Mathematical Programming 45 (1989) 475-501
North-Holland



# NEW CRASH PROCEDURES FOR LARGE SYSTEMS OF LINEAR CONSTRAINTS

Nicholas I.M. GOULD and John K. REID

*Computer Science and Systems Division, Harwell Laboratory, Oxfordshire, UK*

Many algorithms for solving linearly constrained optimization problems maintain sets of basic variables. The calculation of the initial basis is of great importance as it determines to a large extent the amount of computation that will then be required to solve the problem. In this paper, we suggest a number of simple methods for obtaining an initial basis and perform tests to indicate how they perform on a variety of real-life problems.

*Key words*: Linear constraints, simple bounds, initial basis, feasible point.

## 1. Introduction

In this paper we shall be concerned with finding a vector $\boldsymbol{x}$ that satisfies the system of linear equations

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{1.1a}$$

and simple bound constraints

$$\boldsymbol{b}_l \leq x \leq \boldsymbol{b}_u, \tag{1.1b}$$

where $\boldsymbol{A}$ is an $m$ by $n$ real matrix, $\boldsymbol{b}$ is a real $m$-vector, $\boldsymbol{b}_l$ and $\boldsymbol{b}_u$ are real $n$-vectors with possibly infinite coefficients and the inequalities (1.1b) are taken componentwise. We assume that $\boldsymbol{b}_l$ and $\boldsymbol{b}_u$ satisfy the inequality $\boldsymbol{b}_l \leq \boldsymbol{b}_u$. We will refer to the problem of finding a solution to (1.1) as the *feasible point problem* and say that any $\boldsymbol{x}$ which solves the problem is a *feasible point.* We shall be particularly concerned with the feasible point problem when $m$ and $n$ are large and the matrix $\boldsymbol{A}$ is sparse. Notice that it is straightforward to reduce a problem with general linear inequalities to this form by introducing an extra "slack" variable for each inequality.

Many optimization problems require the minimum or maximum value of a real function

$$f(\boldsymbol{x}) \tag{1.2}$$

over vectors $\boldsymbol{x}$ for which (1.1) is satisfied. Such problems are often called linearly constrained optimization problems. Of particular importance are the linear programming problem, where $f(\boldsymbol{x})$ is linear, and the quadratic programming problem, where $f(\boldsymbol{x})$ is quadratic. Typical algorithms for the solution of such problems start from

a feasible point and generate a sequence of "improved" feasible points, the linearity of (1.1) enabling feasibility to be maintained. For such schemes, it is then crucial that an initial feasible point be determined. Unfortunately, the problem of finding a feasible point is often as hard as solving the underlying optimization problem. Indeed, the feasible point problem has the same complexity bound as the linear programming problem (see, for instance, Papadimitriou and Steiglitz, 1982, pp. 170-173).

One way of tackling the feasible point problem is to introduce extra, *artificial*, variables $v_i$ and $w_i$ $(i=1,\ldots,m)$ and $y_i$ and $z_i$ $(i=1,\ldots,n)$ and consider the problem of minimizing

$$e^{\mathrm{T}}\begin{pmatrix} v+w \\ y+z \end{pmatrix} \tag{1.3a}$$

subject to the linear constraints

$$Ax+v-w+Ay-Az=b \tag{1.3b}$$

and simple bounds

$$b_{\mathrm{l}} \leqslant x \leqslant b_{\mathrm{u}}, \quad v \geqslant 0, \quad w \geqslant 0, \quad y \geqslant 0 \text{ and } z \geqslant 0, \tag{1.3c}$$

where $e$ is a vector of ones. Clearly, (1.1) has a feasible solution if and only if (1.3) has an optimal solution of zero. The problem (1.3) is one of many *initial point* or *phase-1* problems that have been suggested; others include replacing $e$ in (1.3a) by a vector of weights or including a contribution from $f(x)$.

Problem (1.3) is a linear program that may be solved by the simplex method (Dantzig, 1963). At each step of this method, the variables are partitioned into two sets called *basic* and *nonbasic* variables. There are always $m$ basic variables and the corresponding columns of

$$(A \quad I \quad -I \quad A \quad -A)$$

form a nonsingular matrix $B$ called the *basis matrix* or *basis* for short. The nonbasic components of $x$ always satisfy their bounds (each usually lies at a bound) and the nonbasic artificial variables always have the value zero. The corresponding values of the basic variables also satisfy their bounds and may be found from (1.3b) by solving a set of equations whose matrix is the basis $B$. The simplex method changes the basis by one column at a time in such a way that $B$ remains nonsingular, the bounds are satisfied and the function (1.3a) decreases. It continues until an optimal solution is found, which gives a feasible solution for the original problem if the corresponding value (1.3a) is zero.

Note that at a feasible point we may still have some artificial basic variables, though they must have the value zero; indeed this will always happen if the rank

of $\boldsymbol{A}$ is smaller than $m$ (for example if an equation is accidentally repeated) for otherwise $\boldsymbol{B}$ would be singular. We would normally expect most of the basic variables to have values away from their bounds.

Given any set of $m$ columns of $(\boldsymbol{A}\ \boldsymbol{I})$ that forms a nonsingular matrix, a basis and corresponding basic solution of (1.3) may be found as follows. Take the given set of columns to define a tentative basis $\boldsymbol{B}$. If column $i$ of $\boldsymbol{A}$ is not in $\boldsymbol{B}$, give the variable $x_i$ a value satisfying its bounds (1.1b). If column $i$ of $\boldsymbol{I}$ is not in $\boldsymbol{B}$, give the variable $v_i$ the value zero. Solve the equation

$$\boldsymbol{A}\boldsymbol{x}+\boldsymbol{v}=\boldsymbol{b} \tag{1.4}$$

for the remaining values. If any variable $v_i$ is negative, replace it in the basic set by the corresponding $w_i$ and change the sign of the corresponding column of $\boldsymbol{B}$. If any $x_i$ lies above its upper bound, replace it in the basic set by the corresponding $y_i$, setting the value of $y_i$ to the discrepancy and resetting $x_i$ to its upper bound. If any $x_i$ lies below its lower bound, replace it in the basic set by the corresponding $z_i$, set the value of $z_i$ to the discrepancy, reset $x_i$ to its lower bound, and change the sign of the corresponding column of $\boldsymbol{B}$.

Note that a practical implementation does not need to store two extra copies of $\boldsymbol{A}$ or any copies of $\boldsymbol{I}$. All that is necessary is to attach suitable flags to the basic variables to indicate their origin.

The time taken by the simplex algorithm in solving a particular problem (1.3) is very roughly proportional to $n_0$, the number of nonzero components of $\boldsymbol{v}$, $\boldsymbol{w}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ in the initial basic solution. ($n_0$ is a lower bound on the number of simplex iterations. In some implementations, when a variable $y_j$ or $z_j$ reaches zero, it can be replaced by $x_j$ by a very economical change of basis. The total computational cost is then less directly related to $n_0$.) It is therefore highly desirable to keep this number small. An initial choice of $\boldsymbol{I}$ for the basis is commonly used in linear programming teaching texts. However, this choice is normally undesirable as it is likely that few, if any, components of $\boldsymbol{v}$ will still be present in an optimal basic solution and considerable work will be necessary to replace them.

Practical codes usually contain some heuristic algorithm that aims to find a good initial basis quickly. Usually this is a triangular matrix because it is then easy to ensure that it is nonsingular and solving the corresponding sets of equations is also easy. A very sophisticated algorithm is unlikely to compete with the simplex algorithm applied to (1.3) from a poor start. Therefore, a crude algorithm is tolerated as long as it is quick and gives the simplex method a reasonably good start. This is the reason for calling the process "crashing". Unfortunately, the literature appears to be greatly lacking in descriptions of crash algorithms but we describe those that we have found in Section 2. In Section 3, we describe an algorithm that has been in use at Harwell for some ten years; in Section 4, we describe a new heuristic algorithm that is based on first permuting $\boldsymbol{A}$ to block lower triangular form; and in Section 5, we consider alternative ways to find this block form. Numerical results are given in Sections 6 to 8, and concluding comments made in Section 9.

## 2. Existing crash algorithms

A summary of crashing techniques available in 1968 is given by Carstens (1968) in the book by Orchard-Hays (1968), and we have been unable to find a comparable recent summary. For example, in his book on linear programming, Chvátal (1983) mentions exchanging artificial variables for any slack variables that have been introduced to convert inequalities to the equations (1.1a), and this is the only crash algorithm described.

Carstens assumes that a starting set of basic variables is given. Failing other information, it may consist entirely of artificial variables, but often experience with similar problems allows a better choice to be made. If the starting basis $\boldsymbol{B}$ is not equal to $\boldsymbol{I}$, he multiplies (1.1a) by $\boldsymbol{B}^{-1}$ to yield an equivalent problem in which $\boldsymbol{A}$ is replaced by $\boldsymbol{B}^{-1}\boldsymbol{A}$ and $\boldsymbol{b}$ is replaced by $\boldsymbol{B}^{-1}\boldsymbol{b}$. For the new problem, the starting basis is $\boldsymbol{I}$, so there is no loss of generality in considering only the case $\boldsymbol{B} = \boldsymbol{I}$. The aim is to replace artificial columns of the basis by columns of $\boldsymbol{A}$. When column $j$ of $\boldsymbol{A}$ replaces column $i$ of $\boldsymbol{B}$, the entry $a_{ij}$ plays a special role and is called the *pivot.*

A sequence of nonzeros of $\boldsymbol{A}$ is chosen for pivots. Each pivot $a_{ij}$ must lie in a column $j$ that is nonbasic and has zeros in all of the rows of preceding pivots. This leads to a basis that is a symmetric permutation of a lower triangular matrix (the permutation brings the successive pivot rows into positions $1, 2, \ldots$). The fact that it is a permutation of a triangular matrix ensures that it is nonsingular and it is hoped that many artificial variables will be removed.

There remains considerable choice for the pivots and Carstens distinguishes two classes of algorithms, which he calls "GAIN switch on" and "GAIN switch off". With the GAIN switch off, the objective function (1.3a) is ignored and the choice is made on sparsity grounds alone. If column $j$ of $\boldsymbol{A}$ has $c_j$ nonzeros and row $i$ has $r_i$ nonzeros, he mentions the following possibilities:

(a) consider the nonbasic columns in order of increasing $c_j$, and choose the pivot $a_{ij}$ to be a nonzero that minimizes $r_i$;

(b) consider the rows in order of increasing $r_i$, and choose the pivot $a_{ij}$ to be a nonzero that minimizes $c_j$ for $j$ nonbasic;

(c) consider the nonzeros in order of increasing $(r_i - 1)(c_j - 1)$ for $j$ nonbasic.

With the GAIN switch on, a basis change is made only if it leads to an improvement in the objective function (1.3a). Therefore, any column without an advantageous reduced cost (see, for instance, Chvátal, 1983, for the meaning of this terminology) is rejected. Once a column has been selected, the pivot is chosen for the greatest improvement in (1.3a). The columns may be taken sequentially or grouped and the best improvement in the group chosen.

Carstens also suggest hybrid algorithms in which the objective function is used as a tie-breaker for a sparsity choice or a sparsity measure is used as a tie-breaker on function reduction.

Carstens makes no firm recommendations, but prefers the GAIN switch off when the starting basis is totally or mostly artificial and the GAIN switch on when it has

few artificials. He also mentions the possibility of a further "grand cycle", that is using the basis found as a fresh starting basis for another round of crashing, but says that there is rarely any advantage in doing this.

Following conversations with Martin Beale, John Tomlin and Mike Saunders, one of us (Reid) conducted some numerical experiments ten years ago that favoured generating a sparse basis containing as many columns of $\boldsymbol{A}$ as possible but ignoring the objective function (GAIN switch off in Carstens' terminology). This algorithm is explained in the next section. The other algorithms that he tried were significantly slower and frequently produced worse results. Note that in practice $\boldsymbol{b}$ is often sparse and many of the constraints are simple positivity bounds $x_i \geq 0$. Under these circumstances, a very sparse $\boldsymbol{B}$ is likely to be reducible and yield many zero components of $\boldsymbol{x}$, so there may be few infeasibilities.

The package MINOS (Murtagh and Saunders, 1987) contains a crash algorithm that is a variant of Carstens' GAIN-switch-off algorithm and is typical of those of mathematical programming systems of the 1970s. Here, a pivot $a_{ij}$ may be selected because its row contains zeros in all the columns that have so far been chosen as basic or because its column contains zeros in all the rows that have been pivotal. In the former case, the new column is placed at the front of the set of columns so far selected. In the latter case, it is placed behind the set of columns so far selected. Carstens considered only the latter case. In MINOS, a pivot of the former kind is preferred, and ties are otherwise broken by choosing the largest nonzero. The pivots are chosen in three passes:

(1) for $j = 1, 2, \ldots n$, if $x_j$ is free (its bounds are infinite), column $j$ is considered;

(2) for $j = 1, 2, \ldots n$, column $j$ is considered if it has not already been chosen, but is not selected unless a pivot can be found in a row that was not originally an inequality (see the first paragraph of Section 1);

(3) for $j = 1, 2, \ldots n$, column $j$ is considered if it has not already been chosen; and artificials are used to complete the basis.

