ORSA Journal on Computing
Vol. 1, No. 2, Spring 1989

0899-1499/89/0102-0084 $01.25

Data Structures and Programming Techniques for the Implementation of Karmarkar's Algorithm

ILAN ADLER *Department of Industrial Engineering and Operations Research, University of California, Berkeley, CA 94720, ARPANET: ilan343@violet.berkeley.edu*

NARENDRA KARMARKAR *AT&T Bell Laboratories, Murray Hill, NJ 07974, ARPANET: karmar@research.att.com*

MAURICIO G. C. RESENDE *Department of Industrial Engineering and Operations Research, University of California, Berkeley, CA 94720. (current address: AT&T Bell Laboratories, Murray Hill, NJ 07974), ARPANET: mgcr@research.att.com*

GERALDO VEIGA *Department of Industrial Engineering and Operations Research, University of California, Berkeley, CA 94720*

(Received April, 1988; accepted November, 1988)

This paper describes data structures and programming techniques used in an implementation of Karmarkar's algorithm for linear programming. Most of our discussion focuses on applying Gaussian elimination toward the solution of a sequence of sparse symmetric positive definite systems of linear equations, the main requirement in Karmarkar's algorithm. Our approach relies on a direct factorization scheme, with an extensive *symbolic factorization* step performed in a preparatory stage of the linear programming algorithm. An *interpretative* version of Gaussian elimination makes use of the symbolic information to perform the actual numerical computations at each iteration of algorithm. We also discuss ordering algorithms that attempt to reduce the amount of *fill-in* in the *LU* factors, a procedure to build the linear system solved at each iteration, the use of a dense window data structure in the Gaussian elimination method, a preprocessing procedure designed to increase the sparsity of the linear programming coefficient matrix, and the special treatment of dense columns in the coefficient matrix.

All variants of Karmarkar's algorithm for linear programming [25] are closely related with respect to their main computational requirement—the solution of a sequence of sparse symmetric positive definite systems of linear equations with strong structural and numerical correlation. A typical example of such systems is the sequence that arises in the *dual-affine scaling* variant of Karmarkar's algorithm[1]

$$(AD_k^2A^T)d_y = b, \qquad (1)$$

where A is a sparse $m \times n$ matrix, D_k is a positive diagonal $m \times m$ matrix, and d_y and b are m-vectors. In the above system, the diagonal matrix changes from iteration to iteration while A remains fixed. A successful implementation of Karmarkar's algorithm depends on an effective variant of the method, and on specialized data structures and programming techniques to handle the sequence of linear systems.

A wealth of solution methods for solving large sparse systems of linear equations has been developed, most of them falling under two categories:

(i) *Direct methods* involve the factorization of the system coefficient matrix, usually obtained through Gaussian elimination. Implementations of methods in this class require the use of specific data structures and special pivoting strategies, in an attempt to reduce the amount of *fill-in* during Gaussian elimination. Notable examples of software using this type of technique are SPARSPAK,[14] YSMP[11] and MA27.[8,9]

(ii) *Iterative methods* generate a sequence of approximate solutions to the system of linear equations, usually involving only matrix-vector multiplications in the computation of each iterate. Methods like Jacobi, Gauss-Seidel, Chebychev, Lanczos (see [19] for a description of these methods) and the conjugate gradient are attractive by virtue of their low storage requirements, displaying, however, slow convergence unless an effective preconditioner is applied.

Subject classification: Analysis of algorithms: data structures; Programming, linear: algorithms, large-scale systems.

These two approaches are not necessarily disjoint, as advanced implementations of iterative methods are hybrid schemes, typically based on the conjugate gradient algorithm, with a preconditioner computed by techniques similar to Gaussian elimination. In most implementations of this procedure, the conjugate gradient method is used to solve the resulting system of equations that hopefully displays a more favorable eigenvalue structure.

The relative merits of each approach depends on such factors as the characteristics of the problem and the host machine. Size, density, nonzero pattern, range of coefficients, structure of eigenvalues and desired accuracy of the solution are some of the problem attributes to be considered. Beyond simple characteristics as speed and size of memory, other aspects of the host machine's architecture play a decisive role both in the selection of a solution method and in specific implementation details. Recent research in sparse matrix techniques concentrate on specializing algorithms that can achieve the most benefit from parallelism, pipelining or vectorization. Also important in the comparison of the two approaches in implementations dedicated to a specific machine is the data transfer rates between various memory media, like disk, main memory, cache memory and register files.

The main purpose of this paper is to present data structures and programming techniques for an implementation of Karmarkar's algorithm that makes use of direct factorization at each iteration. In this context, the systems of linear equations are solved via an *interpretative* Gaussian elimination scheme for direct factorization. We illustrate our approach on the *dual-affine* variant of Karmarkar's algorithm. Since most variants of Karmarkar's algorithm share the same main computational step, the techniques described here can be applied to other implementations using the direct factorization approach, as well as to building the preconditioner in hybrid methods. We limit our discussion to implementations on sequential computer architectures. Others [2, 17, 28, 30, 32, 41] have discussed implementations of variants of Karmarkar's algorithm but have not considered many of the issues treated in this paper.

