar{f}_s$ can be computed as

$$\bar{f}_s = f_1 - \bar{f}_1. \tag{6.2.8}$$

Therefore, the parametric value of s is

$$s(\theta) = u_1 - x_1 + \theta(f_1 - \bar{f}_1). \tag{6.2.9}$$

In a primal simplex ratio test, either $x_1 + \bar{f}_1\theta$ is tested or $s(\theta)$ is, depending upon whether x_1 is increasing or decreasing as a function of the incoming basic variable. This computation is simple, as long as the initial values of f are saved throughout the execution of the method.

This argument also applies to the first ratio test done on each iteration to find the variable that blocks the decrease of θ. If $s(\theta)$ blocks the decrease of θ, then x_1 exits the basis and becomes nonbasic at its upper bound.

It should be noted that this problem does not occur with the parametric reduced costs, since the dual variables of the linear program in MINOS standard form have at least one infinite bound (ignoring the rare case of nonbasic free variables).

The computational results for the self-dual method are presented in Tables 6.2.1–6.2.4. The columns for primal and dual iterations correspond to the number of primal simplex and dual simplex iterations, respectively, done by the self-dual algorithm.

Table 6.2.5 compares the effects of scaling and the different crash bases on the self-dual algorithm. The meaning of each entry is the same as in Section 3. Table 6.2.6 compares the self-dual algorithm to the two-phase algorithm for iterations and CPU time. The entry $W - L$ indicates the number of times that the self-dual algorithm had less iterations (CPU time) than the two-phase algorithm. The self-dual algorithm seems to reduce the number of iterations on some problems, but the extra work per iteration increases the CPU time enough so that the self-dual algorithm is faster than the two-phase algorithm for only a few problems.

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.60
ADLITTLE	21	73	94	11.59
SHARE2B	44	63	107	15.96
SHARE1B	241	249	490	101.22
BEACONFD	27	11	38	9.56
ISRAEL	240	82	322	82.22
BRANDY	120	312	432	129.76
E226	277	201	478	163.66
CAPRI	28	144	172	48.37
BANDM	275	180	455	187.31
STAIR	307	351	658	381.97
ETAMACRO	472	373	845	362.25

Table 6.2.1 Self-Dual Algorithm—Unscaled with UB Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.62
ADLITTLE	33	39	72	8.57
SHARE2B	67	79	146	22.08
SHARE1B	128	185	313	61.90
BEACONFD	31	17	48	12.05
ISRAEL	177	60	237	61.20
BRANDY	218	325	543	161.91
E226	295	205	500	166.47
CAPRI	45	159	204	57.81
BANDM	182	161	343	131.11
STAIR	186	335	521	316.63
ETAMACRO	635	227	862	367.55

Table 6.2.2 Self-Dual Algorithm—Scaled with UB Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.65
ADLITTLE	29	94	123	14.79
SHARE2B	53	59	112	17.02
SHARE1B	127	128	255	50.85
BEACONFD	39	11	50	13.47
ISRAEL	210	21	231	58.00
BRANDY	175	226	401	122.26
E226	261	149	410	136.39
CAPRI	30	138	168	48.59
BANDM	288	265	553	224.54
STAIR	311	362	673	375.62
ETAMACRO	419	425	844	375.73

Table 6.2.3 Self-Dual Algorithm—Unscaled with **SB** Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.62
ADLITTLE	46	70	116	14.39
SHARE2B	91	81	172	25.37
SHARE1B	105	155	260	52.57
BEACONFD	36	12	48	11.92
ISRAEL	176	33	209	54.16
BRANDY	135	209	344	106.08
E226	268	177	445	145.49
CAPRI	31	190	221	62.50
BANDM	270	268	538	212.70
STAIR	184	287	471	288.15
ETAMACRO	668	256	924	400.67

Table 6.2.4 Self-Dual Algorithm—Scaled with **SB** Basis

Starting Basis		UB		SB	
	Scaling Status	Unscaled	Scaled	Unscaled	Scaled
UB	Unscaled		6W-5L	4W-4L	6W-5L
	Scaled	5W-6L		4W-6L	5W-5L
SB	Unscaled	4W-4L	6W-4L		5W-6L
	Scaled	5W-6L	5W-5L	6W-5L	

Table 6.2.5 Self-Dual Algorithm—Comparison of Scaling versus Nonscaling

Starting Basis		UB		SB	
Scaling Status		Unscaled	Scaled	Unscaled	Scaled
Self-Dual	CPU	2W-10L	1W-11L	2W-10L	0W-12L
versus Two-Phase	Iterations	5W-5L	4W-7L	4W-7L	4W-6L

Table 6.2.6 Self-Dual Algorithm—Comparison to Two-Phase Algorithm

Section 7. Variants of the Self-Dual Parametric Algorithm

In this section, two variants of the self-dual parametric algorithm are discussed. Both variants are motivated by considering the equivalence of the self-dual algorithm and Lemke's method, shown by Lustig (1987). The first method is equivalent to choosing a different covering vector for Lemke's algorithm. The second method tries to account for the prevalence of fixed variables in the initial crash basis.

7.1 Normalized Covering Vectors

When Lemke's algorithm is applied to the linear program

$$\begin{array}{rl} \text{minimize} & c^T x \\ \text{subject to} & Ax \geq b, \\ & x \geq 0, \end{array} \tag{7.1.1}$$

an extra variable θ is adjoined to form the set of equations

$$\begin{pmatrix} -b \\ c \end{pmatrix} + \begin{pmatrix} f \\ d \end{pmatrix} \bar{\theta} + \begin{pmatrix} 0 & A \\ -A^T & 0 \end{pmatrix} \begin{pmatrix} y \\ x \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix}, \tag{7.1.2a}$$

$$u^T y + v^T x = 0, \tag{7.1.2b}$$

$$u, v, x, y \geq 0. \tag{7.1.2c}$$

For the self-dual algorithm to converge, the initial values of f and d must be chosen so that the parametric primal basic variables and parametric reduced costs are nonnegative. Typically, f and d are chosen so that

$$f_i = \begin{cases} 1, & \text{if } b_i > 0; \\ 0, & \text{if } b_i \leq 0 \end{cases}, \quad \text{for} \quad i = 1, \ldots, m, \tag{7.1.3}$$

and

$$d_j = \begin{cases} 1, & \text{if } c_j < 0; \\ 0, & \text{if } c_j \geq 0 \end{cases}, \quad \text{for} \quad j = 1, \ldots, n. \tag{7.1.4}$$

In a study of the general linear complementarity problem, Eaves (1971) suggested varying the choice of the initial values of f and d when using Lemke's algorithm. The same suggestion, of course, is applicable to the self-dual algorithm. Krueger (1986) also investigated the effects of varying the initial values of f and d. For the self-dual algorithm, f and d can be chosen so that the various ratio tests remain unaffected if the rows or columns of the original linear program are rescaled. If the original problem is not scaled first, this will only be in the sense that the initial value of θ is *unit-free*, i.e., a change in units of any primal or dual variable will not change the initial value of θ with respect to this covering vector. If the original problem has been scaled first and then f and d chosen so that θ is to be initially scale-free, it will be so for all subsequent changes in θ.