In addition, MINOS ignores (treats as if zero) any entry that does not satisfy the inequality

$$|a_{ij}| \geq u \max_l |a_{lj}| \tag{2.1}$$

for a preset tolerance $u$ (with default value 0.1). The chosen pivots therefore satisfy (2.1) and the basis may contain small entries outside the (permuted) triangular form.

## 3. The upper-triangular algorithm

The algorithm that Reid developed generates a basis that is upper triangular when permuted to place the pivots on the diagonal in their order of generation, as opposed to lower triangular in Carstens' GAIN-off algorithms. This means that a column that is chosen late is required to have a nonzero in at least one row that has not

yet been pivotal, a less stringent requirement than needing zeros in all the rows that have been pivotal.

The matrix $A$ is first scanned for column singletons and as each is found, it is introduced into the basis and the row containing its entry is flagged. When testing later columns for being singletons, flagged rows are ignored. During the search, the number $k_1$ of nonzeros in the shortest column that is neither empty nor a singleton (excluding flagged rows, of course) is determined. Note that any slack variables introduced into the problem to handle inequalities (see the first paragraph of Section 1) correspond to singleton columns.

If the basis is not completed during the scan, a second search is commenced during which columns of length up to $k_1$ are accepted. If an accepted column is not a singleton, the greatest candidate pivot is taken, artificial variables $v_i$ are introduced for the other candidates and the rows of all of the candidates are flagged. During this scan, the length $k_2$ of the shortest rejected candidate column is recorded.

Such sweeps continue until the basis is complete or until a preset limit on the number of sweeps is reached (default value 10). During sweep $s$, the length $k_s$ of the shortest rejected candidate column is recorded, together with the number of such candidates. Sweep $s+1$ normally uses the threshold $k_s$, but if sweep $s$ finds fewer columns of length $k_s$ than the average needed for the remaining sweeps, sweep $s+1$ uses the threshold $k_s+1$ in order to make rapid progress more certain. Note that if sweep $s$ fails to find any columns, sweep $s+1$ is bound to find one or more columns.

Our experience is that usually three or four sweeps suffice, but that occasionally six or seven are needed. If the limit is reached, the basis is completed with artificials. For the sake of numerical stability, we accept pivots only if they satisfy the relative pivot tolerance (2.1).

## 4. The tearing algorithm

We now describe a new algorithm for attempting to solve problem (1.1). The strategy we adopt is one of breaking the problem into smaller pieces and solving an initial point problem for each piece. We shall assume that we have a good method for solving dense linear programs involving $t$ or fewer equations; we have in mind a typical value of $t = 5$. Call such a method DLP. We shall assume that DLP generates basic optimal solutions, that is, gives optimal solutions and corresponding sets of basic variables. The method works as follows.

We start by permuting the rows and columns of $A$ so that the permuted matrix has the form

$$\begin{pmatrix} A_{11} & & & & \\ A_{21} & A_{22} & & & \\ \vdots & \vdots & \ddots & & \\ A_{r1} & A_{r2} & \cdots & A_{rr} & \\ A_{r+11} & A_{r+12} & \cdots & A_{r+1r} & \mathbf{0} \end{pmatrix}; \tag{4.1}$$

here each matrix $\boldsymbol{A}_{ij}$ is $m_i$ by $n_j$, and the integers $m_i$ and $n_j$ are positive, excepting $m_{r+1}$ and $n_{r+1}$ which may be zero. We consider alternative algorithms for permuting to this form in the next section. The sizes $m_i$ of the blocks are typically very small in practice (see Sections 6 and 7).

We now assume that $\boldsymbol{A}$ has been ordered as in (4.1) and that $\boldsymbol{x}$ and $\boldsymbol{b}$ have been partitioned so that (1.1) may be written as minimizing

$$\boldsymbol{e}^{\mathrm{T}}(\boldsymbol{v}+\boldsymbol{w}) \tag{4.2a}$$

subject to

$$\begin{pmatrix} \boldsymbol{A}_{11} & & & & \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} & & & \\ \vdots & \vdots & \ddots & & \\ \boldsymbol{A}_{r1} & \boldsymbol{A}_{r2} & \cdots & \boldsymbol{A}_{rr} & \\ \boldsymbol{A}_{r+11} & \boldsymbol{A}_{r+12} & \cdots & \boldsymbol{A}_{r+1r} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_r \\ \boldsymbol{x}_{r+1} \end{pmatrix} + \begin{pmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{v}_r \\ \boldsymbol{v}_{r+1} \end{pmatrix} - \begin{pmatrix} \boldsymbol{w}_1 \\ \boldsymbol{w}_2 \\ \vdots \\ \boldsymbol{w}_r \\ \boldsymbol{w}_{r+1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \vdots \\ \boldsymbol{b}_r \\ \boldsymbol{b}_{r+1} \end{pmatrix} \tag{4.2b}$$

and

$$\boldsymbol{b}_{\mathrm{l}i} \leq \boldsymbol{x}_i \leq \boldsymbol{b}_{\mathrm{u}i}, \quad \boldsymbol{v}_i \geq \boldsymbol{0}, \quad \boldsymbol{w}_i \geq \boldsymbol{0}, \qquad i = 1, \ldots, r+1. \tag{4.2c}$$

In its most simple form, our method proceeds in stages, as follows.

We shall assume, for now, that $m_i \leq t$ for all $1 \leq i \leq r$. The first block of (4.2b) and constraints (4.2c) require that $\boldsymbol{x}_1$ satisfies

$$\boldsymbol{A}_{11}\boldsymbol{x}_1 + \boldsymbol{v}_1 - \boldsymbol{w}_1 = \boldsymbol{b}_1, \qquad \boldsymbol{b}_{\mathrm{l}1} \leq \boldsymbol{x}_1 \leq \boldsymbol{b}_{\mathrm{u}1}, \quad \boldsymbol{v}_1 \geq \boldsymbol{0}, \quad \boldsymbol{w}_1 \geq \boldsymbol{0}. \tag{4.3}$$

We want both artificial variables $\boldsymbol{v}_1$ and $\boldsymbol{w}_1$ to be zero. As $m_1$ is no larger than $t$, we may achieve this using DLP to minimize

$$\boldsymbol{e}^{\mathrm{T}}(\boldsymbol{v}_1 + \boldsymbol{w}_1)$$

subject to the constraints (4.3). This is the first stage of the process. As a by-product we would also expect to obtain a set of $m_1$ basic variables, perhaps including some of the artificial variables $\boldsymbol{v}_1$ and $\boldsymbol{w}_1$. Let $\bar{\boldsymbol{x}}_1$, $\bar{\boldsymbol{v}}_1$ and $\bar{\boldsymbol{w}}_1$ be the values of $\boldsymbol{x}_1$, $\boldsymbol{v}_1$ and $\boldsymbol{w}_1$ obtained. (In the case of this first block, $\bar{\boldsymbol{v}}_1$ and $\bar{\boldsymbol{w}}_1$ are zero if (1.1) has a feasible point.)

Moving to the $k$th stage of the method, we assume that we have obtained values $\bar{\boldsymbol{x}}_i$, $\bar{\boldsymbol{v}}_i$ and $\bar{\boldsymbol{w}}_i$ for the variables $\boldsymbol{x}_i$, $\boldsymbol{v}_i$ and $\boldsymbol{w}_i$ for $1 \leq i < k$ and a set of $m_1 + \cdots + m_{k-1}$ basic variables. We then wish to satisfy

$$\boldsymbol{A}_{kk}\boldsymbol{x}_k + \boldsymbol{v}_k - \boldsymbol{w}_k = \boldsymbol{b}_k - \sum_{i=1}^{k-1} \boldsymbol{A}_{ki}\bar{\boldsymbol{x}}_i, \qquad \boldsymbol{b}_{\mathrm{l}k} \leq \boldsymbol{x}_k \leq \boldsymbol{b}_{\mathrm{u}k}, \quad \boldsymbol{v}_k \geq 0, \quad \boldsymbol{w}_k \geq 0, \tag{4.4}$$

with both $\boldsymbol{v}_k$ and $\boldsymbol{w}_k$ being zero—we say that such a solution to (4.4) is a *satisfactory* solution. We attempt to find a satisfactory solution by minimizing

$$\boldsymbol{e}^{\mathrm{T}}(\boldsymbol{v}_k + \boldsymbol{w}_k)$$

subject to the constraints (4.4) using DLP—recall that we have assumed that $m_k \leq t$. There are two possible outcomes. Firstly, we may reduce all of the artificial variables to zero. (The basic set may still possibly include some artificial variables with values of zero.) Secondly, there may be no satisfactory solution to (4.4). Nonetheless, $m_k$ basic variables will be obtained (including some artificial variables). In either case, the values $\bar{x}_k$, $\bar{v}_k$ and $\bar{w}_k$ of $x_k$, $v_k$ and $w_k$ obtained are passed, together with the set of $m_k$ basic variables obtained, to stage $k+1$.

We apply this process for stages $k = 2, 3, \ldots, r$. In the last stage ($k = r+1$) we cannot alter the residuals of the last block of (4.2b) by choice of $x_{r+1}$; therefore we merely pick $x_{r+1}$ to satisfy (4.2c), set the artificial variables

$$\bar{v}_{r+1} = \max\left(\mathbf{0}, b_{r+1} - \sum_{i=1}^{r} A_{r+1,i}\bar{x}_i\right) \quad \text{and} \quad \bar{w}_{r+1} = \max\left(\mathbf{0}, -b_{r+1} + \sum_{i=1}^{r} A_{r+1,i}\bar{x}_i\right) \tag{4.5}$$

and select $m_{r+1}$ of the variables $v_{r+1}$ and $w_{r+1}$ to be basic (including all nonzero components of $v_{r+1}$ and $w_{r+1}$). We thus end up with values for all of the variables, a set of $m$ basic variables and, we hope, most of the artificial variables at zero.

So far, we have assumed that $m_i \leq t$ for all $1 \leq i \leq r$. Our experience (see Section 6) is that this is almost always the case. However, if $m_k > t$, we further partition (4.4) into

$$\begin{aligned} &A_{kk1}x_k + v_{k1} - w_{k1} = b_{k1} - \sum_{i=1}^{k-1} A_{ki1}\bar{x}_i, \quad v_{k1} \geq \mathbf{0}, \quad w_{k1} \geq \mathbf{0}, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad b_{lk} \leq x_k \leq b_{uk}, \\ &A_{kk2}x_k + v_{k2} - w_{k2} = b_{k2} - \sum_{i=1}^{k-1} A_{ki2}\bar{x}_i, \quad v_{k2} \geq \mathbf{0}, \quad w_{k2} \geq \mathbf{0}, \end{aligned} \tag{4.6}$$

where $A_{kk1}$ has $t$ rows and as large a rank as possible. This latter condition can be ensured by reordering the rows of $A_{kk}$ and we assume that this has been done. We may then apply DLP to attempt to find a satisfactory solution to the first set of equations in (4.6). If we let $\bar{x}_k$, $\bar{v}_{k1}$ and $\bar{w}_{k1}$ be the solution obtained, we pass this solution and associated set of basic variables, together with an appropriate basic subset (of cardinality $m_k - t$) of the variables

$$\bar{v}_{k2} = \max\left(\mathbf{0}, b_{k2} - \sum_{i=1}^{k} A_{ki2}\bar{x}_i\right), \qquad \bar{w}_{k2} = \max\left(\mathbf{0}, -b_{k2} + \sum_{i=1}^{k} A_{ki2}\bar{x}_i\right), \tag{4.7}$$

to the next stage. We would, of course, not expect the artificial variables, $v_{k2}$ and $w_{k2}$, in the basis to be zero but are assured at least of a set of $m_k$ basic variables.

In this form of our algorithm, one would suspect that bad choices for the early variables $x_k$ could have unfortunate consequences when one tries to satisfy the later equations. With this in mind, we prefer the following variation.