Throughout this paper, we describe algorithms in a pseudo-code similar to the one used by Tarjan.[38] Our intention is to express the algorithms in a compact notation that conveys the operations in detail, without obscuring them with the minutiae required by an actual programming language. In Section 1, we reintroduce Algorithm I of Adler et al.[1] and present the *static* data structures for the linear programming coefficient matrix. In Section 2, we review the algebraic aspects of Gaussian elimination for symmetric dense matrices, and introduce a basic version of the procedure for sparse matrices. The importance of reordering the rows and columns of a system of linear equations is the subject of Section 3. Different orderings can result in lower *fill-in* in the LU factors and lower number of operations in the Gaussian elimination procedure. We indicate how properties of Karmarkar's algorithm influence the choice of the ordering algorithm. Section 4 describes an *interpretative* procedure for sparse Gaussian elimination, an important feature of the implementation of Karmarkar's algorithm presented in [1]. In Section 5, we discuss how to build $AD_k^2A^T$ at each iteration of the linear programming algorithm. By using an approximate scaling matrix, we describe how matrix $AD_k^2A^T$ can be updated based on the matrix used in the previous iteration. Section 6 discusses the use of a *dense window* in the Gaussian elimination procedure to reduce the storage requirements of the *interpretative* scheme. In Section 7 we discuss preprocessing of the input coefficient matrix. We show how to identify some fixed and null variables, some trivial cases of infeasibility and how to reduce the density of some input coefficient matrices. Numerical results are presented for these preprocessing schemes. In Section 8 we discuss a procedure for treating linear programs with dense columns in their coefficient matrices. This procedure makes use of a preconditioned conjugate gradient algorithm for solving the required symmetric system of sparse linear equations at each iteration of Karmarkar's algorithm. A summary and conclusion are presented in Section 9.

1. Linear Program Formulation and Description of Algorithm

Most of the discussion in this paper is based on Algorithm I of Adler et al.,[1] which we reintroduce in this section, altering the presentation slightly. This is an implementation of the so-called *dual-affine scaling* variant of Karmarkar's algorithm, which is applied to linear programs in inequality form. Since most linear programs are formulated in standard equality form, the algorithm is applied to the dual linear program, hence the *dual-affine* designation. Instead of expressing the linear program directly in inequality form, we consider the following standard equality form:

$$\text{P:} \qquad \text{minimize } c^Tx \tag{1.1}$$

$$\text{subject to: } Ax = b \tag{1.2}$$

$$x \geq 0 \tag{1.3}$$

where A is the full-rank $m \times n$ linear programming coefficient matrix, b is the m-dimensional resource vector, c is the n-dimensional objective vector, and x is the n-dimensional primal vector.

Since Algorithm I requires problems in inequality form and most practical problems are formulated in equality form, we apply the algorithm to the dual

problem of (P):

$$\text{D:} \qquad \text{maximize } b^T y \tag{1.4}$$

$$\text{subject to: } A^T y \leqslant c \tag{1.5}$$

where y is the m-dimensional dual vector.

PSEUDO-CODE 1—ALGORITHM I

```
procedure Algorithm I
                    (A, b, c, y^0, stopping criterion, γ)
1   k := 0
2   do stopping criterion not satisfied →
3     v^k := c − A^T y^k;
4     D_k := diag(1/v_1^k, ..., 1/v_m^k);
5     d_y := (A D_k^2 A^T)^{-1} b;
6     d_v := −A^T d_y;
7     α := γ × min{−v_i^k/(d_v)_i : (d_v)_i < 0,
                                        i = 1, ..., m};
8     y^{k+1} := y^k + α d_y;
9     x^{k+1} := D_k^2 d_v;
10    k := k + 1;
11  od
end Algorithm I;
```

Algorithm I, presented in Pseudo-code 1, requires as input an interior dual feasible solution y^0. However, a nonempty feasible dual interior cannot be assured, and an initial dual point may not be readily available, Adler et al.[1] propose a *Big M* scheme. According to this approach, an artificial variable is added to the dual problem resulting in the following problem:

$$\text{D}_a\text{:} \qquad \text{maximize } b^T y - M y_a \tag{1.6}$$

$$\text{subject to: } A^T y - e^T y_a \leqslant c \tag{1.7}$$

where $e^T = (1, 1, \ldots, 1)^T$ and M is a large constant. An interior dual feasible point is easily available for problem D_a. Furthermore, given a suitably large constant M, solving D_a either yields an optimal solution, or indicates primal or dual infeasibility for problem D. Theoretically, a very large value of M is required for the statement above to hold true. Due to numerical computation considerations, a lower value for M is necessary. This value is determined as a function of the problem data, effective for practical problems.