Because an initial crash basis can contain both primal decision variables x_j, $j = 1, \ldots, n$, and slack variables u_i, $i = 1, \ldots, m$, it is important to treat each of them separately when developing a unit-free covering vector. Furthermore, the value d_j for the reduced cost $\bar{c}_j$ will depend upon whether its complementary variable was a decision variable or a slack variable.

Let $\hat{A} = [-A\ I]$, $\hat{c}^T = (c^T\, 0) \in \Re^{m+n}$, and relabel the m slack variables u_i to be x_{m+i} so that the linear program is now

$$\begin{aligned} \text{minimize} \quad & \hat{c}^T x \\ \text{subject to} \quad & \hat{A}x = -b, \\ & x \geq 0. \end{aligned} \tag{7.1.5}$$

Let the initial crash basis be represented by an index set $\mathcal{B}$. Suppose h decision variables x_{j_k}, $1 \leq k \leq h$, $1 \leq j_k \leq n$, and g slack variables x_{j_k}, $h+1 \leq k \leq m$, $n+1 \leq j_k \leq n+m$, $(m = g+h)$ are the initial basic variables and $\mathcal{B}_k = j_k$ so that the first h basic variables are decision variables. Let $\mathcal{N}$ index the n nonbasic variables. $B = \hat{A}_{\mathcal{B}}$ is nonsingular, and therefore the linear program (7.1.5) can be rewritten

$$\begin{aligned} \text{minimize} \quad & (c^T - c^T{}_{\mathcal{B}} B^{-1}\hat{A})x \\ \text{subject to} \quad & Ix_{\mathcal{B}} + B^{-1}\hat{A}_{\mathcal{N}}x_{\mathcal{N}} = -B^{-1}b, \\ & x \geq 0. \end{aligned} \tag{7.1.6}$$

Writing $\tilde{x} = x_{\mathcal{N}}$, $\tilde{c}^T = (c^T{}_{\mathcal{N}} - c^T{}_{\mathcal{B}}B^{-1}\hat{A}_{\mathcal{N}})$, $\tilde{A} = -B^{-1}\hat{A}_{\mathcal{N}}$, and $\tilde{b} = B^{-1}b$, this linear program can be rewritten as

$$\begin{aligned} \text{minimize} \quad & \tilde{c}^T\tilde{x} \\ \text{subject to} \quad & \tilde{A}\tilde{x} \geq \tilde{b}, \\ & \tilde{x} \geq 0. \end{aligned} \tag{7.1.7}$$

Lemke's method is initialized for (7.1.7) by writing

$$\begin{pmatrix} -\tilde{b} \\ \tilde{c} \end{pmatrix} + \begin{pmatrix} f \\ d \end{pmatrix}\bar{\theta} + \begin{pmatrix} 0 & \tilde{A} \\ -\tilde{A}^T & 0 \end{pmatrix}\begin{pmatrix} \tilde{y} \\ \tilde{x} \end{pmatrix} = \begin{pmatrix} \tilde{u} \\ \tilde{v} \end{pmatrix}, \tag{7.1.8a}$$

$$\tilde{u}^T\tilde{y} + \tilde{v}^T\tilde{x} = 0, \tag{7.1.8b}$$

$$\tilde{u}, \tilde{v}, \tilde{x}, \tilde{y} \geq 0, \tag{7.1.8c}$$

where $\tilde{y} = (y\ v)_{\mathcal{B}}$, $\tilde{u} = x_{\mathcal{B}}$, and $\tilde{v} = (y\ v)_{\mathcal{N}}$. The initial ratio test in Lemke's method for (7.1.8) is

$$\bar{\theta} = -\min\left\{ \min_{i=1,\ldots,m} \left\{ \frac{-\tilde{b}_i}{f_i}, f_i > 0 \right\}, \min_{j=1,\ldots,n} \left\{ \frac{\tilde{c}_j}{d_j}, d_j > 0 \right\} \right\}. \tag{7.1.9}$$

If a row or column of the original linear program (7.1.1) is multiplied by a scalar σ, then the ratio test (7.1.9) will be unit-free if f and d are chosen so that the blocking variable in (7.1.9) is independent of the scaling. Hence, f and d will depend on the original values of A, b, and c.

First, consider when a column of the original problem (7.1.1) is scaled by a value σ. Let x_1 be a variable in the initial crash basis. Then the initial value of $x_1 = -\tilde{b}_1$. Suppose that $-\tilde{b}_1 < 0$ so that x_1 is initially infeasible. If the column

$$\begin{pmatrix} c_1 \\ A_{\cdot 1} \end{pmatrix} \tag{7.1.10}$$

is multiplied by a value σ, so that

$$\begin{pmatrix} c'_1 \\ A'_{\cdot 1} \end{pmatrix} = \sigma \begin{pmatrix} c_1 \\ A_{\cdot 1} \end{pmatrix}, \tag{7.1.11}$$

then the value of x_1 in the scaled problem is $x'_1 = -\tilde{b}_1/\sigma$. We desire that the ratio

$$\frac{-\tilde{b}_1}{f_1} \tag{7.1.12}$$

be unaffected by this scaling. If

$$f_1 = \|\,(\, c'_1 \quad A'^T_{\cdot 1}\,)\,\|^{-1}, \tag{7.1.13}$$

then

$$\frac{x'_1}{f_1} = \frac{\frac{1}{\sigma}(-\tilde{b}_1)}{\frac{1}{\sigma}\|\,(\, c_1 \quad A^T_{\cdot 1}\,)\,\|^{-1}} = \frac{-\tilde{b}_1}{\|\,(\, c_1 \quad A^T_{\cdot 1}\,)\,\|^{-1}}. \tag{7.1.14}$$

Hence, the ratio (7.1.12) is not affected by any scaling of the column for x_1.

Suppose x_1 is initially nonbasic at its lower bound, and the initial value of the dual variable $\tilde{v}_1 = \tilde{c}_1 < 0$. If a scaling as in (7.1.11) is done, then the initial value of $\tilde{v}_1$ is $\tilde{v}'_1 = \sigma \tilde{c}_1$. In the initial ratio test for Lemke's method, the ratio

$$\frac{\tilde{c}_1}{d_1} \tag{7.1.15}$$

is computed. If

$$d_1 = \|\,(\, c'_1 \quad A'^T_{\cdot 1}\,)\,\|, \tag{7.1.16}$$

then the ratio (7.1.15) is

$$\frac{\tilde{v}'_1}{d_1} = \frac{\sigma \tilde{c}_1}{\sigma \|\,(\, c_1 \quad A^T_{\cdot 1}\,)\,\|} = \frac{\tilde{c}_1}{\|\,(\, c_1 \quad A^T_{\cdot 1}\,)\,\|}. \tag{7.1.17}$$

Hence, the ratio (7.1.15) is unaffected by any scaling of the column for x_1.