We assume that we have obtained values $\bar{x}_i$, $\bar{v}_i$ and $\bar{w}_i$ for the variables $x_i$, $v_i$ and $w_i$, for $1 \leq i < k$, and a set of $m_1 + \cdots + m_{k-1}$ basic variables. As before, we first attempt to find a satisfactory solution to (4.4) using DLP—again we have assumed

that $m_k \leq t$. If we find such a solution, we pass it along with the associated set of $m_k$ basic variables (possibly including some artificial variables with values of zero) to the $(k+1)$st stage. If we fail to obtain such a solution, it is likely that values of $\bar{x}_i$, $1 \leq i < k-1$, found in the previous stages are inappropriate. We will, nonetheless, have found a set of $m_k$ basic variables (including some artificial variables). There may be scope for "backtracking" through the set of recently allocated blocks of variables and equations, changing the values of the variables whilst continuing to satisfy the equations. We might then be able to find a satisfactory solution to (4.4) after all. If we work with blocks $j, j+1, \ldots, k$, we need the inequality

$$m_j + m_{j+1} + \cdots + m_k \leq t \tag{4.8}$$

to be satisfied so that we will not consider more equations than DLP can accommodate. We use our linear programming method to attempt to find a satisfactory solution to

$$\begin{pmatrix} \boldsymbol{A}_{jj} & & & \\ \boldsymbol{A}_{j+1j} & \boldsymbol{A}_{j+1j+1} & & \\ \vdots & \vdots & \ddots & \\ \boldsymbol{A}_{kj} & \boldsymbol{A}_{kj+1} & \cdots & \boldsymbol{A}_{kk} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_j \\ \boldsymbol{x}_{j+1} \\ \vdots \\ \boldsymbol{x}_k \end{pmatrix} + \begin{pmatrix} \boldsymbol{v}_j \\ \boldsymbol{v}_{j+1} \\ \vdots \\ \boldsymbol{v}_k \end{pmatrix} - \begin{pmatrix} \boldsymbol{w}_j \\ \boldsymbol{w}_{j+1} \\ \vdots \\ \boldsymbol{w}_k \end{pmatrix} = \begin{pmatrix} \boldsymbol{b}_j \\ \boldsymbol{b}_{j+1} \\ \vdots \\ \boldsymbol{b}_k \end{pmatrix} - \sum_{i=1}^{j-1} \begin{pmatrix} \boldsymbol{A}_{ji} \\ \boldsymbol{A}_{j+1i} \\ \vdots \\ \boldsymbol{A}_{ki} \end{pmatrix} \bar{\boldsymbol{x}}_i \tag{4.9a}$$

and

$$\boldsymbol{b}_{1i} \leq \boldsymbol{x}_i \leq \boldsymbol{b}_{ui}, \quad \boldsymbol{v}_i \geq \boldsymbol{0}, \quad \boldsymbol{w}_i \geq \boldsymbol{0}, \qquad i = j, \ldots, k. \tag{4.9b}$$

Let $\bar{\boldsymbol{x}}_i$, $\bar{\boldsymbol{v}}_i$ and $\bar{\boldsymbol{w}}_i$, for $j \leq i \leq k$, be the values obtained. Then these new values replace the existing values, and are passed to the next stage. Notice, however, that we do not necessarily keep the property of having $m_i$ basic variables among $\boldsymbol{x}_i$, $\boldsymbol{v}_i$ and $\boldsymbol{w}_i$ (some blocks may gain basic variables at the expense of others). Therefore, as well as satisfying (4.8), we need to choose $j$ so that the condition,

$$\begin{aligned} &\text{number of basic variables in } \boldsymbol{x}_i, \boldsymbol{v}_i \text{ and } \boldsymbol{w}_i, \text{ for } i = j, j+1, \ldots, k, \\ &\quad \text{equals} \quad m_j + m_{j+1} + \cdots + m_k, \end{aligned} \tag{4.10}$$

is true, which may reduce the amount of backtracking possible. An important feature is that we do *not* alter the attempted solution to the first $j-1$ blocks of equations.

## 5. Permuting to block lower-triangular form

We have considered two alternative algorithms, along with several minor variations, for permuting to the block lower-triangular form (4.1). The $P^5$ algorithm of Erisman, Grimes, Lewis and Poole (1985) (see also Duff, Erisman and Reid, 1986, Chapter 8) is available in the Harwell Subroutine Library (Harwell Subroutine Library, 1988) as MC33. The algorithm is designed to permute a square matrix to bordered block

triangular form, but it may readily be adapted to the rectangular case and the border columns may be permuted forward to give the required form. It has the additional properties that the diagonal blocks $\boldsymbol{A}_{ii}$, $i = 1, 2, \ldots r$, are full and have at least as many columns as rows ($m_i \leq n_i$). As a consequence, no set of rows from the first $r$ blocks can be structurally dependent, that is linearly dependent even if arbitrary changes are made to the values of the nonzero entries. Any null rows must be included in block $r+1$ and any set of structurally dependent rows must have some members in block $r+1$. Our experience is that the rows of this block are usually all of these kinds. The sizes $m_i$ of the blocks are typically very small in practice (see Section 6).

The $P^5$ algorithm chooses the columns one at a time to maximize the number of entries in rows of minimum row count, where the row count is the number of nonzeros in the row when columns that have been chosen are excluded. When several columns are best from this point of view, a column with greatest column count is chosen (unless the number of entries in rows with minimum row count is one). This biases the later columns towards sparseness, which is seen as desirable as far as the success of the later stages of the algorithm are concerned. However, our backtracking algorithm (last paragraph of Section 4) might be expected to be successful if there is little dependence on the values of the earlier variables ($\bar{x}_i$, $i = 1, \ldots, j-1$, in equation (4.9a)), which we are not free to change. Therefore, we experimented with a variation of the $P^5$ algorithm in which ties are broken by minimizing rather than maximizing the column count and giving no special attention to the case where the number of entries in rows with minimum row count is one, but unfortunately found that on the whole this variation gave poorer results.

Another possibility for helping the backtracking algorithm is to aim for the block bidiagonal form

$$\begin{pmatrix} \bar{\boldsymbol{A}}_{11} & & & & \\ \bar{\boldsymbol{A}}_{21} & \bar{\boldsymbol{A}}_{22} & & & \\ & \bar{\boldsymbol{A}}_{32} & \bar{\boldsymbol{A}}_{33} & & \\ & & & \ddots & \\ & & & & \bar{\boldsymbol{A}}_{ss} \end{pmatrix}, \tag{5.1}$$

which corresponds to the normal matrix $\boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}$ being block tridiagonal. This is the foundation of our alternative algorithm, which we call the normal matrix method. There are established techniques for permuting a symmetric matrix to block tridiagonal form that are based on examining the associated graph, finding a pseudo-diameter, and constructing a rooted level structure based on an end point of a pseudo-diameter (see, for example, Gibbs, Poole and Stockmeyer, 1976). We have used the variant of Sloan (1986), whose code is available as MC40 in the Harwell Subroutine Library. Several strategies have been proposed for ordering within the level sets (blocks of the block tridiagonal form), which is important if a small local bandwidth is to be obtained. Again, we have used Sloan's code, but we have adjusted

his weights so that the block tridiagonal form is preserved. Each node in a level set is then ordered in turn to minimize the current front size (number of nodes that are neighbours of ordered nodes but have not yet themselves been ordered). This is the variant proposed by Gibbs (1976) and called the Gibbs–King algorithm by Lewis (1982). Sloan (1986) allows departure from the form (5.1) that the rooted level structure suggests and chooses each variable to minimize a weighted sum of the front size and the level set number. We tried this variant, but found that its results were generally inferior for our application.

This analysis of the normal matrix provides us with an ordering for the rows of $A$. If there are any null rows, they are permuted to the end and included in the last block.

Given an ordering for the rows of $A$, the form (4.1) may be constructed by column permutations as follows. Permute the columns with entries in row 1 to the front; follow these with columns with the property of having a zero in row 1 and an entry in row 2; then columns with the property of having zeros in rows 1 and 2 and an entry in row 3; and so on. If the normal matrix $AA^{\mathrm{T}}$ is block tridiagonal, the permuted $A$ will have the form (5.1) with identical block row sizes. However, these sizes are usually quite large and in practice the column permutation will break each diagonal block $\bar{A}_{ii}$ in (5.1) into a block lower triangular form with small blocks. Thus overall we have the form

$$\begin{pmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ \vdots & \vdots & \ddots & \\ A_{r1} & A_{r2} & \cdots & A_{rr} \end{pmatrix}, \tag{5.2}$$

with many zero blocks in the left-hand lower corner. Note that this is comparable to the form (4.1), but that the diagonal blocks are no longer full and may have more rows than columns. Therefore any block may include a set of structurally dependent rows and the last block may have some null rows. When the Sloan variant of the ordering algorithm for the normal matrix is used, the large blocks are not necessarily preserved, but there should be more small zero blocks since a small local bandwidth of the normal matrix corresponds to the sparsity pattern of the row of $A$ intersecting the sparsity patterns of only nearby rows.

To illustrate the two alternative algorithms, we show the sparsity pattern of the original matrix and the two reordered matrices for the problem Forplan in Figures 1–3 and for the problem Capri in Figures 4–6. These problems are part of the Netlib set (Gay, 1985). The patterns illustrated are typical; for $P^5$, the nonzeros tend to "bow" under the "diagonal", while the reverse is true for the normal matrix method. Notice also that the normal equations method does yield some large zero blocks in the lower left-hand part of the matrix.

It has been pointed out to us by Mike Saunders that this procedure will not be useful if $A$ has a dense column since $AA^{\mathrm{T}}$ is full in this case. He suggests placing the full columns at the front and applying our procedure to the rest.

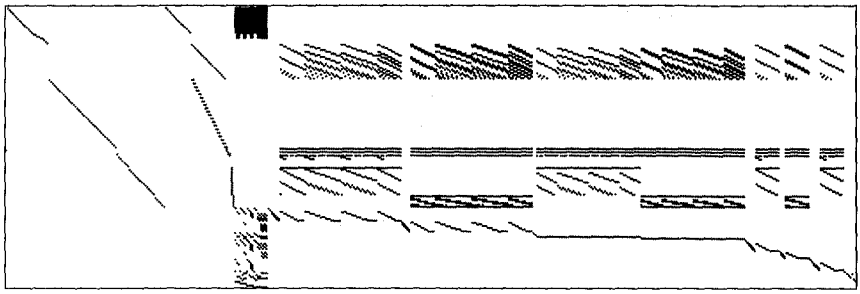

Fig. 1. The sparsity pattern of the problem Forplan before reordering.

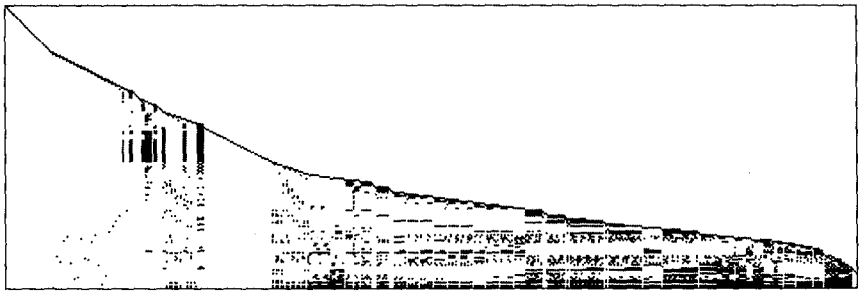

Fig. 2. The sparsity pattern of the problem Forplan after $P^5$ reordering.

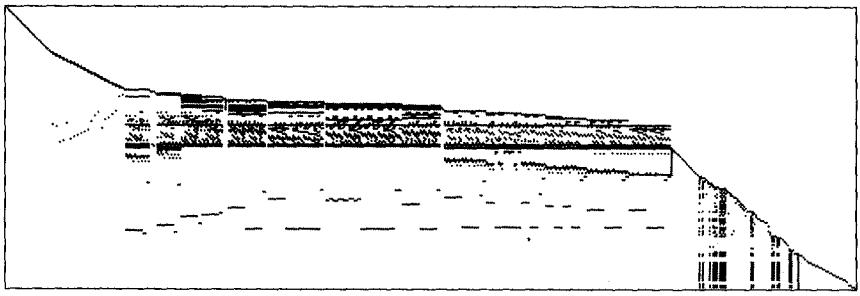

Fig. 3. The sparsity pattern of the problem Forplan after normal matrix reordering.