Algorithm I also requires that a *stopping criterion* be defined. In the experiments reported in [1], the algorithm terminates whenever either the relative improvement in the dual objective value is smaller than a given tolerance or unboundedness of the dual is detected. In theory, dual unboundedness is detected in Algorithm I whenever the ratio test in line 7 fails. In a practical implementation, an empirical criterion, such as imposing an upper-bound on the dual objective value, is used. An alternative stopping criterion uses the tentative primal solution computed at each step of the linear programming algorithm, terminating if primal feasibility and complementary slackness are satisfied.

In a practical implementation, Algorithm I is actually used in a Phase I/Phase II scheme. Initially, the algorithm is applied to D_a with a modified stopping criterion. In this Phase I stage, the algorithm either identifies an interior dual feasible solution, or, if no such solution exists, finds a point that satisfies the stopping criterion for problem D. With ϵ_f defined as the feasibility tolerance, the modified stopping criterion for Phase I is formulated as follows:

(i) If $y_a^k < 0$ at iteration k, then y^k is an interior feasible solution for problem D.
(ii) If the algorithm satisfies the regular stopping criterion and $y_a^k > \epsilon_f$, D is declared infeasible.
(iii) If the algorithm satisfies the regular stopping criterion and $y_a^k < \epsilon_f$, either unboundedness is detected or an optimal solution is found. In this case, D has no interior feasible solution.

If Phase I terminates according to condition (ii), Algorithm I is applied to problem D, starting with the last iterate of Phase I and using the regular stopping criterion.

In addition to the coefficients of the linear program, an initial dual feasible interior point and a stopping criterion, Algorithm I expects as input a parameter $0 < \gamma < 1$, that determines the step size at each iteration. At the start of each iteration, the algorithm computes the vector of dual slacks v^k (line 3) and the corresponding scaling matrix D_k (line 4). The code in line 5 indicates the solution of a symmetric system of linear equations that results in the search direction in the space of the dual variables. Note that before solving the system of linear equations, we must build its coefficient matrix $AD_k^2A^T$. The code in line 6 computes the associated search direction in the space of the dual slacks. Compared to other variants of Karmarkar's algorithm, the *dual-affine* has the advantage of allowing for inexact computation of the search direction without associated loss of feasibility. The code in line 7 initially determines the maximum dual feasible step, determined by the vector of dual slacks. The actual step length is computed by multiplying the maximum step value by a *safety factor* γ. (Other proposed variations of Karmarkar's algorithm compute the step in each iteration by minimizing a potential function on the search direction (see [40] and [20]). The iteration is completed by updating the dual solution (line 8) and computing the tentative primal solution (line 9).

Closing this section, we describe the basic data structures for storing the linear programming coefficient matrix. The storage scheme takes advantage of

the fact that none of the matrix computations in the algorithms described in this paper require random access of matrix elements. In fact, all operations access matrices either by rows or by columns. Also, these are all *static* data structures, that remain fixed throughout the execution of our implementation of Karmarkar's algorithm.

The linear programming coefficient matrix A is stored column-wise as a sequence of sparse vectors. A pointer to the starting position of each column provides the means for random access of an individual column. In addition, some of the operations performed in the preparatory stage of the algorithm require access of rows. The easiest way to achieve this goal is to store a duplicate representation of A row-wise. Within each column, nonzero elements are stored in order of increasing row index. To identify the nonzero entries in any given column, it is necessary to traverse the column, beginning with its first element.

Figure 1 depicts the column-wise representation of matrix A. This data structure is composed of five arrays, referred to by their FORTRAN designations:

- **ia** contains pointers to the first entry of each column. Component of **ia**$(n+1)$ points to the position after the last entry in A.
- **ja** contains the row indices corresponding to each entry of A.
- **a** contains the nonzero elements of A.
- **b** contains the dense representation of the resource vector b.
- **c** contains the dense representation of the objective vector c.

Since the columns are stored consecutively, the number of nonzero elements in the column i of A is given by

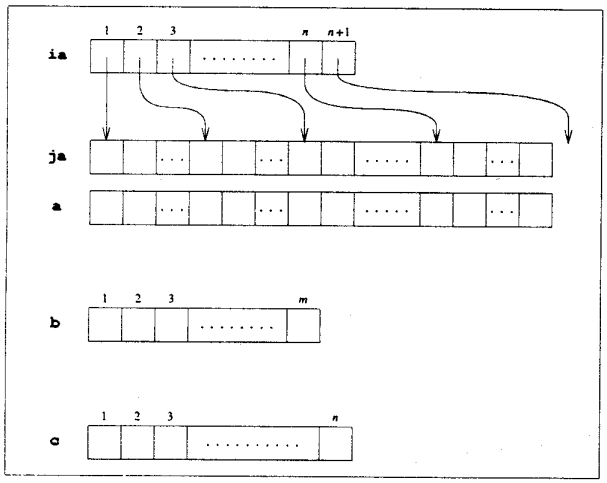

Figure 1. Data structure for coefficient matrix.

ia$(i+1)$ − **ia**(i). The row-wise data structure for A is the same as the column-wise data structure for A^T, and is stored in FORTRAN arrays **iat**, **jat** and **at**. The computational benefit derived from storing the resource vector b and the objective vector c in sparse form is insignificant. Consequently, they are stored in dense form in FORTRAN arrays **b** and **c**, respectively.