When a row of the original problem is scaled, the values of the slack variables and their reduced costs are affected. Suppose the first row $(\, A_{1\cdot} \quad b_1\,)$ is scaled by a value σ so that

$$(\, A'_{1\cdot} \quad b'_1\,) = \sigma (\, A_{1\cdot} \quad b_1\,). \tag{7.1.18}$$

Suppose $u_1 = b_1 - A_{1\cdot}x$ is in the initial crash basis and that $u_1 = \tilde{x}_{h+1} < 0$. If the above scaling is done, then $u_1' = \sigma u_1$. We desire that the ratio test

$$\frac{\tilde{x}_{h+1}}{f_{h+1}} \tag{7.1.19}$$

is independent of any scaling on the first row. If

$$f_{h+1} = \|(A_{1\cdot}' \quad b_1')\|, \tag{7.1.20}$$

then the ratio (7.1.19) is

$$\frac{u_1'}{f_{h+1}} = \frac{\sigma u_1}{\sigma\|(A_{1\cdot} \quad b_1)\|} = \frac{u_1}{\|(A_{1\cdot} \quad b_1)\|}. \tag{7.1.21}$$

Hence, this choice of f_{h+1} causes the ratio (7.1.19) to be unit-free.

Now suppose that u_1 is initially nonbasic at a value of $u_1 = 0$ and that its corresponding dual variable $\tilde{y}_1$ is negative. $\tilde{y}_1 = \tilde{c}_{h+1} = -c_B B^{-1} e_1$, where e_1 is the first column of the identity matrix. When the first row of A is scaled as in equation (7.1.18), the first column of B^{-1} is divided by σ. Hence, a scaling causes $\tilde{y}_1' = \tilde{y}_1/\sigma$. We desire that the ratio

$$\frac{\tilde{c}_{h+1}}{d_1} \tag{7.1.22}$$

be independent of scaling. If we set

$$d_1 = \|(A_{1\cdot}' \quad b_1')\|^{-1}, \tag{7.1.23}$$

then the ratio (7.1.22) is

$$\frac{\tilde{y}_1'}{d_1} = \frac{\frac{1}{\sigma}\tilde{c}_{h+1}}{\frac{1}{\sigma}\|(A_{1\cdot} \quad b_1)\|^{-1}} = \frac{\tilde{c}_{h+1}}{\|(A_{1\cdot} \quad b_1)\|^{-1}}, \tag{7.1.24}$$

which is seen to be independent of any scaling of the first row.

In summary, the value of f_i will depend on whether the i^{th} basic variable is a decision variable x_j, $1 \le j \le n$, or a slack variable x_j, $n+1 \le j \le m+n$. This distinction is determined by the value of $\mathcal{B}_i = j$. Hence, if $x_{\mathcal{B}_i} < 0$,

$$f_i = \begin{cases} \|(c_j \quad A_{\cdot j}^T)\|^{-1} & \text{if } j = \mathcal{B}_i,\ 1 \le j \le n \\ \|(A_{k\cdot} \quad b_k)\| & \text{if } k = \mathcal{B}_i - n,\ 1 \le k \le m \end{cases} \tag{7.1.25}$$

($f_i = 0$ if $x_{\mathcal{B}_i}$ is feasible). The value of d_j is determined for $\tilde{c}_j < 0$ by

$$d_j = \begin{cases} \|(c_j \quad A_{\cdot j}^T)\| & \text{if } 1 \le j \le n \\ \|(A_{k\cdot} \quad b_k)\|^{-1} & \text{if } k = j - n,\ 1 \le k \le m \end{cases} \tag{7.1.26}$$

($d_j = 0$ if $\bar{c}_j \geq 0$). These values can be computed prior to the first iteration of the self-dual algorithm.

The computational results for the self-dual method with a normalized covering vector are presented in Tables 7.1.1–7.1.4. The columns for primal and dual iterations correspond to the number of primal simplex and dual simplex iterations, respectively, done by the self-dual algorithm using that covering vector.

Table 7.1.5 summarizes the effects of scaling for the self-dual algorithm with a normalized covering vector. It seems that the choice of scaling or the starting basis has little effect on this variant.

Table 7.1.6 compares the self-dual algorithm with a normalized covering vector (NCV) to the two-phase algorithm. It seems that this variant offers an improvement over the two-phase procedure for only a few of the problems. The CPU time improvements occurred only when the problem was unscaled and started from the **UB** basis.

Table 7.1.7 compares the effects of using the normalized covering vector versus using the default vector in the self-dual algorithm. The results are mixed, and there does not seem to be a clear advantage to using either covering vector. It is interesting to note the results for when the problems were scaled and started from the **SB** basis. Here, the iteration counts were higher when using the normalized covering vector, but the CPU time was less in three of the cases. This is because for those three problems, less dual simplex iterations were done when the normalized covering vector was used, while the total number of iterations increased. This indicates the sensitivity of the self-dual algorithms (and its variants) to fluctuations in the number of primal and dual simplex iterations and their effects on the total CPU time.

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.62
ADLITTLE	46	70	116	14.39
SHARE2B	91	81	172	25.37
SHARE1B	105	155	260	52.57
BEACONFD	36	12	48	11.92
ISRAEL	176	33	209	54.16
BRANDY	135	209	344	106.08
E226	268	177	445	145.49
CAPRI	31	190	221	62.50
BANDM	270	268	538	212.70
STAIR	184	287	471	288.15
ETAMACRO	668	256	924	400.67

Table 7.1.1 Self-Dual Algorithm Normalized Variant—Unscaled with UB Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.61
ADLITTLE	51	42	93	11.36
SHARE2B	71	90	161	23.27
SHARE1B	206	178	384	74.34
BEACONFD	29	12	41	10.17
ISRAEL	305	76	381	93.66
BRANDY	213	345	558	161.11
E226	238	154	392	123.01
CAPRI	87	181	268	78.32
BANDM	253	199	452	175.63
STAIR	233	220	453	280.44
ETAMACRO	504	162	666	279.46

Table 7.1.2 Self-Dual Algorithm Normalized Variant—Scaled with UB Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.65
ADLITTLE	51	66	117	13.88
SHARE2B	53	82	135	19.23
SHARE1B	112	172	284	54.16
BEACONFD	33	6	39	10.05
ISRAEL	209	122	331	82.38
BRANDY	114	215	329	98.42
E226	324	195	519	171.33
CAPRI	57	195	252	71.79
BANDM	262	255	517	201.80
STAIR	307	217	524	298.66
ETAMACRO	437	228	665	289.38