## 6. Numerical experiments

In this section we describe the results of testing the method suggested in the previous sections on a variety of real-world linear programming problems. All of the problems are available in MPS format (see, for example, Murtagh, 1981, Chapter 9) and were converted, if necessary, to the generic form

$$\begin{aligned} &\text{minimize} \quad \boldsymbol{c}^T\boldsymbol{x} \\ &\text{subject to} \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \ \boldsymbol{b}_l \leq \boldsymbol{x} \leq \boldsymbol{b}_u, \end{aligned} \tag{6.1}$$

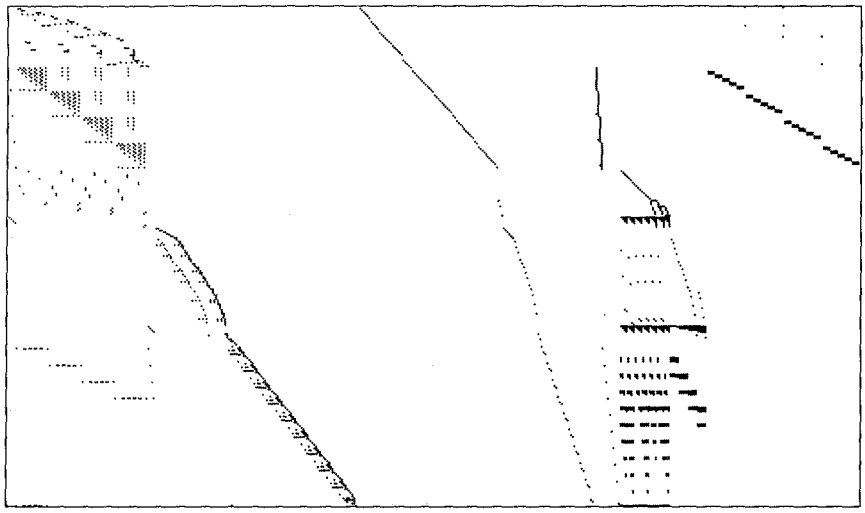

Fig. 4. The sparsity pattern of the problem Capri before reordering.

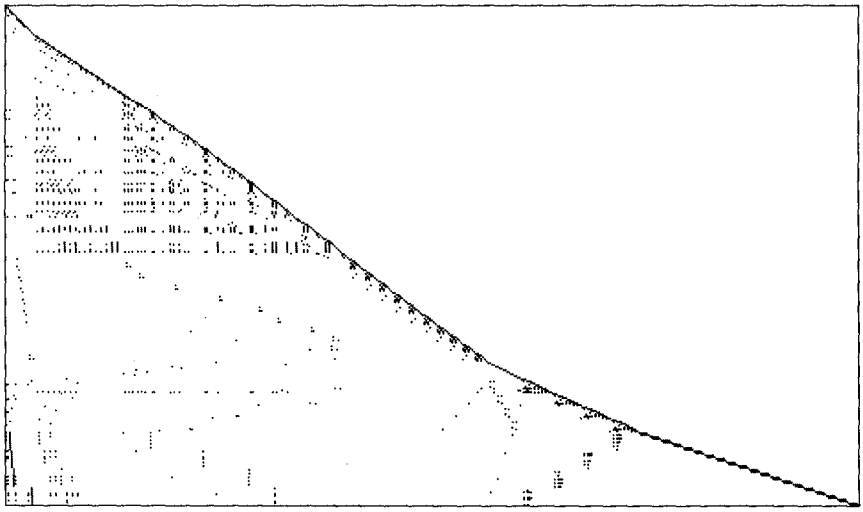

Fig. 5. The sparsity pattern of the problem Capri after $P^5$ reordering.

by adding slack variables to any inequality constraints. Our aim is to find a feasible solution to the set of constraints for the given problem and we judge the success of a crash algorithm by the number of infeasibilities (nonzero artificial variables) since it is our experience that the computer time taken by the simplex method to reach feasibility is usually approximately proportional to the number of infeasibilities. We give details of two sets of linear programming test problems, those collected at Harwell and the extensive set available from Netlib (Dongarra and Grosse, 1987) as collected by Gay (1985). The characteristics of the Harwell

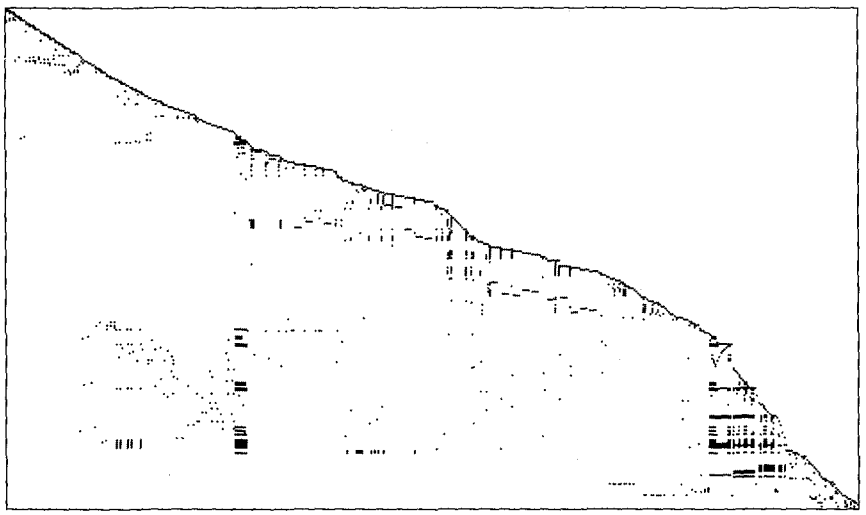

Fig. 6. The sparsity pattern of the problem Capri after normal matrix reordering.

problems are shown in Table 6.1. Note that this data confirms our Section 4 contention that when the $P^5$ algorithm is used, each block row size $m_i$ is typically very small.

All of our computation was performed on the IBM3084Q at Harwell. Our codes were written in Fortran 77 and compiled by level 1.4.1 of the VS compiler with OPT = 3.

The problems were considered both as defined from their raw data in the form (6.1) (referred to as *unscaled*) and also after row and column scalings have been

Table 6.1

The Harwell test problem attributes

| Problem name | No. rows ($m$) | No. cols ($n$) | No. nonzeros | $P^5$ block sizes | |
|---|---|---|---|---|---|
| | | | | $m_1, \ldots, m_r$ | $m_{r+1}$ |
| Sc50a | 50 | 78 | 160 | 50 (1) | 0 |
| Sc50b | 50 | 78 | 148 | 50 (1) | 0 |
| Blend | 74 | 114 | 522 | 74 (1) | 0 |
| Sc105 | 105 | 163 | 340 | 105 (1) | 0 |
| Boeing 2 | 166 | 305 | 1358 | 165 (1) | 0 |
| Boeing 1 | 351 | 726 | 3827 | 351 (1) | 0 |
| Stair | 356 | 532 | 3813 | 344 (1), 6 (2) | 0 |
| Finnis | 497 | 1019 | 2542 | 497 (1) | 0 |
| Powell | 524 | 1028 | 6401 | 511 (1), 6 (2) | 1 |
| Shell | 536 | 1527 | 3058 | 528 (1), 4 (2) | 0 |
| Perold | 625 | 1442 | 5962 | 579 (1), 23 (2) | 0 |
| BP | 821 | 1876 | 10705 | 810 (1), 2 (2), 1 (3) | 4 |
| GUB | 929 | 3333 | 10022 | 929 (1) | 0 |

Legend: $k(l)$ means $k$ blocks of $l$ rows.

applied to the matrix $\boldsymbol{A}$ (*scaled*). Given the raw data for the problems, the rows and columns of the matrix $\boldsymbol{A}$ are rescaled using the scheme of Curtis and Reid (1972), implemented as subroutine MC19 in the Harwell Subroutine Library. To be specific, diagonal matrices $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ are found to convert (6.1) to the equivalent problem

$$\begin{aligned} &\text{minimize} \quad \bar{\boldsymbol{c}}^{\mathrm{T}}\bar{\boldsymbol{x}} \\ &\text{subject to} \quad \bar{\boldsymbol{A}}\bar{\boldsymbol{x}} = \bar{\boldsymbol{b}},\ \bar{\boldsymbol{b}}_{\mathrm{l}} \leqslant \bar{\boldsymbol{x}} \leqslant \bar{\boldsymbol{b}}_{\mathrm{u}}, \end{aligned} \tag{6.2}$$

where $\bar{\boldsymbol{A}} = \boldsymbol{D}_1\boldsymbol{A}\boldsymbol{D}_2$, $\bar{\boldsymbol{b}} = \boldsymbol{D}_1\boldsymbol{b}$, $\bar{\boldsymbol{c}} = \boldsymbol{D}_2\boldsymbol{c}$, $\bar{\boldsymbol{b}}_{\mathrm{l}} = \boldsymbol{D}_2^{-1}\boldsymbol{b}_{\mathrm{l}}$ and $\bar{\boldsymbol{b}}_{\mathrm{u}} = \boldsymbol{D}_2^{-1}\boldsymbol{b}_{\mathrm{u}}$. The scheme is intended to produce a scaled matrix $\bar{\boldsymbol{A}}$ whose nonzero entries are approximately equal in magnitude and does this by minimizing the sum of squares of the logarithms of the absolute values of the nonzero entries.

We applied the algorithms of Sections 3, 4 and 5 to the unscaled and scaled test problems. As far as the relative performances of the crash algorithms are concerned, we found little difference between the scaled and unscaled cases, and have therefore chosen to present only the unscaled results here. This should not be interpreted as a recommendation against scaling, which was frequently (but not always) advantageous to the efficiency of the complete solution.

In Table 6.2 we show the number of infeasibilities after the application of our $P^5$ tearing algorithm, both without backtracking and with $t = 5$, 10 and 50 backtracking. We concluded from this data that some backtracking is certainly worthwhile, but that a large value of $t$ does not give sufficient gains to justify its extra expense. This data also suggests that any value of $t$ between 5 and 10 would be suitable for general use and we have chosen to recommend 5.

Table 6.2

Harwell test problem results for the $P^5$ tearing algorithm without and with backtracking

| Problem name | No. rows | Without | | $t = 5$ | | $t = 10$ | | $t = 50$ | |
|---|---|---|---|---|---|---|---|---|---|
| | | No. infeas. | Time (secs) | No. infeas. | Time (secs) | No. infeas. | Time (secs) | No. infeas. | Time (secs) |
| Sc50a | 50 | 0 | 0.03 | 0 | 0.03 | 0 | 0.02 | 0 | 0.03 |
| Sc50b | 50 | 0 | 0.02 | 0 | 0.03 | 0 | 0.03 | 0 | 0.03 |
| Blend | 74 | 0 | 0.05 | 0 | 0.06 | 0 | 0.06 | 0 | 0.07 |
| Sc105 | 105 | 0 | 0.05 | 0 | 0.05 | 0 | 0.06 | 0 | 0.09 |
| Boeing2 | 166 | 24 | 0.19 | 17 | 0.21 | 15 | 0.24 | 5 | 0.69 |
| Boeing1 | 351 | 58 | 0.79 | 13 | 1.13 | 5 | 1.21 | 3 | 2.32 |
| Stair | 356 | 83 | 0.40 | 77 | 0.46 | 76 | 0.57 | 68 | 2.09 |
| Finnis | 497 | 41 | 0.57 | 31 | 0.63 | 31 | 0.72 | 22 | 14.78 |
| Powell | 524 | 61 | 1.21 | 33 | 1.24 | 15 | 1.39 | 8 | 46.64 |
| Shell | 536 | 53 | 0.53 | 31 | 0.59 | 25 | 0.70 | 19 | 2.05 |
| Perold | 625 | 84 | 0.72 | 75 | 5.79 | 63 | 7.83 | 59 | 64.52 |
| BP | 821 | 139 | 1.70 | 102 | 1.86 | 97 | 2.13 | 79 | 30.72 |
| GUB | 929 | 5 | 5.68 | 4 | 5.57 | 4 | 5.75 | 5 | 6.92 |

Table 6.3
Test problem results using $P^5$ tearing with $t = 5$, Reid, and MINOS algorithms

| Problem name | No. rows | $P^5$ tearing | | Reid | | | MINOS |
|---|---|---|---|---|---|---|---|
| | | No. infeas. | Time (secs) | No. infeas. | Time (secs) | Time to match tearing | No. infeas. |
| Sc50a | 50 | 0 | 0.03 | 0 | 0.04 | 0.04 | |
| Sc50b | 50 | 0 | 0.03 | 0 | 0.04 | 0.04 | |
| Blend | 74 | 0 | 0.06 | 0 | 0.08 | 0.08 | |
| Sc105 | 105 | 0 | 0.05 | 0 | 0.07 | 0.07 | |
| Boeing2 | 166 | 17 | 0.21 | 34 | 0.17 | 0.76 | |
| Boeing1 | 351 | 13 | 1.13 | 111 | 0.55 | 9.11 | |
| Stair | 356 | 77 | 0.46 | 117 | 0.27 | 1.78 | 139 |
| Finnis | 497 | 31 | 0.63 | 92 | 0.86 | 5.31 | |
| Powell | 524 | 33 | 1.24 | 65 | 0.58 | 1.37 | 131 |
| Shell | 536 | 31 | 0.59 | 43 | 0.82 | 1.00 | 12 |
| Perold | 625 | 75 | 5.79 | 156 | 1.94 | 49.85 | |
| BP | 821 | 102 | 1.86 | 189 | 1.67 | 11.21 | 229 |
| GUB | 929 | 4 | 5.57 | 6 | 2.99 | 4.91 | 20 |

We show the number of infeasibilities after the application of the crash algorithm of Reid (Section 3) in Table 6.3. For these problems, it always produced more infeasibilities than the tearing algorithm with $t = 5$, though on one problem it produced fewer than the tearing algorithm without backtracking. To measure the difference, we show the time taken for the Harwell LP code LA04, starting from the basis constructed by Reid's crash, to reach the number of infeasibilities obtained for the tearing method with $t = 5$. It can be seen that the gain can be quite worthwhile, though in one case (GUB), even when the time for simplex iteration was included, the Reid algorithm was faster to the same number of infeasibilities.