2. Direct Factorization of Symmetric Positive Definite Systems

In most of the recently proposed interior point algorithms for linear programming, the time required to perform an iteration is dominated by the solution of a sparse symmetric positive definite system of linear equations. In some instances, this system is embedded in the solution of a least-squares problem,[17] or in the computation of a projected gradient onto the null space of matrix A.

In the variant of Karmarkar's algorithm presented in Pseudo-code 1, the system of linear equations (2.1) yields the search direction d_y for each iteration k of the algorithm,

$$(AD_k^2A^T)d_y = b, \tag{2.1}$$

where A is the $m \times n$ linear programming coefficient matrix, b the resource vector, D_k the scaling matrix for iteration k, and d_y the resulting search direction in the space of the dual variables. Under the full-rank assumption for A, system (2.1) is symmetric and positive definite. There is a sharp division of solution techniques depending on whether a system of equations is symmetric positive definite or not. *Direct methods* designed for the solution of symmetric positive definite systems do not require numerical pivoting for stability.[18] As a consequence, in the case of sparse systems, as discussed in Sections 3 and 4, we can order the matrix and perform the *symbolic factorization* step based solely on the nonzero pattern, without regard to the actual numerical values.

In this section, we describe the algebraic procedure for the direct solution of a symmetric positive definite system of linear equations. Gaussian elimination and other equivalent methods for solving systems of linear equations consist of obtaining an LU decomposition of the system matrix, followed by the solution of two triangular systems of equations (see [10], [14], and [19] for a complete treatment of this subject). Also, considerations of sparsity are paramount in the practical implementation of Gaussian elimination. We present, in the end of this section, a basic implementation of a sparse symmetric Gaussian elimination procedure.

Consider system (2.1) that determines the feasible search direction computed in each iteration k of Algorithm I. Rewriting (2.1) in a more compact form

we have:

$$B_k d_y = b, \tag{2.2}$$

with

$$B_k = AD_k^2 A^T. \tag{2.3}$$

Even in the unsymmetric case, such systems of linear equations are usually solved by means of the LU factorization,

$$B_k = LU, \tag{2.4}$$

where L is an $m \times m$ unit lower triangular matrix (a matrix that has exclusively ones in its main diagonal), and U is an $m \times m$ upper-triangular matrix. Using (2.2) and (2.4) we have

$$(LU)d_y = b. \tag{2.5}$$

In the case of positive definite matrices, this LU factorization always exists and is unique. Note that if the system is also symmetric, the factor L can be trivially obtained from U with

$$L = D_U^{-1} U^T, \tag{2.6}$$

where D_U is a diagonal matrix with elements drawn from the diagonal of U. Furthermore, the LU factors are related to the Cholesky factors as follows:

$$\tilde{L} = (D_U)^{1/2} L, \tag{2.7}$$

and

$$\tilde{L}^T = (D_U)^{-1/2} U \tag{2.8}$$

where $\tilde{L}$ and $\tilde{L}^T$ are the Cholesky factors of B_k. In some circumstances, we need the symmetry displayed by the Cholesky factors, as in the computation of symmetric preconditioners for the conjugate gradient method described in Section 8. Otherwise, the LU factorization, which does not involve square root calculations, is preferred.

Conceptually, solving system (2.5) is equivalent to solving two triangular systems of linear equations. In a first step, we perform a *forward substitution*, solving

$$Lz = b, \tag{2.9}$$

where z is an m-vector. This is followed by a *back substitution*, solving

$$Ud_y = z. \tag{2.10}$$

This version of the Gaussian elimination procedure is equivalent to applying row transformations to the upper-triangular portion of B_k. After $m - 1$ iterations, the upper-triangular matrix is transformed into U. We start with the $m \times m$ matrix

$$U^{(0)} = upper(B_k). \tag{2.11}$$

where $upper(B_k)$ denotes the upper-triangular portion of B_k, including the diagonal elements. In each iteration q of the Gaussian elimination procedure, we perform an elementary transformation that creates zeros in the lower triangular portion of column q of $U^{(q)}$,

$$U^{(q)} = L^{(q)} U^{(q-1)}, \tag{2.12}$$

where

$$L^{(q)} = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -l_{q+1,q} & & \\ & & \vdots & \ddots & \\ & & -l_{m,q} & & 1 \end{bmatrix}, \tag{2.13}$$

and

$$l_{i,q} = u_{q,i}^{(q)} / u_{q,q}^{(q)}, \tag{2.14}$$

with $u_{i,j}^{(q)}$ denoting element (i, j) of $U^{(q)}$. For symmetric positive definite systems, $u_{q,q}^{(q)} > 0$ and the operation in (2.14) is well defined. After $m - 1$ iterations, we obtain the upper-triangular factor