Table 7.1.3 Self-Dual Algorithm Normalized Variant—Unscaled with SB Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.62
ADLITTLE	58	54	112	13.65
SHARE2B	70	101	171	24.56
SHARE1B	132	154	286	55.40
BEACONFD	36	43	79	20.24
ISRAEL	175	100	275	66.91
BRANDY	120	226	346	99.10
E226	280	176	456	145.00
CAPRI	45	181	226	63.37
BANDM	278	261	539	203.64
STAIR	184	186	370	220.98
ETAMACRO	444	356	800	352.97

Table 7.1.4 Self-Dual Algorithm Normalized Variant—Scaled with SB Basis

Starting Basis		UB		SB	
	Scaling Status	Unscaled	Scaled	Unscaled	Scaled
UB	Unscaled		4W-7L	4W-7L	7W-4L
	Scaled	7W-4L		4W-6L	6W-5L
SB	Unscaled	7W-4L	6W-4L		6W-5L
	Scaled	4W-7L	5W-6L	5W-6L	

Table 7.1.5 Self-Dual Algorithm Normalized Variant—
Comparison of Scaling versus Nonscaling

Starting Basis		UB		SB	
Scaling Status		Unscaled	Scaled	Unscaled	Scaled
Self-Dual (NCV)	CPU	3W-9L	0W-12L	0W-12L	0W-12L
versus Two-Phase	Iterations	3W-8L	2W-9L	4W-7L	5W-6L

Table 7.1.6 Self-Dual Algorithm Normalized Variant—
Comparison to Two-Phase Algorithm

Starting Basis		UB		SB	
Scaling Status		Unscaled	Scaled	Unscaled	Scaled
Self-Dual (NCV)	CPU	5W-7L	6W-6L	6W-5L	7W-4L
versus Self-Dual	Iterations	5W-6L	4W-7L	6W-5L	4W-7L

Table 7.1.7 Self-Dual Algorithm Normalized Variant—
Comparison to Self-Dual Algorithm

7.2 Fixed Variables

Some general linear programs have constraints of the form

$$a_i^T x = b_i. \tag{7.2.1}$$

When such a linear program is converted to MINOS standard form, the slack variable x_{n+i} corresponding to this constraint will have equal lower and upper bounds. Any variable x_j with equal lower and upper bounds is known as a *fixed variable*. In the dual of the linear program, the complement of a fixed variable is *free*; in other words, the complementary variable has infinite lower and upper bounds. Hence, once a fixed variable becomes nonbasic in the course of executing any variant of the simplex method, that variable will never become basic at a later iteration.

Fixed variables can appear in the initial crash basis, because they must be included in order to maintain the lower triangularity of the crash basis. If the fixed variable is not at its bound, then Phase I will consider the fixed variable to be infeasible and include the variable in the Phase I objective. Eventually, this will cause the fixed variable to attain its bound. At that point, the variable will exit the basis if the equation (7.2.1) is not redundant given the other linear equations in the linear program. There seems to be an intuitive advantage to having as few fixed variables in the set of basic variables as possible. It is interesting to note that the two-phase simplex method causes a fixed variable to exit the basis by choosing incoming columns before choosing exiting columns in each iteration.

The self-dual algorithm has the property that it can choose exiting columns before incoming columns. The idea behind this variant of the self-dual algorithm is to cause as many fixed variables as possible to exit the basis before proceeding with the usual self-dual method. Because of the freedom allowed in the initial choice of f and d, f can be chosen so that some fixed variable "wins" the initial ratio test in the self-dual algorithm. The dual-simplex method is invoked and that fixed variable exits the set of basic variables, never to reenter. The method can then be restarted by choosing a new f and d so that another fixed variable will now exit the basis. If all basic fixed variables are at their bounds or are nonbasic, then the usual self-dual algorithm is invoked. The algorithm will converge since either a fixed variable exits the basis on an iteration or θ is reduced.

The initial values of f and d are chosen in the default way and the new value of θ is computed. The fixed variable that is furthest from its bound is then chosen as the variable to become nonbasic. For example, suppose x_j is the i^{th} basic variable and is designated as the most infeasible fixed variable, and that the current value of $x_j > l_j$. Then f_i is modified so that

$$f_i = \frac{x_j - l_j}{\theta}, \tag{7.2.2}$$

which would force x_j to be the exiting variable if the initial ratio test in the self-dual algorithm were recomputed. After x_j exits the basis, f and d are reset to their initial values depending upon the new values of the basic variables x_B and reduced costs $\bar{c}$.

The fixed-variable variant of the self-dual algorithm can be used with the default initial choices of f and d (equations (7.1.3) and (7.1.4)) or with the normalized covering vector considered in the previous section. The computational results with the default covering vector are shown in Tables 7.2.1–7.2.4, while the results with the normalized covering vector are shown in Tables 7.2.5–7.2.8.

Tables 7.2.9–7.2.10 summarize the effects of scaling on the fixed-variable variant of the self-dual algorithm with both covering vectors. The results are varied enough to suggest no trends relative to this algorithm for scaling or the choice of basis.

Tables 7.2.11–7.2.12 compare the fixed-variable (FV) variant of the self-dual algorithm to the two-phase algorithm for the default covering vector and the normalized covering vector (NCV). For both covering vectors, the number of iterations decreased for some of the problems (especially when the problem was unscaled and using the **UB** basis), but improvements in CPU time occurred for only a few problems.

Tables 7.2.13–7.2.14 compares the fixed-variable (FV) variant of the self-dual algorithm to the regular self-dual algorithm for each of the covering vectors. This allows one to see if eliminating the fixed variables first yields any improvements for the self-dual algorithm. There seems to be no clear advantage or clear disadvantage to using this variant of the self-dual algorithm.