For five of the problems, we also show in Table 6.3 the number of infeasibilities that are obtained by the MINOS crash algorithm that we described at the end of Section 2.

In Tables 6.4 to 6.6 we give details of further tests performed on the Netlib linear programming test set (Gay, 1985), excluding the two largest cases (80bau3b and Pilot). We note that the Netlib problems Czprob, Fffff800, Shell, Stair and 25fv47 appear in the Harwell test set under the names GUB, Powell, Shell, Stair and BP respectively.

Table 6.4 confirms our earlier experience of small blocks and Table 6.5 reinforces our earlier conclusion in favour of the choice of $t = 5$ for the amount of backtracking in the tearing algorithm. However, the comparison with Reid's algorithm is not so straightforward here. In Table 6.6 we see that in 10 of the cases (out of 50), Reid's algorithm produced fewer infeasibilities and for these we show the time for the tearing method to match the Reid result by simplex iterations. In all these cases, the time taken by the Reid algorithm is the lesser. Being the simpler algorithm, it

Table 6.4
Additional Netlib test problem attributes

| Problem name | No. rows ($m$) | No. cols ($n$) | No. nonzeros | $P^S$ block sizes | |
|---|---|---|---|---|---|
| | | | | $m_1, \ldots, m_r$ | $m_{r+1}$ |
| Adlittle | 56 | 138 | 424 | 54 (1), 1 (2) | 0 |
| Afiro | 27 | 51 | 102 | 27 (1) | 0 |
| Bandm | 305 | 472 | 2494 | 291 (1), 7 (2) | 0 |
| Beaconfd | 173 | 295 | 3408 | 139 (1), 3 (2), 3 (3), 1 (4) | 15 |
| Bore3d | 233 | 333 | 1446 | 213 (1), 4 (1), 1 (3), 1 (9) | 0 |
| Brandy | 220 | 303 | 2202 | 175 (1), 5 (2), 1 (3) | 32 |
| Capri | 271 | 466 | 1864 | 271 (1) | 0 |
| Etamacro | 400 | 734 | 2188 | 397 (1), 1 (2) | 0 |
| E226 | 223 | 472 | 2768 | 211 (1), 2 (2), 1 (5) | 3 |
| Forplan | 161 | 489 | 4565 | 147 (1), 3 (2) | 8 |
| Ganges | 1309 | 1706 | 6937 | 1261 (1), 7 (2) | 34 |
| Gfrd-pnc | 616 | 1160 | 2445 | 573 (1), 21 (2) | 1 |
| Grow7 | 140 | 301 | 2612 | 140 (1) | 0 |
| Grow15 | 300 | 645 | 5620 | 300 (1) | 0 |
| Grow22 | 440 | 946 | 8252 | 440 (1) | 0 |
| Israel | 174 | 316 | 2443 | 174 (1) | 0 |
| Nesm | 662 | 2930 | 13260 | 662 (1) | 0 |
| Pilot.ja | 940 | 1956 | 12100 | 820 (1), 52 (2) | 16 |
| Pilotnov | 975 | 2242 | 12460 | 809 (1), 68 (2), 2 (3) | 24 |
| Pilot.we | 722 | 2850 | 9001 | 569 (1), 66 (2), 7 (3) | 0 |
| Pilot4 | 410 | 1093 | 5164 | 386 (1), 12 (2) | 0 |
| Recipe | 91 | 180 | 654 | 88 (1) | 3 |
| Scagr7 | 129 | 185 | 465 | 129 (1) | 0 |
| Scagr25 | 471 | 671 | 1725 | 471 (1) | 0 |
| Scfxm1 | 330 | 600 | 2732 | 300 (1), 8 (2), 2 (3) | 8 |
| Scfxm2 | 660 | 1200 | 5469 | 601 (1), 16 (2), 4 (3) | 15 |
| Scfxm3 | 990 | 1800 | 8206 | 902 (1), 24 (2), 6 (3) | 22 |
| Scorpion | 388 | 466 | 1534 | 248 (1), 12 (2), 12 (3), 6 (4) | 56 |
| Scrs8 | 490 | 1275 | 3288 | 448 (1), 14 (3) | 0 |
| Scsd1 | 77 | 760 | 2388 | 73 (1), 2 (2) | 0 |
| Scsd6 | 147 | 1350 | 4316 | 145 (1), 1 (2) | 0 |
| Scsd8 | 397 | 2750 | 8584 | 393 (1), 2 (2) | 0 |
| Sctap1 | 300 | 660 | 1872 | 300 (1) | 0 |
| Sctap2 | 1090 | 2500 | 7334 | 1090 (1) | 0 |
| Sctap3 | 1480 | 3340 | 9734 | 1480 (1) | 0 |
| Sc205 | 205 | 317 | 665 | 205 (1) | 0 |
| Seba | 515 | 1036 | 4360 | 515 (1) | 0 |
| Share1b | 117 | 253 | 1179 | 82 (1), 8 (2), 5 (3), 1 (4) | 0 |
| Share2b | 96 | 162 | 777 | 96 (1) | 0 |
| Ship04l | 402 | 2166 | 6380 | 352 (1), 4 (2) | 42 |
| Ship04s | 402 | 1506 | 4400 | 352 (1), 4 (2) | 42 |
| Ship08l | 778 | 4363 | 12882 | 696 (1), 8 (2) | 66 |
| Ship08s | 778 | 2467 | 7194 | 696 (1), 8 (2) | 66 |
| Ship12l | 1151 | 5533 | 16276 | 1028 (1), 7 (2) | 89 |
| Ship12s | 1151 | 2869 | 8284 | 1028 (1), 7 (2) | 89 |
| Sierra | 1227 | 2735 | 8001 | 1117 (1), 55 (2) | 0 |
| Standata | 359 | 1258 | 3173 | 359 (1) | 0 |
| Standgub | 361 | 1366 | 3281 | 360 (1) | 1 |
| Standmps | 467 | 1258 | 3821 | 455 (1) | 12 |
| Vtp.base | 198 | 328 | 944 | 198 (1) | 0 |

Legend: $k(l)$ means $k$ blocks of $l$ rows.

Table 6.5
Additional test problem results for the $P^5$ tearing algorithm without and with backtracking

| Problem name | No. rows | Without | | $t = 5$ | | $t = 10$ | | $t = 50$ | |
|---|---|---|---|---|---|---|---|---|---|
| | | No. infeas. | time (secs) | No. infeas. | time (secs) | No. infeas. | time (secs) | No. infeas. | time (secs) |
| Adlittle | 56 | 3 | 0.04 | 2 | 0.05 | 2 | 0.05 | 0 | 0.13 |
| Afiro | 27 | 0 | 0.01 | 0 | 0.01 | 0 | 0.02 | 0 | 0.02 |
| Bandm | 305 | 41 | 0.31 | 31 | 0.84 | 30 | 1.04 | 15 | 1.08 |
| Beaconfd | 173 | 28 | 0.61 | 28 | 0.62 | 28 | 0.63 | 22 | 0.83 |
| Bore3d | 233 | 12 | 0.19 | 10 | 0.20 | 22 | 0.24 | 18 | 0.63 |
| Brandy | 220 | 28 | 0.31 | 22 | 0.33 | 17 | 0.37 | 8 | 0.83 |
| Capri | 271 | 70 | 0.22 | 27 | 0.27 | 23 | 0.36 | 13 | 1.43 |
| Etamacro | 400 | 45 | 0.34 | 37 | 0.39 | 37 | 0.45 | 32 | 1.37 |
| E226 | 223 | 21 | 0.43 | 16 | 0.44 | 14 | 0.47 | 10 | 0.98 |
| Forplan | 161 | 35 | 1.02 | 31 | 1.25 | 24 | 1.35 | 15 | 1.97 |
| Ganges | 1309 | 181 | 1.17 | 146 | 1.13 | 132 | 1.31 | 116 | 3.78 |
| Gfrd-pnc | 616 | 12 | 0.36 | 12 | 0.36 | 12 | 0.40 | 7 | 0.79 |
| Grow7 | 140 | 0 | 0.22 | 0 | 0.21 | 0 | 0.21 | 0 | 0.25 |
| Grow15 | 300 | 0 | 0.46 | 0 | 0.46 | 0 | 0.47 | 0 | 0.56 |
| Grow22 | 440 | 0 | 0.68 | 0 | 0.68 | 0 | 0.70 | 0 | 0.85 |
| Israel | 174 | 16 | 0.34 | 3 | 0.34 | 1 | 0.37 | 0 | 0.66 |
| Nesm | 662 | 123 | 2.03 | 107 | 2.15 | 92 | 2.62 | 65 | 8.49 |
| Pilot.ja | 940 | 144 | 2.41 | 113 | 2.15 | 108 | 2.97 | 94 | 12.54 |
| Pilotnov | 975 | 179 | 2.45 | 165 | 6.12 | 161 | 12.82 | 156 | 8.54 |
| Pilot.we | 722 | 116 | 1.43 | 104 | 1.54 | 104 | 2.02 | 100 | 15.42 |
| Pilot4 | 410 | 33 | 0.70 | 29 | 0.73 | 27 | 0.79 | 25 | 1.67 |
| Recipe | 91 | 0 | 0.06 | 0 | 0.06 | 0 | 0.07 | 0 | 0.09 |
| Scagr7 | 129 | 19 | 0.06 | 6 | 0.08 | 6 | 0.11 | 4 | 0.29 |
| Scagr25 | 471 | 73 | 0.23 | 24 | 0.31 | 24 | 0.42 | 24 | 2.09 |
| Scfxm1 | 330 | 51 | 0.32 | 46 | 0.35 | 46 | 0.42 | 37 | 1.30 |
| Scfxm2 | 660 | 103 | 0.71 | 95 | 0.78 | 91 | 2.85 | 77 | 2.98 |
| Scfxm3 | 990 | 153 | 1.10 | 143 | 1.25 | 137 | 1.44 | 116 | 4.35 |
| Scorpion | 388 | 80 | 0.18 | 31 | 0.23 | 32 | 0.28 | 34 | 1.37 |
| Scrs8 | 490 | 16 | 0.51 | 14 | 0.54 | 14 | 0.67 | 14 | 1.99 |
| Scsd1 | 77 | 0 | 0.42 | 0 | 0.42 | 0 | 0.43 | 0 | 0.45 |
| Scsd6 | 147 | 0 | 0.73 | 0 | 0.74 | 0 | 0.74 | 0 | 0.81 |
| Scsd8 | 397 | 0 | 1.26 | 0 | 1.24 | 0 | 1.26 | 0 | 1.45 |
| Sctap1 | 300 | 32 | 0.34 | 7 | 0.36 | 7 | 0.42 | 7 | 1.16 |
| Sctap2 | 1090 | 51 | 2.61 | 7 | 2.66 | 7 | 2.76 | 1 | 4.28 |
| Sctap3 | 1480 | 62 | 3.76 | 3 | 3.80 | 3 | 3.98 | 0 | 5.76 |
| Sc205 | 205 | 0 | 0.11 | 0 | 0.11 | 0 | 0.12 | 0 | 0.18 |
| Seba | 515 | 2 | 0.75 | 2 | 0.74 | 2 | 0.77 | 2 | 0.98 |
| Share1b | 117 | 29 | 0.11 | 28 | 0.13 | 23 | 0.16 | 11 | 0.48 |
| Share2b | 96 | 12 | 0.07 | 10 | 0.09 | 8 | 0.11 | 8 | 0.98 |
| Ship04l | 402 | 5 | 1.73 | 5 | 1.73 | 5 | 1.78 | 5 | 2.09 |
| Ship04s | 402 | 8 | 0.98 | 8 | 0.97 | 8 | 1.00 | 8 | 1.28 |
| Ship08l | 778 | 7 | 4.68 | 7 | 4.67 | 7 | 4.69 | 7 | 5.21 |
| Ship08s | 778 | 12 | 1.72 | 12 | 1.76 | 12 | 1.81 | 12 | 2.30 |
| Ship12l | 1151 | 14 | 6.25 | 14 | 6.24 | 14 | 6.26 | 11 | 7.14 |
| Ship12s | 1151 | 15 | 2.56 | 15 | 2.55 | 15 | 2.59 | 12 | 3.24 |
| Sierra | 1227 | 163 | 1.41 | 138 | 1.57 | 107 | 1.76 | 84 | 5.37 |
| Standata | 359 | 7 | 2.22 | 6 | 2.22 | 6 | 2.22 | 6 | 2.67 |
| Standgub | 361 | 7 | 2.24 | 6 | 2.24 | 6 | 2.26 | 6 | 2.71 |
| Standmps | 467 | 31 | 2.34 | 30 | 2.34 | 30 | 2.48 | 17 | 4.31 |
| Vtp.base | 198 | 17 | 0.13 | 12 | 0.15 | 12 | 0.17 | 10 | 0.51 |