$$U = U^{(m-1)} = L^{(m-1)} L^{(m-2)} \ldots L^{(1)} U^{(0)}. \tag{2.15}$$

The unit lower triangular factor can be expressed as

$$L = (L^{(1)})^{-1} (L^{(2)})^{-1} \ldots (L^{(m-1)})^{-1}. \tag{2.16}$$

PSEUDO-CODE 2—SYMMETRIC GAUSSIAN ELIMINATION

```
procedure SGE(m, A, U)
1  for q = 1, ..., m − 1 →
2    for i = q + 1, ..., m →
3      u_ii := u_ii − u_qi^2/u_qq;
4      for j = i + 1, ..., m →
5        u_ij := u_ij − u_qj u_qi/u_qq;
6      rof
7    rof
8  rof
end SGE;
```

Procedure SGE presented in Pseudo-code 2 executes the symmetric Gaussian elimination algorithm in its simplest form, without taking into consideration the sparsity patterns of the matrices B_k and the corresponding LU factors. For large-scale linear programs, the coefficient matrix A is usually sparse. The successful implementations of interior point methods[1, 17, 30, 32, 42] suggest that positive results are obtained whenever the level of sparsity of A is preserved in $AD_k^2 A^T$ as much as possible.

Next, we illustrate the difficulties encountered in the efficient implementation of the Gaussian elimination procedure for sparse symmetric matrices. At this

stage, we assume that system (2.2) is such that matrix B_k has been analyzed and ordered towards sparsity preservation. Ordering for sparsity is the subject of Section 3 of this paper. The nonzero pattern for the LU factors and sequence of pivots are determined before the actual factorization procedure. Accordingly, ordering the matrix and creating data structures take into consideration the nonzero pattern alone, disregarding their numerical values. For the purposes of this discussion, we assume the solution procedure to be divided into the following steps:

(i) Analyze the sparsity pattern of B_k and determine a suitable symmetric permutation. Build the data structure for the LU factors, including *fill-in* entries.
(ii) Perform Gaussian elimination to obtain LU factors.
(iii) Perform *forward* and *back substitutions.*

In step (i), we build the data structure depicted in Figure 2, where we initially load the super-diagonal elements of B_k in a row-wise representation, with the diagonal elements in a separate dense array. As implied by (2.15), this is the starting matrix $U^{(0)}$ in the sequence $\{U^{(0)}, U^{(1)}, \ldots, U^{(m)}\}$ built by the symmetric Gaussian elimination procedure. This data structure is similar to the one described by Figure 1, which stores the coefficient matrix A. Matrix U, however, is stored row-wise, with its diagonal elements in a separate array. The data structure is composed of four arrays, referred to by their FORTRAN designations:

diag contains the diagonal elements of U.
iaat contains pointers to the first off-diagonal entry of each row of U. Component **iaat**(m) points to the position after the last nonzero entry in U.
jaat contains the column indices corresponding to each entry of U.
aat contains the nonzero elements of U stored row-wise. L can be obtained implicitly from U as implied in (2.6).

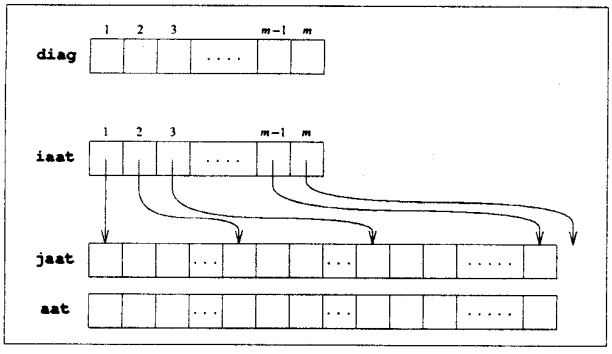

Figure 2. Data structure for LU factors.

PSEUDO-CODE 3—BASIC SPARSE SYMMETRIC GAUSSIAN ELIMINATION

```
procedure BSSGE(m, diag, iaat, jaat, aat)
1   for q = 1, ..., m − 1 →
2     for i = q + 1, ..., m → p(i) := 0 rof;
3     for i = iatt(q), ..., iaat(q + 1) − 1 →
      p(jaat(i)) := i rof;
4     for i = iaat(q), ..., iaat(q + 1) − 1 →
5       l := jaat(i);
6       diag(l) := diag(l) − aat(i)²/diag(q);
7       for j = iaat(l), ..., iaat(l + 1) − 1 →
8         if p(jaat(j)) ≠ 0 →
9           aat(j) := aat(j) − aat(i) ×
            aat(p(jaat(j))/diag(q);
10        fi
11      rof
12    rof
13  rof
end BSSGE;
```

Pseudo-code 3 describes the operation corresponding to step (ii)—performing the Gaussian elimination procedure. For each pivot row q, the code in lines 2–12 applies an elementary transformation to $U^{(q-1)}$ as indicated in (2.12). First, in lines 2 and 3, we build a dense array representation of row q in dense array p, with links to the position of each nonzero element in the data structure. This will be used to speed up the operation performed in lines 7–11, where row q is merged with each transformed row. In lines 4–12, the procedure scans pivot column q of $L^{(q)}$. Since $L^{(q)}$ and $(U^{(q)})^T$ have identical structures, we actually traverse row q of $U^{(q)}$. As output, this procedure provides the upper-triangular factor U stored in the same data structure used as input.