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.62
ADLITTLE	23	79	102	12.39
SHARE2B	44	63	107	15.85
SHARE1B	127	203	330	70.02
BEACONFD	27	10	37	9.46
ISRAEL	240	82	322	82.35
BRANDY	208	250	458	146.02
E226	286	148	434	147.25
CAPRI	48	176	224	64.78
BANDM	292	233	525	213.84
STAIR	246	146	392	233.80
ETAMACRO	407	254	661	299.10

Table 7.2.1 Self-Dual Algorithm Fixed Variable Variant—Unscaled with UB Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.62
ADLITTLE	28	42	70	8.25
SHARE2B	67	79	146	20.95
SHARE1B	148	135	283	56.03
BEACONFD	28	23	51	13.62
ISRAEL	177	60	237	61.32
BRANDY	214	302	516	161.35
E226	264	140	404	133.92
CAPRI	39	252	291	83.85
BANDM	293	177	470	180.18
STAIR	149	141	290	189.93
ETAMACRO	561	364	925	412.55

Table 7.2.2 Self-Dual Algorithm Fixed Variable Variant—Scaled with UB Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.61
ADLITTLE	44	84	128	15.47
SHARE2B	53	59	112	16.02
SHARE1B	136	150	286	60.61
BEACONFD	35	8	43	10.73
ISRAEL	210	21	231	60.16
BRANDY	213	161	374	121.40
E226	265	185	450	151.29
CAPRI	58	244	302	90.02
BANDM	296	278	574	236.15
STAIR	323	151	474	289.22
ETAMACRO	414	292	706	325.61

Table 7.2.3 Self-Dual Algorithm Fixed Variable Variant—Unscaled with **SB** Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.60
ADLITTLE	51	59	110	12.18
SHARE2B	91	81	172	24.95
SHARE1B	151	156	307	59.23
BEACONFD	34	24	58	15.36
ISRAEL	176	33	209	55.79
BRANDY	157	174	331	104.48
E226	364	206	570	184.81
CAPRI	47	249	296	89.98
BANDM	249	295	544	216.29
STAIR	158	173	331	203.05
ETAMACRO	309	162	471	205.79

Table 7.2.4 Self-Dual Algorithm Fixed Variable Variant—Scaled with **SB** Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.61
ADLITTLE	49	52	101	11.99
SHARE2B	60	81	141	20.19
SHARE1B	89	123	212	43.53
BEACONFD	28	10	38	9.87
ISRAEL	265	93	358	87.80
BRANDY	200	320	520	161.34
E226	303	143	446	149.89
CAPRI	71	144	215	61.64
BANDM	284	303	587	232.33
STAIR	249	148	397	230.72
ETAMACRO	299	204	503	227.17

Table 7.2.5 Self-Dual Algorithm Fixed Variable Normalized—Unscaled with UB Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.63
ADLITTLE	48	44	92	11.26
SHARE2B	71	90	161	23.44
SHARE1B	191	172	363	74.43
BEACONFD	29	10	39	9.95
ISRAEL	305	76	381	94.73
BRANDY	206	249	455	137.77
E226	219	150	369	116.69
CAPRI	72	209	281	82.75
BANDM	248	219	467	179.08
STAIR	285	170	455	287.49
ETAMACRO	358	205	563	248.36

Table 7.2.6 Self-Dual Algorithm Fixed Variable Normalized—Scaled with UB Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.65
ADLITTLE	54	47	101	10.17
SHARE2B	53	82	135	19.52
SHARE1B	80	113	193	40.47
BEACONFD	32	5	37	8.16
ISRAEL	209	122	331	80.80
BRANDY	185	230	415	130.76
E226	321	213	534	173.90
CAPRI	49	179	228	66.72
BANDM	308	232	540	216.70
STAIR	305	121	426	254.39
ETAMACRO	899	388	1287	593.00

Table 7.2.7 Self-Dual Algorithm Fixed Variable Normalized—Unscaled with **SB** Basis

Problem Name	Primal Iterations	Dual Iterations	Total Iterations	CPU Time (Seconds)
AFIRO	4	2	6	0.64
ADLITTLE	52	54	106	12.74
SHARE2B	70	101	171	24.34
SHARE1B	124	223	347	71.47
BEACONFD	41	22	63	16.61
ISRAEL	175	100	275	70.12
BRANDY	142	249	391	119.78
E226	283	170	453	143.52
CAPRI	61	207	268	80.18
BANDM	321	336	657	254.85
STAIR	247	153	400	254.91
ETAMACRO	321	206	527	233.27

Table 7.2.8 Self-Dual Algorithm Fixed Variable Normalized—Scaled with **SB** Basis

Starting Basis		UB		SB	
	Scaling Status	Unscaled	Scaled	Unscaled	Scaled
UB	Unscaled		5W-6L	8W-3L	5W-5L
	Scaled	6W-5L		6W-5L	8W-3L
SB	Unscaled	3W-8L	5W-6L		3W-7L
	Scaled	5W-5L	3W-8L	7W-3L	

Table 7.2.9 Self-Dual Algorithm Fixed Variable Variant—
Comparison of Scaling versus Nonscaling

Starting Basis		UB		SB	
	Scaling Status	Unscaled	Scaled	Unscaled	Scaled
UB	Unscaled		7W-4L	4W-6L	8W-2L
	Scaled	4W-7L		3W-7L	5W-6L
SB	Unscaled	6W-4L	7W-3L		6W-4L
	Scaled	2W-8L	6W-5L	4W-6L	

Table 7.2.10 Self-Dual Algorithm Fixed Variable Normalized—
Comparison of Scaling versus Nonscaling

Starting Basis		UB		SB	
Scaling Status		Unscaled	Scaled	Unscaled	Scaled
Self-Dual (Fixed Vars)	CPU	2W-10L	2W-10L	0W-12L	2W-10L
versus Two-Phase	Iterations	8W-3L	4W-7L	4W-7L	4W-7L

Table 7.2.11 Self-Dual Algorithm Fixed Variable Variant—
Comparison to Two-Phase Algorithm

Starting Basis		UB		SB	
Scaling Status		Unscaled	Scaled	Unscaled	Scaled
Self-Dual (FV) (NCV)	CPU	4W-8L	1W-11L	3W-9L	1W-11L
versus Two-Phase	Iterations	6W-4L	2W-8L	5W-6L	2W-9L

Table 7.2.12 Self-Dual Algorithm Fixed Variable Normalized—
Comparison to Two-Phase Algorithm

Starting Basis		UB		SB	
Scaling Status		Unscaled	Scaled	Unscaled	Scaled
Self-Dual (Fixed Vars)	CPU	6W-6L	6W-5L	6W-6L	6W-6L
versus Self-Dual	Iterations	5W-4L	5W-4L	4W-5L	4W-5L

Table 7.2.13 Self-Dual Algorithm Fixed Variable Variant—
Comparison to Self-Dual Algorithm

Starting Basis		UB		SB	
Scaling Status		Unscaled	Scaled	Unscaled	Scaled
Self-Dual (FV) (NCV)	CPU	7W-5L	5W-7L	6W-5L	5W-7L
vs. Self-Dual (NCV)	Iterations	5W-4L	6W-3L	5W-4L	4W-5L

Table 7.2.14 Self-Dual Algorithm Fixed Variable Normalized—
Comparison to Normalized Self-Dual Algorithm

Section 8. Summary and Comparisons of Results

In this section, the computational results are used to compare all the algorithms against each other. Although one would expect that one of the many possible composite algorithms would turn out to be a clear winner, the evidence suggests that no algorithm is best for all problems. The two-phase and self-dual algorithms were used for a sensitivity analysis, and the evidence indicates that the choice of the best algorithm is not clear even when the problems have the same sparsity structure. In the last section, some suggestions for further research are offered.