Table 6.6
Additional test problems using $P^5$ tearing with $t = 5$, Reid, and MINOS algorithms

| Problem name | No. rows | $P^5$ tearing | | | Reid | | | MINOS |
|---|---|---|---|---|---|---|---|---|
| | | No. infeas. | Time (secs) | Time to match Reid | No infeas. | Time (secs) | Time to match tearing | No. infeas. |
| Adlittle | 56 | 2 | 0.05 | | 8 | 0.04 | 0.09 | 8 |
| Afiro | 27 | 0 | 0.01 | | 0 | 0.02 | 0.04 | 1 |
| Bandm | 305 | 31 | 0.84 | | 69 | 0.19 | 1.32 | 78 |
| Beaconfd | 173 | 28 | 0.62 | 0.83 | 24 | 0.12 | | 30 |
| Bore3d | 233 | 10 | 0.20 | 0.37 | 8 | 0.12 | | 10 |
| Brandy | 220 | 22 | 0.33 | | 52 | 0.16 | 0.64 | 68 |
| Capri | 271 | 27 | 0.27 | | 61 | 0.18 | 0.79 | 76 |
| Etamacro | 400 | 37 | 0.39 | | 50 | 0.28 | 0.85 | 67 |
| E226 | 223 | 16 | 0.44 | | 31 | 0.17 | 0.47 | 38 |
| Forplan | 161 | 31 | 1.25 | 2.35 | 13 | 0.21 | | |
| Ganges | 1309 | 146 | 1.13 | 6.50 | 0 | 1.03 | | 204 |
| Gfrd-pnc | 616 | 12 | 0.36 | 1.81 | 2 | 0.66 | | 2 |
| Grow7 | 140 | 0 | 0.21 | | 0 | 0.16 | 0.16 | 0 |
| Grow15 | 300 | 0 | 0.46 | | 0 | 0.43 | 0.43 | 0 |
| Grow22 | 440 | 0 | 0.68 | | 0 | 0.87 | 0.87 | 0 |
| Israel | 174 | 3 | 0.34 | | 7 | 0.09 | 0.12 | 8 |
| Nesm | 662 | 107 | 2.15 | | 436 | 2.16 | 44.77 | 410 |
| Pilot.ja | 940 | 113 | 2.15 | | 129 | 1.84 | 2.73 | 314 |
| Pilotnov | 975 | 165 | 6.12 | 11.03 | 133 | 2.29 | | |
| Pilot.we | 722 | 104 | 1.54 | 9.92 | 69 | 2.36 | | 131 |
| Pilot4 | 410 | 29 | 0.73 | | 80 | 0.55 | 2.20 | 92 |
| Recipe | 91 | 0 | 0.06 | | 20 | 0.05 | 0.11 | 21 |
| Scagr7 | 129 | 6 | 0.08 | | 14 | 0.05 | 0.18 | 24 |
| Scagr25 | 471 | 24 | 0.31 | | 32 | 0.29 | 1.86 | 78 |
| Scfxm1 | 330 | 46 | 0.35 | | 59 | 0.23 | 0.52 | 58 |
| Scfxm2 | 660 | 95 | 0.78 | | 118 | 0.69 | 1.18 | 122 |
| Scfxm3 | 990 | 143 | 1.25 | | 177 | 1.35 | 2.36 | 186 |
| Scorpion | 388 | 31 | 0.23 | | 52 | 0.17 | 0.37 | 37 |
| Scrs8 | 490 | 14 | 0.54 | | 33 | 0.61 | 1.09 | 38 |
| Scsd1 | 77 | 0 | 0.42 | | 7 | 0.19 | 0.35 | 5 |
| Scsd6 | 147 | 0 | 0.74 | | 14 | 0.42 | 0.79 | 34 |
| Scsd8 | 397 | 0 | 1.24 | | 17 | 2.26 | 3.15 | 103 |
| Sctap1 | 300 | 7 | 0.36 | | 33 | 0.19 | 0.86 | 51 |
| Sctap2 | 1090 | 7 | 2.66 | | 51 | 2.02 | 6.16 | 72 |
| Sctap3 | 1480 | 3 | 3.80 | | 61 | 3.53 | 11.54 | 86 |
| Sc205 | 205 | 0 | 0.11 | | 0 | 0.07 | 0.07 | 0 |
| Seba | 515 | 2 | 0.74 | | 8 | 0.44 | 5.77 | 133 |
| Share1b | 117 | 28 | 0.13 | | 31 | 0.08 | 0.11 | 73 |
| Share2b | 96 | 10 | 0.09 | 0.19 | 4 | 0.04 | | 4 |
| Ship04l | 402 | 5 | 1.73 | 4.37 | 3 | 1.06 | | 14 |
| Ship04s | 402 | 8 | 0.97 | | 8 | 0.70 | 0.70 | 18 |
| Ship08l | 778 | 7 | 4.67 | 7.02 | 6 | 3.77 | | 17 |
| Ship08s | 778 | 12 | 1.76 | | 13 | 1.89 | 1.91 | 25 |
| Ship12l | 1151 | 14 | 6.24 | | 33 | 6.79 | 7.81 | 51 |
| Ship12s | 1151 | 15 | 2.55 | | 35 | 2.85 | 3.57 | 52 |
| Sierra | 1227 | 138 | 1.57 | | 207 | 2.71 | 5.43 | 210 |
| Standata | 359 | 6 | 2.22 | | 11 | 0.53 | 0.59 | 29 |
| Standgub | 361 | 6 | 2.24 | | 11 | 0.59 | 0.66 | |
| Standmps | 467 | 30 | 2.34 | | 51 | 0.69 | 1.52 | |
| Vtp.base | 198 | 12 | 0.15 | | 51 | 0.09 | 1.30 | 59 |

may also require less time to get to the same number of infeasibilities even when the time for simplex iterations is included. Thus the Reid algorithm is more efficient overall in 20 of the cases. This came to us as a surprise; we expected to find evidence for abandoning the Reid algorithm in favour of the tearing algorithm, but rather found that both are worthy algorithms.

## 7. Using alternative algorithms

We show in Tables 7.1 to 7.4 the results on our test problems of using the normal equations reordering algorithm instead of the $P^5$ algorithm. They correspond to Tables 6.1, 6.2, 6.4, 6.5, respectively.

Table 7.1

The Harwell test problem attributes, using normal matrix algorithm

| Problem | No. rows ($m$) | Block sizes $m_1, \ldots, m_r$ |
|---|---|---|
| Sc50a | 50 | 44 (1), 3 (2) |
| Sc50b | 50 | 41 (1), 3 (2), 1 (3) |
| Blend | 74 | 48 (1), 5 (2), 1 (4), 1 (5), 1 (7) |
| Sc105 | 105 | 91 (1), 7 (2) |
| Boeing2 | 166 | 162 (1), 2 (2) |
| Boeing1 | 351 | 345 (1), 3 (2) |
| Stair | 356 | 278 (1), 24 (2), 1 (3), 3 (4), 1 (6), 1 (9) |
| Finnis | 497 | 483 (1), 7 (2) |
| Shell | 536 | 504 (1), 11 (2), 2 (3), 1 (4) |
| Perold | 625 | 490 (1), 45 (2), 11 (3), 1 (4), 1 (8) |
| BP | 821 | 625 (1), 49 (2), 15 (3), 8 (4), 3 (5), 1 (6) |
| GUB | 929 | 903 (1), 1 (26) |

Legend: $k(l)$ means $k$ blocks of $l$ rows.

Table 7.2

Harwell test problem results for the normal matrix algorithm without and with backtracking

| Problem name | No. rows | Without | | $t = 5$ | | $t = 10$ | | $t = 50$ | |
|---|---|---|---|---|---|---|---|---|---|
| | | No. infeas. | Time (secs) | No. infeas. | Time (secs) | No. infeas. | Time (secs) | No. infeas. | Time (secs) |
| Sc50a | 50 | 7 | 0.02 | 4 | 0.02 | 2 | 0.03 | 0 | 0.07 |
| Sc50b | 50 | 4 | 0.02 | 4 | 0.02 | 2 | 0.02 | 0 | 0.06 |
| Blend | 74 | 0 | 0.04 | 0 | 0.05 | 0 | 0.05 | 0 | 0.05 |
| Sc105 | 105 | 14 | 0.04 | 9 | 0.05 | 8 | 0.07 | 1 | 0.19 |
| Boeing2 | 166 | 20 | 0.11 | 11 | 0.13 | 10 | 0.17 | 5 | 0.56 |
| Boeing1 | 351 | 50 | 0.30 | 28 | 0.53 | 27 | 0.60 | 19 | 1.69 |
| Stair | 356 | 156 | 0.31 | 145 | 0.40 | 122 | 0.54 | 81 | 2.19 |
| Finnis | 497 | 56 | 0.28 | 35 | 0.32 | 32 | 0.40 | 19 | 1.58 |
| Powell | 524 | 144 | 0.54 | 146 | 0.64 | 132 | 0.85 | 136 | 23.15 |
| Shell | 536 | 62 | 0.29 | 38 | 0.33 | 35 | 0.42 | 26 | 1.85 |
| Perold | 625 | 129 | 0.46 | 105 | 0.56 | 106 | 0.71 | 66 | 3.15 |
| BP | 821 | 184 | 0.81 | 161 | 0.96 | 140 | 1.18 | 111 | 5.63 |
| GUB | 929 | 27 | 0.79 | 27 | 0.81 | 22 | 0.87 | 22 | 0.83 |

Table 7.3

Netlib test problem attributes, using normal equations algorithm

| Problem | No. rows ($m$) | Block sizes $m_1, \ldots, m_r$ |
|---|---|---|
| Adlittle | 56 | 41 (1), 3 (2), 3 (3) |
| Afiro | 27 | 23 (1), 2 (2) |
| Bandm | 305 | 209 (1), 31 (2), 10 (3), 1 (4) |
| Beaconfd | 173 | 115 (1), 16 (2), 5 (3), 1 (4), 1 (7) |
| Bore3d | 233 | 155 (1), 17 (2), 3 (3), 1 (4), 2 (5), 1 (6), 1 (15) |
| Brandy | 220 | 117 (1), 15 (2), 6 (3), 1 (6), 1 (8), 1 (19), 1 (22) |
| Capri | 271 | 193 (1), 17 (2), 5 (3), 4 (4), 1 (13) |
| Etamacro | 400 | 319 (1), 29 (2), 5 (3), 2 (4) |
| E226 | 223 | 200 (1), 6 (2), 2 (3), 1 (5) |
| Forplan | 161 | 127 (1), 7 (2), 1 (3), 1 (4), 1 (13) |
| Ganges | 1309 | 1056 (1), 31 (2), 13 (3), 6 (4), 1 (5), 9 (12), 1 (15) |
| Gfrd-pnc | 616 | 560 (1), 25 (2), 2 (3) |
| Grow7 | 140 | 140 (1) |
| Grow15 | 300 | 300 (1) |
| Grow22 | 440 | 440 (1) |
| Israel | 174 | 174 (1) |
| Nesm | 662 | 652 (1), 5 (2) |
| Pilot.ja | 940 | 705 (1), 69 (2), 14 (3), 5 (4), 1 (5), 1 (6), 1 (7), 1 (17) |
| Pilotnov | 975 | 719 (1), 79 (2), 13 (3), 2 (4), 1 (5), 1 (6), 1 (7), 1 (8), 1 (25) |
| Pilot.we | 722 | 564 (1), 47 (2), 12 (3), 1 (4), 2 (5), 1 (6), 1 (8) |
| Pilot4 | 410 | 332 (1), 26 (2), 3 (3), 3 (4), 1 (5) |
| Recipe | 91 | 69 (1), 7 (2), 2 (4) |
| Scagr7 | 129 | 105 (1), 12 (2) |
| Scagr25 | 471 | 348 (1), 60 (2), 1 (3) |
| Scfxm1 | 330 | 261 (1), 21 (2), 4 (3), 1 (4), 1 (5), 1 (6) |
| Scfxm2 | 660 | 508 (1), 44 (2), 9 (3), 3 (4), 1 (5), 2 (6), 1 (8) |
| Scfxm3 | 990 | 752 (1), 62 (2), 14 (3), 7 (4), 1 (5), 4 (6), 1 (7), 1 (8) |
| Scorpion | 388 | 175 (1), 18 (2), 6 (3), 8 (4), 6 (5), 1 (6), 2 (7), 2 (9), 1 (11), 2 (14), 1 (20) |
| Scrs8 | 490 | 412 (1), 39 (2) |
| Scsd1 | 77 | 64 (1), 5 (2), 1 (3) |
| Scsd6 | 147 | 117 (1), 11 (2), 1 (3), 1 (5) |
| Scsd8 | 397 | 319 (1), 39 (2) |
| Sctap1 | 300 | 300 (1) |
| Sctap2 | 1090 | 1090 (1) |
| Sctap3 | 1480 | 1480 (1) |
| Sc205 | 205 | 175 (1), 15 (2) |
| Seba | 515 | 513 (1), 1 (2) |
| Share1b | 117 | 64 (1), 6 (2), 9 (3), 1 (4), 2 (5) |
| Share2b | 96 | 83 (1), 5 (2), 1 (3) |
| Ship04l | 402 | 278 (1), 1 (2), 1 (3), 1 (4), 1 (23), 1 (43), 1 (49) |
| Ship04s | 402 | 304 (1), 4 (2), 1 (3), 2 (5), 2 (17), 1 (43) |
| Ship08l | 778 | 507 (1), 9 (2), 3 (9), 3 (16), 3 (18), 3 (19), 1 (67) |
| Ship08s | 778 | 641 (1), 8 (2), 3 (8), 3 (10), 1 (67) |
| Ship12l | 1151 | 957 (1), 8 (2), 1 (3), 1 (5), 1 (60), 1 (110) |
| Ship12s | 1151 | 992 (1), 9 (2), 1 (6), 2 (7), 1 (11), 1 (110) |
| Sierra | 1227 | 1004 (1), 75 (2), 23 (3), 1 (4) |
| Standata | 359 | 335 (1), 9 (2), 2 (3) |
| Standgub | 361 | 335 (1), 10 (2), 2 (3) |
| Standmps | 467 | 446 (1), 4 (2), 3 (3), 1 (4) |
| Vtp.base | 198 | 160 (1), 17 (2), 1 (4) |