This procedure can be made more efficient by avoiding the repetitive scheme of creating a dense representation of each pivot row. This is the central purpose of the improved Gaussian elimination procedure described later.

In step (iii) operations corresponding to (2.9) and (2.10) are carried out. In practice, the *forward substitution* operation is incorporated to the Gaussian elimination procedure, by applying all elementary transformations to the resource vector b. However, splitting the computational procedures is important for some variants of Karmarkar's algorithm.[1,20,43] In these variants, computing the search direction in each iteration requires the solution of linear systems with identical coefficient matrices but different right-hand sides. The implementation of the *forward* and *back substitution* operations are described in several matrix computation references.[10,14,19]

3. Ordering for Sparsity

In the previous section, no attention was given to the sparsity pattern of the system of linear equations. In most practical large-scale linear programs, the coefficient matrix A is of low density. With some exceptions described in Section 8, this matrix will generate, at each iteration of the linear programming algorithm, a sparse matrix B_k. However, the application of Gaussian elimination to this matrix causes *fill-in*. The nonzero structure of the L and U factors displays entries in positions which are zero-valued in B_k. For this reason, the data structure depicted in Figure 2 is based on the nonzero pattern of the upper-triangular factor U, including the *fill-in* elements. Consequently, procedure BSSGE does not add new elements to the data structure, using instead the memory positions already allocated to the *fill-in* elements. In general terms, the amount of storage and number of operations necessary when applying Gaussian elimination to the B_k matrix grows with the number of *fill-in* elements created during this process. It is possible to build examples where the number of operations necessary to perform the Gaussian elimination method increases in spite of a reduction in the number of *fill-in* elements.

The total number of *fill-in* elements depends on the ordering of rows and columns of B_k. Consider, for example, problem *Bandm* from the NETLIB collection.[12] Starting with the sparse linear programming coefficient matrix presented in Figure 3, we illustrate the *fill-in* phenomenon by depicting the nonzero pattern for the B_k in Figure 4, and the nonzero pattern of the corresponding LU factors in Figure 5. In spite of the low density of the B_k, its LU factors turn out to be very dense.

The high level of *fill-in* can be reduced by applying an appropriate symmetric permutation to B_k. Algebraically, we select a permutation matrix P and transform the system for each iteration k of the linear programming algorithm as follows:

$$(PB_kP^T)Pd = Pb. \tag{3.1}$$

Even for the symmetric case, finding the permutation that yields the minimum *fill-in* in the LU factors is an NP-hard problem.[44] As a practical consequence, we cannot expect to find an efficient algorithm that identifies an optimal ordering. Hence, heuristics are devised that efficiently compute a permutation that approximates the effect of the optimal one, at least in matrices derived from real-world applications.

We examine below two frequently used sparsity preserving ordering heuristics. The first follows the Markowitz criterion,[29] which is designed for unsymmetric matrices. In general terms, the Markowitz ordering heuristic begins with a given matrix B_k, and at each stage of the Gaussian elimination, reorders columns and rows in such way as to minimize the

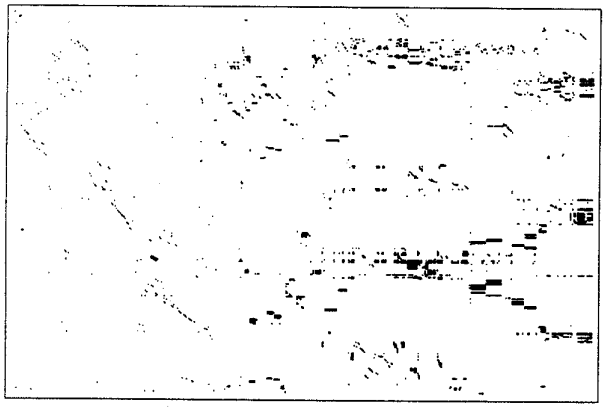

Figure 3. Nonzero pattern of coefficient matrix A.

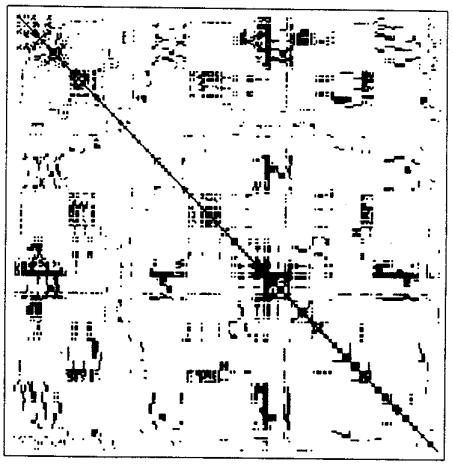

Figure 4. Nonzero pattern of AA^T without ordering.