8.1 Comparing Many Algorithms

In the course of comparing two different simplex variants on a specific linear program, two computer runs are made. It is important to be conscious of any differences between the two runs that may affect the interpretation of the results of these runs. The major differences between two such runs will be in the choice of scaling and the choice of the initial basis. In the earlier sections, the effects of these choices for each algorithm were compared, and it seemed that choosing the SCALE option of MINOS or either starting crash basis had little effect on the results. In this type of comparison, the algorithm chosen was a fixed control and the choice of scaling and the starting bases were varied. Tables were also presented that compared each composite algorithm to the two-phase algorithm. Here, the choices of scaling and the starting basis were fixed when two algorithms were compared. From these pairwise comparisons, one can measure how often one algorithm was better than another. However, they offer no measure as to *how much* better that algorithm was.

A scoring system can be used to measure the relative speeds of the different composite algorithms to the two-phase algorithm. We are interested in comparing the relative decrease or increase in the iteration counts and CPU time, which we will call a *performance measure*. Let q_a be the value of a performance measure for Algorithm a $(a = 0, 1)$ on some problem. Suppose that Algorithm 0 is the two-phase algorithm, whose performance measures will be used as the standard for comparing all of the other algorithms. Then the *performance factor* p_i for Algorithm i is defined as

$$p_i = 100(\frac{q_i}{q_0}). \tag{8.1.1}$$

Hence, if the performance measure being used was CPU time, then a value $p_1 = 120$ would mean that Algorithm 1 was 20 percent slower than the two-phase algorithm, while, similarly, a value $p_1 = 70$ would mean that Algorithm 1 was 30 percent faster than the two-phase algorithm. It should be noted that the performance factors for the two-phase algorithm will always be 100.

Performance factors can be computed by comparing the runs of two algorithms on the same problem using the same scaling option and the same starting basis. Hence, for each starting basis, scaling option, and algorithm, 12 performance factors are computed (1 for each problem). The minimum, maximum, and average values of these 12 performance factors are then computed. In the tables that follow, the performance factors are rounded to the nearest integer. Performance factors are computed for iteration counts and CPU time.

Table 8.1.1 presents the performance factors for iterations counts. Each algorithm has at least one problem for which it gives a 24 percent improvement in the number of iterations as compared to the two-phase algorithm. The average number of iterations was in most

Algorithm	Scaling Status	Starting Basis	Iterations Perf. Factors Min	Max	Mean
Two-Phase	Unscaled	UB	100	100	100
		SB	100	100	100
	Scaled	UB	100	100	100
		SB	100	100	100
Weighted Objective	Unscaled	UB	81	114	97
		SB	81	122	97
	Scaled	UB	93	105	100
		SB	76	120	98
Markowitz Criterion	Unscaled	UB	73	197	125
		SB	92	222	144
	Scaled	UB	62	577	161
		SB	69	475	150
Self-Dual	Unscaled	UB	61	171	106
		SB	58	137	105
	Scaled	UB	78	170	111
		SB	91	142	107
Self-Dual (Normalized Covering Vector)	Unscaled	UB	74	192	109
		SB	70	153	107
	Scaled	UB	81	153	118
		SB	90	203	113
Self-Dual (Fixed Variables)	Unscaled	UB	76	157	98
		SB	74	130	105
	Scaled	UB	78	135	108
		SB	53	149	107
Self-Dual (Normalized Covering Vector) (Fixed Vars.)	Unscaled	UB	74	178	99
		SB	63	153	109
	Scaled	UB	74	149	114
		SB	59	162	114

Table 8.1.1 Performance Factors for Iteration Counts

cases comparable to that of the two phase algorithm. Most of the algorithms exhibit a wide range of performance factors for iteration counts as compared to the two phase algorithm.

Table 8.1.2 presents the performance factors for CPU time. Once again, all the algorithms had at least one problem that had less CPU time than the two phase algorithm. Overall, only the weighted objective algorithm seems to be consistently competitive. On the average, the self dual algorithm and its variants were 30 to 40 percent slower than the two phase algorithm.

These performance factors indicate that there is no algorithm that consistently per

Algorithm	Scaling Status	Starting Basis	CPU Time Perf. Factors		
			Min	Max	Mean
Two-Phase	Unscaled	UB	100	100	100
		SB	100	100	100
	Unscaled	UB	100	100	100
		SB	100	100	100
Weighted Objective	Scaled	UB	84	117	99
		SB	85	128	101
	Unscaled	UB	93	111	104
		SB	81	126	100
Markowitz Criterion	Scaled	UB	81	217	143
		SB	105	259	169
	Unscaled	UB	74	586	180
		SB	78	516	172
Self-Dual	Scaled	UB	78	220	133
		SB	75	173	136
	Unscaled	UB	98	188	139
		SB	111	177	135
Self-Dual (Normalized Covering Vector)	Scaled	UB	94	246	136
		SB	106	185	134
	Unscaled	UB	104	201	146
		SB	105	240	140
Self-Dual (Fixed Variables)	Scaled	UB	86	214	126
		SB	102	177	137
	Unscaled	UB	98	180	139
		SB	75	186	136
Self-Dual (Normalized Covering Vector) (Fixed Vars.)	Scaled	UB	84	237	124
		SB	85	216	139
	Unscaled	UB	98	202	144
		SB	85	197	144

Table 8.1.2 Performance Factors for CPU Times

forms better than any other algorithm. For each algorithm, there is at least one problem for which that algorithm is best. Given a problem, there seems to be no rule that allows one to choose an algorithm which will perform best on that problem.

8.2 Sensitivity Analysis

Users of commercial linear programming systems frequently solve a particular linear program and then solve some perturbation of that problem. This perturbation usually is of the same structure as the original problem, and may only differ in the elements of A, b, and c. It is often useful to solve such a perturbed problem by using an optimal basis of the original problem as the starting basis for the new problem. Usually, only a few pivots of the simplex method must be done to reach the optimal basis of the new problem.

A perturbation as described above can cause the original optimal basis to become either primal infeasible, dual infeasible, or both. In the case of primal infeasibility, the dual simplex method is usually used. In the case of dual infeasibility, Phase II of the two-phase algorithm is used. When both the primal and dual problems are infeasible, the two-phase algorithm is usually used. In such a case, however, the self-dual algorithm may perform well, since only a few iterations may be needed.

In order to analyze the application of the self-dual algorithm for sensitivity analysis, it is necessary to determine a "real-world" perturbation of a "real-world" problem. A random perturbation of a problem would most likely destroy any degeneracies that were present. Fortunately, the literature contains a good example of a "real-world" perturbation.