Legend: $k(l)$ means $k$ blocks of $l$ rows.

Table 7.4

Netlib test problem results for the normal equations algorithm without and with backtracking

| Problem name | No. rows | Without | | $t = 5$ | | $t = 10$ | | $t = 50$ | |
|---|---|---|---|---|---|---|---|---|---|
| | | No. infeas. | Time (secs) | No. infeas. | Time (secs) | No. infeas. | Time (secs) | No. infeas. | Time (secs) |
| Adlittle | 56 | 7 | 0.03 | 8 | 0.03 | 5 | 0.04 | 0 | 0.19 |
| Afiro | 27 | 0 | 0.01 | 0 | 0.01 | 0 | 0.01 | 0 | 0.01 |
| Bandm | 305 | 67 | 0.21 | 54 | 0.25 | 41 | 0.32 | 30 | 1.22 |
| Beaconfd | 173 | 30 | 0.20 | 29 | 0.21 | 28 | 0.23 | 23 | 0.80 |
| Bore3d | 233 | 4 | 0.13 | 4 | 0.13 | 4 | 0.14 | 4 | 0.24 |
| Brandy | 220 | 49 | 0.15 | 45 | 0.16 | 43 | 0.20 | 36 | 0.76 |
| Capri | 271 | 84 | 0.17 | 62 | 0.21 | 60 | 0.26 | 48 | 1.20 |
| Etamacro | 400 | 64 | 0.23 | 61 | 0.27 | 56 | 0.35 | 30 | 1.29 |
| E226 | 223 | 28 | 0.19 | 17 | 0.20 | 15 | 0.24 | 7 | 1.13 |
| Forplan | 161 | 14 | 0.21 | 7 | 0.21 | 7 | 0.23 | 6 | 4.40 |
| Ganges | 1309 | 116 | 0.74 | 90 | 0.80 | 76 | 0.94 | 48 | 2.90 |
| Gfrd-pnc | 616 | 3 | 0.27 | 2 | 0.28 | 2 | 0.30 | 2 | 0.53 |
| Grow7 | 140 | 0 | 0.12 | 0 | 0.13 | 0 | 0.13 | 0 | 0.16 |
| Grow15 | 300 | 0 | 0.27 | 0 | 0.27 | 0 | 0.28 | 0 | 0.36 |
| Grow22 | 440 | 0 | 0.40 | 0 | 0.40 | 0 | 0.42 | 0 | 0.54 |
| Israel | 174 | 8 | 0.32 | 6 | 0.33 | 6 | 0.35 | 13 | 0.64 |
| Nesm | 662 | 123 | 0.66 | 60 | 0.77 | 75 | 1.04 | 37 | 4.16 |
| Pilot.ja | 940 | 163 | 0.98 | 145 | 1.10 | 136 | 1.35 | 140 | 7.45 |
| Pilotnov | 975 | 208 | 0.91 | 196 | 1.04 | 190 | 1.33 | 170 | 5.95 |
| Pilot.we | 722 | 130 | 0.52 | 115 | 0.60 | 114 | 0.76 | 88 | 4.48 |
| Pilot4 | 410 | 69 | 0.37 | 46 | 0.41 | 49 | 0.50 | 35 | 2.29 |
| Recipe | 91 | 1 | 0.05 | 0 | 0.05 | 0 | 0.05 | 0 | 0.07 |
| Scagr7 | 129 | 17 | 0.06 | 16 | 0.07 | 15 | 0.09 | 3 | 0.40 |
| Scagr25 | 471 | 76 | 0.21 | 70 | 0.25 | 69 | 0.34 | 20 | 1.52 |
| Scfxm1 | 330 | 44 | 0.23 | 37 | 0.26 | 35 | 0.34 | 26 | 1.17 |
| Scfxm2 | 660 | 101 | 0.45 | 89 | 0.51 | 83 | 0.66 | 47 | 2.39 |
| Scfxm3 | 990 | 146 | 0.68 | 129 | 0.75 | 120 | 0.96 | 69 | 3.56 |
| Scorpion | 388 | 57 | 0.15 | 80 | 0.19 | 72 | 0.25 | 57 | 1.04 |
| Scrs8 | 490 | 25 | 0.26 | 16 | 0.28 | 16 | 0.32 | 6 | 0.79 |
| Scsd1 | 77 | 1 | 0.09 | 1 | 0.09 | 1 | 0.10 | 0 | 0.47 |
| Scsd6 | 147 | 6 | 0.16 | 5 | 0.16 | 3 | 0.18 | 4 | 0.42 |
| Scsd8 | 397 | 2 | 0.37 | 1 | 0.37 | 0 | 0.39 | 0 | 0.79 |
| Sctap1 | 300 | 33 | 0.15 | 32 | 0.17 | 31 | 0.22 | 23 | 0.73 |
| Sctap2 | 1090 | 51 | 0.65 | 50 | 0.70 | 50 | 0.80 | 50 | 2.41 |
| Sctap3 | 1480 | 61 | 0.93 | 53 | 0.98 | 54 | 1.10 | 54 | 2.90 |
| Sc205 | 205 | 32 | 0.08 | 18 | 0.10 | 16 | 0.13 | 1 | 0.61 |
| Seba | 515 | 9 | 1.57 | 9 | 1.58 | 9 | 1.61 | 9 | 1.94 |
| Share1b | 117 | 49 | 0.07 | 31 | 0.09 | 15 | 0.13 | 6 | 0.72 |
| Share2b | 96 | 15 | 0.06 | 15 | 0.07 | 14 | 0.09 | 2 | 0.93 |
| Ship04l | 402 | 71 | 0.30 | 70 | 0.31 | 61 | 0.37 | 69 | 1.01 |
| Ship04s | 402 | 34 | 0.26 | 33 | 0.26 | 23 | 0.30 | 32 | 0.78 |
| Ship08l | 778 | 60 | 0.63 | 59 | 0.64 | 54 | 0.71 | 37 | 1.59 |
| Ship08s | 778 | 20 | 0.48 | 17 | 0.48 | 21 | 0.53 | 19 | 1.08 |
| Ship12l | 1151 | 52 | 0.98 | 52 | 0.99 | 46 | 1.04 | 12 | 2.07 |
| Ship12s | 1151 | 27 | 0.69 | 27 | 0.70 | 24 | 0.76 | 24 | 1.36 |
| Sierra | 1227 | 274 | 0.75 | 217 | 0.97 | 197 | 1.31 | 165 | 6.45 |
| Standata | 359 | 14 | 0.22 | 11 | 0.25 | 10 | 0.29 | 10 | 0.87 |
| Standgub | 361 | 14 | 0.23 | 11 | 0.25 | 10 | 0.30 | 10 | 0.90 |
| Standmps | 467 | 22 | 0.32 | 17 | 0.33 | 17 | 0.38 | 6 | 1.80 |
| Vtp.base | 198 | 51 | 0.13 | 39 | 0.16 | 32 | 0.22 | 22 | 0.84 |

It may be seen from Tables 7.1 and 7.3 (when compared with Tables 6.1 and 6.4) that the normal equations algorithm is not quite so successful in obtaining small blocks, though the vast majority of them still have order less than 5. Note that the block sizes are not those of the bidiagonal form (5.1), but those of the form (5.2) obtained after applying row and column permutations to $\boldsymbol{A}$.

The data in Tables 7.2 and 7.4 shows that, as with the $P^5$ algorithm, some backtracking is worthwhile and that any value of $t$ between 5 and 10 would be suitable. We have again chosen to recommend 5.

If the times without backtracking are compared (Table 6.2 with Table 7.2 and Table 6.5 with Table 7.4), it may be seen that the two algorithms usually take about the same time, but that occasionally $P^5$ is much slower. This may be a quirk of the Harwell implementation, which was designed for square matrices. Examples illustrating this slowness are GUB, Standgub, and Standmps. In the case of Standgub and Standmps, the proportion of the total time taken by this part of the calculation is quite high (see Table 8.1).

Taken as a whole, we view the two algorithms as comparable. In terms of the number of infeasibilities, the $P^5$ algorithm is usually superior but this must be counterbalanced by its greater times. Some overall comparisons are made in Section 8.

Other variants that we have tried have been less successful. Modifying the $P^5$ algorithm to favour sparse columns early (see the second paragraph of Section 5) on the whole gave poorer results. We also tried to make the backtracking algorithm with high $t$ values competitive by limiting such backtracking to once every $\frac{1}{2}t$ rows, but found that this lost too much of the gains that are available from backtracking.

## 8. Comparison between the new algorithms

Finally, we use the bases generated by our crash algorithms as an initial basis for the solution of the linear programming problems (6.1) and (6.2). We report on the progress of the linear programming code LA04 in Tables 8.1, 8.2 and 8.3; we give details of the number of iterations and timings to obtain a feasible point and an optimal solution for the problems along with the optimal objective function value obtained. LA04 is a standard simplex code, maintaining and updating a sparse triangular factorization of the basis (see Reid, 1982) and using complete pricing together with a steepest-edge strategy (see Goldfarb and Reid, 1977) as a pricing mechanism. We selected and solved a subset of the larger problems from our two test sets and solved the unscaled versions of the problems.