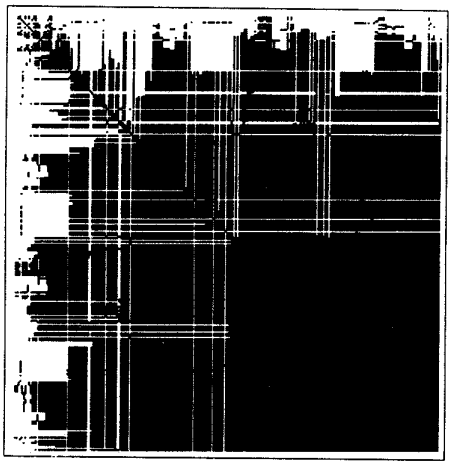

Figure 5. Nonzero pattern of LU factors without ordering.

product of the number of diagonal entries in the pivot row and column. It can be regarded as an attempt to minimize the number of arithmetic operations in the next stage of the Gaussian elimination algorithm, or as an attempt to select the pivot column that introduces the least *fill-in* in the next stage. This local minimization strategy does not guarantee a global minimum for either the amount of *fill-in* or total number of arithmetic operations performed during Gaussian elimination.

The specialization of the Markowitz ordering heuristic for symmetric matrices results in the *minimum degree* ordering heuristic.[35,39] It has received much attention in the literature and its analysis has produced an interesting graph theoretic interpretation of the matrix ordering problem.[14,15,36] At each stage of the Gaussian elimination procedure, the *minimum degree* ordering heuristic performs symmetric row and column interchanges so that the next pivot row is, among the rows in the portion of the matrix still to be factored, the one with the minimum number of nonzero entries. Efficient implementations of this heuristic make it the standard technique in linear equation solvers. The main feature that distinguishes different implementations is a *tie-breaking* strategy used to select the next pivot row among all the rows matching the minimum number of nonzero entries. The possible positive effect of using the *minimum degree* ordering heuristic is illustrated by Figures 6 and 7. They display the nonzero patterns for B_k and the corresponding LU factors, after permuting the rows of the linear programming coefficient matrix according to this ordering heuristic.

Another ordering heuristic results from the effort of trying to reduce the amount of *fill-in* further than the Markowitz criterion. The *minimum local fill-in* ordering heuristic selects, at each stage of the Gaussian elimination procedure, the pivot element that introduces the minimum amount of *fill-in*. This heuristic was suggested by Markowitz,[29] and by Tinney and Walker[39] where it is designated as the *minimum deficiency* algorithm. Compared to the *minimum degree* ordering heuristic, Figures 8 and 9 illustrate the potential for further reductions in the number of *fill-in* elements and the number of operations required by Gaussian elimination. Of course, this ordering heuristic does not necessarily produce a better ordering. Duff, Erisman and Reid[10] illustrate this point with an example where the *minimum degree* heuristic produces an optimal ordering while the *minimum local fill-in* heuristic does not. From an experimental point of view, Duff and Reid[7] conducted a comparison between the two ordering heuristics in the case of unsymmetric matrices and notice that the *minimum local fill-in* ordering performed marginally better. The *minimum local fill-in* heuristic is considerably more expensive, as we must update the sparsity pattern whenever we examine a pivot candidate. Due to its higher computational cost even in a sophisticated implementation, the comparison study rejects the minimum local *fill-in* ordering heuristic.

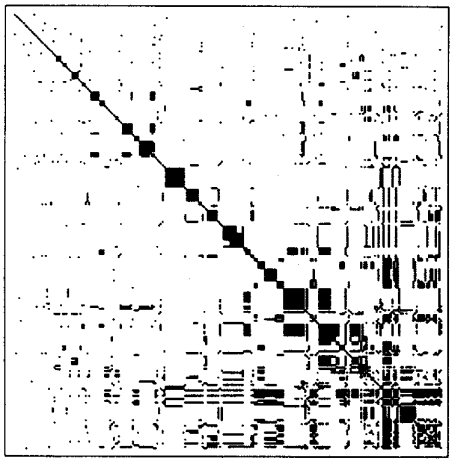

Figure 6. Nonzero pattern of AA^T after ordering (*minimum degree* ordering heuristic).

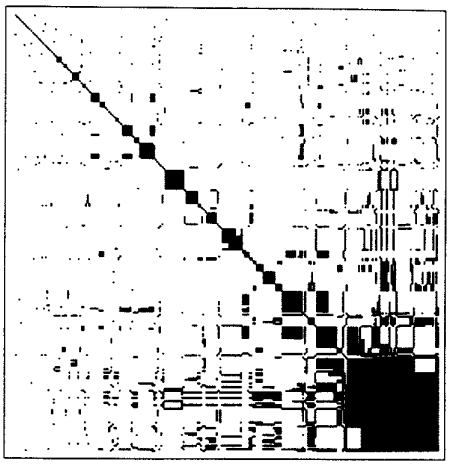

Figure 7. Nonzero pattern of LU factors after ordering (*minimum degree* ordering heuristic).