Manne (1973a) described the model DINAMICO in enough detail to allow the associated linear program to be defined in a modeling language such as GAMS (Bisschop and Meeraus, 1982). GAMS can be used to create an input file for MINOS to use to solve the problem. This procedure was followed, and the optimal basis for the problem was saved. The problem has 319 rows, 424 columns, and 4157 nonzeroes. Manne (1973b) (pg. 152) later described different alternatives to the initial problem that can be solved. All the alternatives described leave the sparsity structure of the matrix A unchanged, but change only either the primal or dual infeasibility of the original problem, not both simultaneously.

Alternative cases 1 and 2 from Manne's paper vary the growth rate g used in the model. Varying this rate changes the primal infeasibility of the optimal basis. Alternative case 5 from Manne's paper makes the optimal basis dual infeasible. Both of these variations were made simultaneously, and the growth rate g was varied from a value of 4 percent to 12 percent. The case $g = .07$ corresponds to the optimal basis. Note that when $g = .06$, the optimal basis for the case $g = .07$ was the same, and hence the only computations that MINOS performed were to determine that the initial basis was optimal without performing any pivots.

Growth Rate	Two-Phase		Self-Dual	
g	Itns	CPU	Itns	CPU
0.04	41	31.48	39	33.71
0.05	45	29.90	30	27.15
0.06	6	12.83	6	13.72
0.08	50	32.83	12	16.52
0.09	115	76.19	137	100.18
0.10	224	132.74	185	139.09
0.11	115	66.84	200	144.03
0.12	124	80.58	181	130.76

Table 8.2.1 Sensitivity Analysis Results

Table 8.2.1 shows the results for the two-phase algorithm and the self-dual algorithm with the default covering vector. Note that the self-dual algorithm had competitive iteration counts for most of the cases.

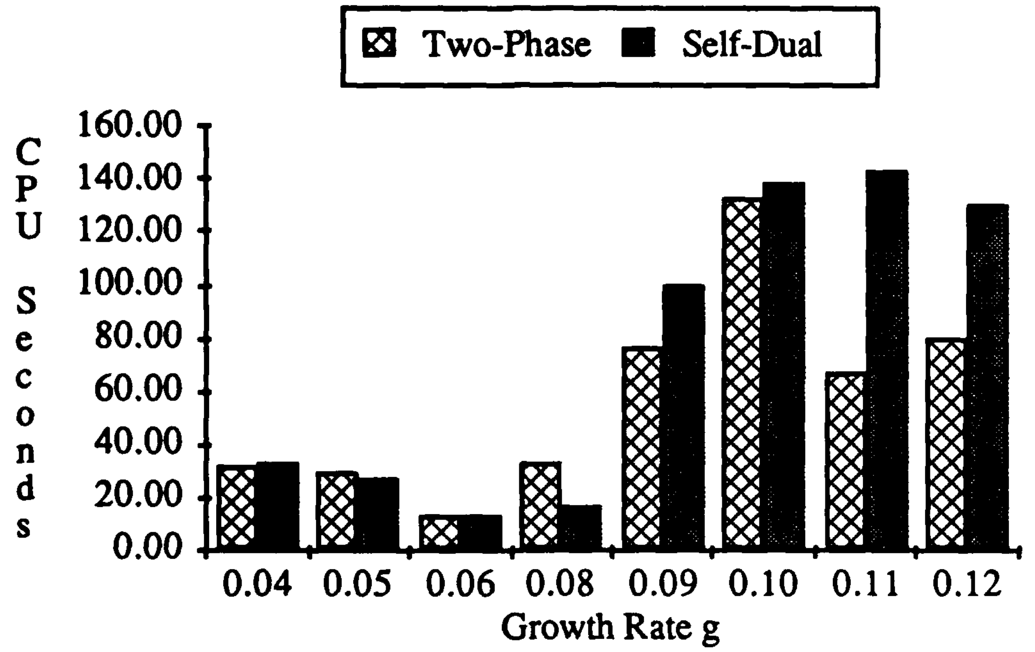


Figure 8.2.2 CPU Times for Sensitivity Analysis

It is interesting to compare the CPU time results using a graph as in Figure 8.2.2. The better CPU times are indicated by smaller bars. The self-dual algorithm outperforms the two-phase algorithm when $g = .05$ and $g = .08$. It is interesting to note the difference in performance for $g = .10$ as compared to $g = .09$, .11, or .12. It seems that the self-dual algorithm is better for small perturbations; conversely, the two-phase algorithm is better for large perturbations. However, it should be noted that the amount of CPU time did not increase monotonically as g increased. This suggests that it is difficult to predict the performance of either algorithm for arbitrary values of g.

All these problems were run from the same crash basis. The problems have the same structure for A, but approximately 120 of the nonzeros differ in value. These results suggest again that no one algorithm will perform consistently better than another on every problem, *even when the problems have the same structure*, i.e., problems which have the same pattern of nonzero data in A, but different values for some of those nonzeroes.

8.3 Summary and Further Research

The computational results contained in this study exhibit the sensitivity of the simplex method to different schemes for choosing the incoming and outgoing variables. Intuitively, one would expect that a composite algorithm would perform better than the two-phase algorithm, but only the weighted objective algorithm was consistently competitive. The two-phase algorithm and weighted objective method have a good average performance. Other algorithms seem to perform better than the two-phase algorithm on a few problems, indicating that no rule exists that allows one to choose the best method for a particular problem. The drawback of most of these alternative methods is that they require more work per iteration than the two-phase algorithm to choose s and r. They can only be faster than the two-phase algorithm if the number of iterations is significantly lowered (30 percent or more). Probably the best way to choose a variant of the simplex method is to use the two-phase algorithm first. If it performs poorly, then trying another variant of the simplex method may be in order. If n is much larger than m, the two-phase algorithm, weighted objective method, and the Markowitz criterion can each be used with partial pricing.

Some open questions that remain are:

1. What happens to a class of problems that are related in structure, but grow in size?
2. Do there exist other variants of the simplex method that will consistently do better than the two-phase algorithm?
3. Are there variants that have advantages over others in degenerate situations?
4. Can the methods of artificial intelligence be used to analyze a problem in advance, and then dynamically vary the choice of variant of the simplex method to use as the computation progresses?
5. Can one develop a matrix generator for creating problems that are in some sense random and yet representative of the variety of structures found in real-world problems?

8.4. Acknowledgements

I would like to thank George B. Dantzig, Richard W. Cottle, Michael A. Saunders and B. Curtis Eaves for their helpful insights during the course of this study. I would also like to thank the Department of Operations Research for allowing me to use their DEC VaxStation II and Macintosh computers for this work.

This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Section 9. Bibliography

Aho, A.V., Hopcroft, J.E., and Uliman,J.D. (1974). *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Menlo Park, California.

Bisschop, J. and Meeraus, A. (1982). On the development of a General Algebraic Modeling System in a strategic planning environment, *Mathematical Programming Study* **20**, pp. 1–29.

Brad). Stephen (1986). Private Communication.