A direct comparison is also made between the three methods in the tables. Any entry prefixed by * is the best of the three or within 20% of the best, any entry prefixed by † is within a factor of 2 of the best, and the remainder are not labelled. The numbers of cases in these categories for the four columns of Table 8.1 are 18-5-5, 10-11-7, 17-10-1, 15-11-2; for Table 8.2 they are 6-8-14, 13-8-7, 18-7-3, 21-6-1; and for Table 8.3 they are 12-3-13, 13-7-8, 10-10-8, 10-8-10. Thus both the $P^5$ tearing

Table 8.1
Details of LP run on test problems using the $P^5$ tearing crash algorithm with $t = 5$ backtracking

| Problem | Feasibility phase | | Feasibility and optimality phase | | |
|---|---|---|---|---|---|
| | Iterations | Time (secs) | Iterations | Time (secs) | Optimal value obtained |
| BP | * 531 | * 28.65 | * 1858 | * 119.92 | 5501.8494 |
| Capri | * 56 | * 0.86 | * 101 | * 1.47 | 2690.0120 |
| Etamacro | * 94 | * 1.59 | * 492 | † 8.17 | −310.1144 |
| Finnis | * 187 | † 3.35 | * 374 | * 6.47 | 112447.8754 |
| Forplan | 129 | 3.70 | † 208 | † 5.02 | −605.1331 |
| Ganges | 222 | 6.50 | † 511 | † 16.35 | −109586.2817 |
| Gfrd-pnc | 39 | † 1.17 | † 611 | † 9.66 | 6902236.5352 |
| GUB | * 493 | † 25.11 | * 1007 | † 52.48 | 2185196.7563 |
| Nesm | 686 | 35.57 | † 2232 | † 124.08 | 14076041.1386 |
| Pilot4 | * 213 | * 5.98 | * 931 | * 35.78 | −2581.1403 |
| Powell | † 160 | 4.93 | * 263 | * 7.10 | 553329.0897 |
| Scagr25 | * 43 | * 1.02 | * 212 | * 3.61 | −14753431.8152 |
| Scfxm2 | * 354 | * 8.69 | * 630 | * 15.71 | 36660.2603 |
| Scfxm3 | † 554 | † 18.68 | * 956 | * 32.39 | 54901.2525 |
| Scorpion | * 43 | † 0.81 | * 103 | * 1.59 | 1878.1234 |
| Scrs8 | † 219 | † 4.66 | * 421 | * 9.68 | 904.2970 |
| Scsd8 | * 1 | † 3.08 | † 582 | † 25.00 | 905.0000 |
| Sctap2 | * 7 | * 4.11 | † 542 | † 18.22 | 1724.8071 |
| Sctap3 | * 3 | * 6.08 | † 706 | † 30.55 | 1424.0000 |
| Seba | * 77 | * 2.82 | * 650 | * 9.63 | 15711.6009 |
| Shell | † 72 | † 2.24 | * 302 | * 5.82 | 1208825346.0000 |
| Ship08l | * 14 | 7.55 | † 572 | † 28.68 | 1909055.1849 |
| Ship08s | † 20 | † 3.51 | * 262 | * 9.39 | 1920098.1832 |
| Ship12l | * 22 | † 11.67 | * 283 | * 24.49 | 1470187.9323 |
| Ship12s | * 24 | † 5.19 | * 234 | * 11.65 | 1489236.1478 |
| Sierra | * 246 | * 10.79 | † 676 | † 23.59 | 15394363.5594 |
| Standgub | * 6 | 2.68 | 102 | 3.87 | 1257.6995 |
| Standmps | 184 | 5.15 | † 298 | 6.69 | 1406.0175 |

* Best of the three algorithms or within 20% of the best.
† Within factor 2 of the best.

algorithm and the Reid algorithm appear to be "winners", but each algorithm outperforms the others in a significant number of cases. Unfortunately, there does not seem to be any particular pattern to explain this behaviour. Timings provide a fairer comparison because the tearing algorithm, particularly with the $P^5$ algorithm in use, is slower and does not produce an initial basis that is a permutation of a triangular matrix (though in practice it is nearly so). This in turn implies that extra computation is required to calculate the initial steepest-edge weights.

Table 8.2
Details of LP run on test problems using Reid's crash algorithm

| Problem | Feasibility phase | | | | Feasibility and optimality phase | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Iterations | | Time (secs) | | Iterations | | Time (secs) | | Optimal value obtained |
| BP | † | 785 | † | 44.94 | * | 2010 | * | 134.89 | 5501.8494 |
| Capri | | 159 | † | 1.36 | † | 184 | * | 1.71 | 2690.0119 |
| Etamacro | | 216 | † | 2.58 | * | 486 | * | 6.42 | −310.1161 |
| Finnis | † | 275 | † | 3.84 | * | 430 | * | 6.28 | 112447.8805 |
| Forplan | | 78 | | 1.49 | * | 162 | * | 2.96 | −605.1328 |
| Ganges | * | 1 | * | 0.90 | * | 400 | * | 14.62 | −109586.5268 |
| Gfrd-pnc | | 210 | | 3.22 | * | 421 | * | 6.71 | 6902236.5352 |
| GUB | * | 451 | * | 16.18 | * | 957 | * | 41.98 | 2185196.7563 |
| Nesm | | 1257 | | 59.04 | † | 2755 | † | 129.10 | 14076032.7917 |
| Pilot4 | † | 272 | * | 6.36 | * | 995 | * | 37.52 | −2583.0383 |
| Powell | * | 108 | * | 2.20 | * | 312 | * | 6.73 | 520039.4539 |
| Scagr25 | | 194 | | 3.42 | † | 351 | † | 6.24 | −14753431.8152 |
| Scfxm2 | † | 393 | † | 9.18 | * | 691 | * | 17.50 | 36660.2603 |
| Scfxm3 | † | 579 | † | 18.03 | * | 982 | * | 33.30 | 54901.2525 |
| Scorpion | † | 65 | * | 0.67 | | 251 | † | 2.65 | 1878.1247 |
| Scrs8 | | 333 | | 6.10 | † | 537 | * | 11.28 | 904.2970 |
| Scsd8 | | 18 | * | 2.43 | | 1078 | | 43.80 | 905.0000 |
| Sctap2 | | 188 | † | 6.29 | * | 329 | * | 10.29 | 1724.8071 |
| Sctap3 | | 266 | † | 11.74 | * | 444 | * | 18.07 | 1424.0000 |
| Seba | | 393 | | 8.62 | † | 949 | † | 13.43 | 15711.6009 |
| Shell | * | 50 | * | 1.36 | † | 442 | † | 7.73 | 1208825346.0000 |
| Ship08l | * | 13 | * | 3.43 | * | 428 | * | 19.44 | 1909055.1849 |
| Ship08s | † | 27 | * | 2.09 | * | 249 | * | 8.02 | 1920098.1832 |
| Ship12l | | 47 | * | 7.17 | | 767 | † | 43.28 | 1470187.9323 |
| Ship12s | | 51 | * | 3.61 | † | 366 | * | 13.80 | 1489236.1478 |
| Sierra | * | 252 | * | 9.46 | * | 510 | * | 17.22 | 15402297.4208 |
| Standgub | † | 12 | * | 0.62 | * | 46 | * | 1.07 | 1257.6995 |
| Standmps | | 145 | | 2.74 | * | 168 | * | 3.13 | 1406.0175 |

* Best of the three algorithms or within 20% of the best.
† Within factor 2 of the best.

## 9. Conclusions

We have proposed three crash algorithms, one based on finding a basis matrix that is a permutation of an upper-triangular matrix and the others based on restructuring the rectangular set of equations to a block lower-triangular form and then using a tearing technique. For the tearing techniques, we found that a modest amount of backtracking is worthwhile. Our experimental results indicate that each of the algorithms is superior to the others about a third of the time. In terms of the number of infeasibilities, they show better performance than the simple crash algorithm in

Table 8.3
Details of LP run on test problems using the normal equations crash algorithm with $t = 5$ backtracking

| Problem | Feasibility phase | | | | Feasibility and optimality phase | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Iterations | | Time (secs) | | Iterations | | Time (secs) | | Optimal value obtained |
| BP | † | 887 | † | 51.16 | * | 2193 | * | 142.15 | 5501.8494 |
| Capri | | 140 | † | 1.44 | | 329 | | 3.42 | 2690.0019 |
| Etamacro | * | 112 | * | 1.77 | † | 622 | † | 9.81 | −310.1146 |
| Finnis | * | 158 | * | 2.69 | * | 392 | * | 6.52 | 112447.8804 |
| Forplan | * | 13 | * | 0.54 | | 371 | | 7.28 | −605.1331 |
| Ganges | | 69 | | 3.67 | † | 683 | † | 27.61 | −109586.5124 |
| Gfrd-pnc | * | 5 | * | 0.81 | * | 434 | * | 7.13 | 6902236.5352 |
| GUB | * | 458 | * | 17.17 | * | 916 | * | 41.18 | 2185196.7563 |
| Nesm | * | 229 | * | 8.83 | * | 1710 | * | 78.47 | 14076041.2188 |
| Pilot4 | | 1230 | | 50.35 | | 2231 | | 97.34 | −2581.1402 |
| Powell | | 344 | | 7.78 | | 680 | | 16.98 | 555678.5230 |
| Scagr25 | | 155 | | 2.54 | † | 304 | † | 4.74 | −14753431.8152 |
| Scfxm2 | * | 324 | * | 7.47 | * | 619 | * | 14.66 | 36660.2603 |
| Scfxm3 | * | 399 | * | 12.44 | * | 1062 | * | 33.84 | 54901.2525 |
| Scorpion | | 89 | | 1.36 | | 221 | † | 2.89 | 178.1247 |
| Scrs8 | * | 116 | * | 2.48 | * | 446 | * | 9.57 | 904.2970 |
| Scsd8 | † | 2 | * | 2.13 | * | 378 | * | 14.50 | 905.0000 |
| Sctap2 | | 208 | | 10.13 | † | 569 | | 21.63 | 1724.8071 |
| Sctap3 | | 299 | | 20.68 | † | 783 | | 40.29 | 1424.0000 |
| Seba | | 463 | | 12.15 | † | 1010 | † | 17.61 | 15711.6009 |
| Shell | | 101 | † | 2.44 | * | 323 | * | 5.81 | 1208825346.0000 |
| Ship08l | | 59 | † | 5.94 | † | 823 | | 39.25 | 1909055.1849 |
| Ship08s | * | 17 | * | 2.47 | † | 363 | † | 12.71 | 1920098.1832 |
| Ship12l | | 52 | † | 8.77 | | 986 | | 52.78 | 1470187.9323 |
| Ship12s | * | 27 | * | 3.81 | † | 415 | † | 15.63 | 1489236.1478 |
| Sierra | † | 374 | † | 13.67 | | 1172 | | 36.53 | 15394362.5804 |
| Standgub | | 15 | † | 0.90 | | 128 | | 2.24 | 1257.6995 |
| Standmps | * | 36 | * | 1.27 | † | 245 | † | 4.11 | 1406.0175 |

* Best of the three algorithms or within 20% of the best.
† Within factor 2 of the best.

MINOS on most (but not all) of our test problems. Based on this satisfactory experience, we propose to place code for the three algorithms in the Harwell Subroutine Library.

## Acknowledgements

We would like to express our thanks to Mike Saunders for providing us with the MINOS data, to David Gay for supplying us with the Netlib test problems, to Iain

Duff for reading the paper and suggesting improvements to the presentation, and to Jennifer Scott for checking the introduction from the point of view of a reader not familiar with linear programming and suggesting an improvement.

## References

D.M. Carstens, "Crashing techniques," in: W. Orchard-Hays, *Advanced Linear-Programming Computing Techniques* (McGraw-Hill, New York, 1968) pp. 131-139.

V. Chvátal, *Linear Programming* (Freeman, New York and San Francisco, 1983).

A.R. Curtis and J.K. Reid, "On the automatic scaling of matrices for Gaussian elimination," *Journal of the Institute of Mathematics and its Applications* 10 (1972) 118-124.

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

J.J. Dongarra and E. Grosse, "Distribution of mathematical software via electronic mail," *Communications of the ACM* 30 (1987) 403-407.

I.S. Duff, A.M. Erisman and J.K. Reid, *Direct Methods for Sparse Matrices* (Oxford University Press, London, 1986).

A.M. Erisman, R.G. Grimes, J.G. Lewis and W.G. Poole Jr., "A structurally stable modification of Hellerman-Rarick's $P^4$ algorithm for reordering unsymmetric sparse matrices," *SIAM Journal on Numerical Analysis* 22 (1985) 369-385.

D.M. Gay, "Electronic mail distribution of linear programming test problems," *Mathematical Programming Society COAL Newsletter* (December 1985).

N.E. Gibbs, "A hybrid profile reduction algorithm," *ACM Transactions on Mathematical Software* 2 (1976) 378-387.

N.E. Gibbs, W.G. Poole Jr. and P.K. Stockmeyer, "An algorithm for reducing the bandwidth and profile of a sparse matrix," *SIAM Journal on Numerical Analysis* 13 (1976) 236-250.

D. Goldfarb and J.K. Reid, "A practicable steepest-edge simplex algorithm," *Mathematical Programming* 12 (1977) 361-371.

Harwell Subroutine Library, "A catalogue of subroutine," Report R9185, HMSO (London, 1988).

J.G. Lewis, "Implementation of the Gibbs-Poole-Stockmeyer and Gibbs-King algorithms," *ACM Transactions on Mathematical Software* 8 (1982) 180-194.

B.A. Murtagh, *Advanced Linear Programming* (McGraw-Hill, New York, 1981).

B.A. Murtagh and M.A. Saunders, "MINOS 5.1 User's guide," Report SOL 83-20R, Department of Operations Research, Stanford University (Stanford, CA, 1987).

W. Orchard-Hays, *Advanced Linear-Programming Computing Techniques* (McGraw-Hill, New York, 1968).

C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).

J.K. Reid, "A sparsity exploiting variant of the Bartels-Golub decomposition for linear programming bases," *Mathematical Programming* 24 (1982) 55-69.

S.W. Sloan, "An algorithm for profile and wavefront reduction of sparse matrices," *International Journal for Numerical Methods in Engineering* 23 (1986) 239-251.