Charnes, A., Cooper, W.W., and Henderson, A. (1953). *An Introduction to Linear Programming*, John Wiley & Sons, New York.

Cottle, R.W. (1972). Monotone solutions of the parametric linear complementarity problem, *Mathematical Programming* **3**, **2**, pp. 210–224.

Dantzig, G.B. (1951). "Maximization of a Linear Function of Variables Subject to Linear Inequalities," in *Activity Analysis of Production and Allocation* (T.C. Koopmans, ed.), John Wiley & Sons, New York.

Dantzig, G.B. (1963). *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey.

Dantzig, G.B. (1987). "Origins of the simplex method," in *Proceedings of the ACM conference on the History of Scientific and Numeric Computation*, May 13-15, 1987, Princeton, New Jersey, to appear.

Eaves, B.C. (1971). The linear complementarity problem, *Management Science* **17**, **9**, pp. 612–634.

Fourer, R. (1982). Solving staircase linear programs by the simplex method, 1: Inversion, *Mathematical Programming* **23**, **3**, pp. 274–313.

Gill, P.E., Murray, W., and Wright, M.H. (1981). *Practical Optimization*, Academic Press, San Francisco.

Gill, P.E., Murray, W., Saunders, M.A., Tomlin, J.A., and Wright, M.H. (1986). On projected Newton barrier methods for linear programming and an equivalence to Karmarkar's projective method, *Mathematical Programming* **36**, **2**, pp. 183–209.

Greenberg, H.J. (1978). "Pivot selection tactics," in *Design and Implementation of Optimization Software* (H.J. Greenberg, ed.), Sijthoff & Noordhoff, Alphen aan den Rijn, The Netherlands, pp. 143–174.

Hadley, G. (1962). *Linear Programming*, Addison-Wesley Publishing Company, Inc., Palo Alto, California.

Hoffman, A., Mannos, M., Soklowsky, D., and Wiegmann, N. (1953). Computational experience in solving linear programs, *Journal of the Society of Industrial and Applied Mathematics* **1**, pp. 17–33.

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming, *Combinatorica*, **4**, **4**, pp. 373–395.

Krueger, F. (1986). *d*-arrangements and Random Polyhedra, Ph.D. Thesis, Department of Operations Research, Stanford University.

Kuhn, H. and Quandt, R. (1962). An experimental study of the simplex method, *Symposia in Applied Mathematics* **15**, American Mathematical Society, Providence, RI, pp. 107-124.

Lemke, C.E. (1965). Bimatrix equilibrium points and mathematical programming, *Management Science* **11**, pp. 681-689.

Lemke, C.E. (1970). "Recent results on complementarity problems," in *Nonlinear Programming* (J.B. Rosen, O.L. Mangasarian, and K. Ritter, eds.), Academic Press, New York, pp. 349-384.

Lustig, I.J. (1987). The equivalence of Dantzig's self-dual parametric algorithm for linear programs to Lemke's algorithm for linear complementarity problems applied to linear programs, Report SOL 87-4, Department of Operations Research, Stanford University, CA.

Manne, A.S. (1973a). "DINAMICO, a dynamic multi-section multi-skill model," in *Multi-Level Planning: Case Studies in Mexico* (L.M. Goreux and A.S. Manne, eds.), North-Holland Publishing Company, Amsterdam, pp. 107-150.

Manne, A.S. (1973b). "Economic alternatives for Mexico: A Quantitative Analysis," in *Multi-Level Planning: Case Studies in Mexico* (L.M. Goreux and A.S. Manne, eds.), North-Holland Publishing Company, Amsterdam, pp. 151-172.

Mayberry, J. (1951). "A geometrical interpretation of the simplex method," in *Symposium on Linear Inequalities and Programming* (A. Orden and L. Goldstein, eds.), Project SCOOP, No. 10, Planning Research Division, Director of Management Analysis Service, Comptroller, USAF, Washington, DC.

McCammon, S.R. (1970). On complementary pivoting, Ph.D. thesis, Department of Mathematics, Rensselaer Polytechnic Institute, Troy, NY.

Megiddo, N. (1983). Linear-time algorithms for linear programming in $\Re^3$ and related problems, *SIAM Journal of Computing*, **12**, **4**, pp. 759-776.

MPS III mathematical programming system: User manual (1975), Ketron, Inc., Arlington, VA.

Murtagh, B. A. and Saunders, M. A. (1983). *MINOS 5.0* user's guide, Report SOL 83-20, Department of Operations Research, Stanford University, CA.

Nishiya. T. and Funabashi, M. (1984). A basis factorization method for multi-stage linear programming problems with an application to optimal operation of an energy plant, Working Paper, Systems Development Laboratory, Hitachi, Ltd..

Orden, A. (1951) "Application of the simplex method to a variety of matrix problems," in *Symposium on Linear Inequalities and Programming* (A. Orden and L. Goldstein, eds.). Project SCOOP, No. 10, Planning Research Division, Director of Management Analysis Service, Comptroller, USAF, Washington, DC.

Ravindran, A. (1970). Computational aspects of Lemke's complementary algorithm applied to linear programs, *Opsearch*, **7**, **4**, pp. 241-262.

Shamir, R. (1987). The efficiency of the simplex method: A survey, *Management Science*, **33**, **3**, pp. 301-334.

Wolfe, P. (1961). An extended composite algorithm for linear programming, Paper P-2373, The Rand Corporation, July, 1961.

Wolfe, P. and Cutler, L. (1963). "Experiments in linear [illegible]rogramming," in *Recent Advances in Mathematical Programming* (R. Graves and [illegible] Wolfe, eds.), McGraw-Hill, New York, pp. 177-200.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SOL 87-8	2. GOVT ACCESSION NO. AD-A183437	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Comparisons of Composite Simplex Algorithms		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Irvin J. Lustig		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0343
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Operations Research - SOL Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR-047-064
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research - Dept. of the Navy 800 N. Quincy Street Arlington, VA 22217		12. REPORT DATE June 1987
		13. NUMBER OF PAGES pp. 63
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Composite Simplex Algorithm, Computational Comparisons, Self-dual Parametric Algorithm, Weighted Objective Method, Markowitz Criterion.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(Please see other side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract

For almost forty years, the simplex method has been the method for choice for solving linear programs. The method consists of first finding a feasible solution to the problem (Phase I), followed by finding the optimum (Phase II). Many algorithms have been proposed which try to combine the processes embedded in the two-phase process. This study will compare the merits of some of these composite algorithms.

Theoretical and computational aspects of the Weighted Objective, Self-Dual Parametric, and Markowitz Criteria algorithms are presented. Different variants of the Self-Dual methods are discussed.

A large amount of computational experience for each algorithm is presented. These results are used to compare the algorithms in various ways. The implementatons of each algorithm are also discussed. One theme that is present throughout all of the computational experience is that there is no one algorithm which is the best algorithm for all problems.