Our motivation for introducing the *minimum local fill-in* heuristic into this discussion is straightforward. Karmarkar's algorithm solves a sequence of systems of linear equations sharing an identical nonzero structure. Therefore, the ordering procedure will be executed only once at the beginning of the algorithm, and the resulting permutation remains valid for the remaining iterations. By contrast, the Gaussian elimination procedure is repeated in every iteration of the linear programming algorithm, and any computational savings achieved by a better ordering is multiplied by the total number of iterations of the algorithm. Except for anomalous test problems, the effort of ordering a matrix according to

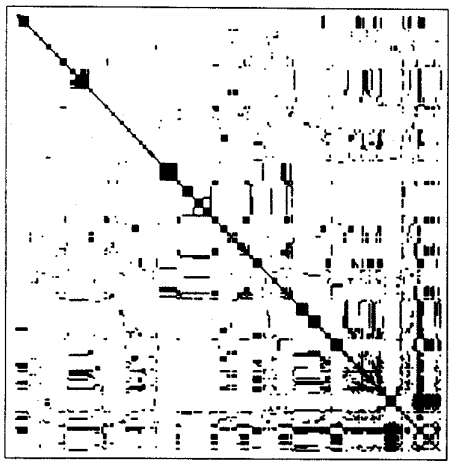

Figure 8. Nonzero pattern of AA^T after ordering (*minimum local fill-in* ordering heuristic).

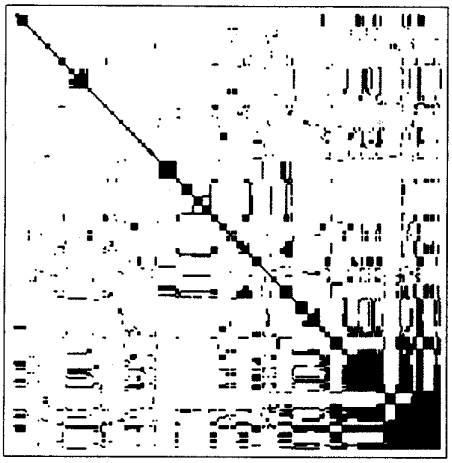

Figure 9. Nonzero pattern of LU factors after ordering (*minimum local fill-in* ordering heuristic).

either heuristic does not constitute a significant portion of the algorithm's total computational effort. Consequently, the extra effort involved in performing the *minimum local fill-in* heuristic does not impact significantly the algorithm's running time. The computational experiments reported in [5] compare the use of two ordering heuristics with the implementation of Karmarkar's algorithm discussed here. For linear programming test problems from the NETLIB collection,[12] it reports savings of up to 36% in total running time in favor of the *minimum local fill-in* heuristic.

4. Interpretative Procedure for Sparse Gaussian Elimination

The straightforward use of existing sparse linear system software modules in the implementation of Karmarkar's algorithm ignores one of its fundamental computational characteristics. The system of linear equation solved at each iteration of the linear programming algorithm is such that its nonzero structure remains unchanged throughout the algorithm. As we presented before in (2.1), at each iteration k of the linear programming algorithm, we solve the following system:

$$(AD_k^2A^T)d_y = b \tag{4.1}$$

From iteration to iteration, only the diagonal scaling matrix D_k changes, which does not affect the nonzero structure of $AD_k^2A^T$.

An *interpretative* procedure for Gaussian elimination symbolically analyzes at first the nonzero pattern of the system of equations, without consideration to its actual numerical values. This initial phase, called *symbolic factorization*, collects information to be used during Gaussian elimination. The actual LU factors for $AD_k^2A^T$ are computed at each iteration of the linear programming algorithm by performing a *numerical factorization*. Since we are solving a sequence of systems of linear equations, all with identical nonzero structure, a single *symbolic factorization* step is performed in the beginning of the algorithm. The extra effort expended in the *symbolic factorization* stage is usually justified by the savings accrued at each iteration of the linear programming algorithm.

Most implementations of the Gaussian elimination procedure for sparse positive definite matrices share a similar blueprint. In a preparatory stage, the code obtains a suitable sparsity ordering and builds a *static* data structure, including the *fill-in* elements that will be created during the process. Unlike some implementations of the symbolic/numerical factorization schemes,[14] an *interpretative* code also compiles a list of the operations to be performed during the Gaussian elimination process. The *numerical factorization* stage consists of a simple procedure that scans this list of operations, applying them to $AD_k^2A^T$ computed at each iteration k of Karmarkar's algorithm, and yielding the corresponding LU factors. The objective is to relieve as much as possible the *numerical factorization* procedure of costly indirect memory access and other overhead operations. The solution for the system of linear equation is obtained from the LU factors as usual, by means of *forward* and *back substitution* operations.

The *interpretative* version of the Gaussian elimination procedure is an important feature of the implementation of Karmarkar's algorithm presented by Adler et al.[1] This approach is motivated by a Gaussian elimination implementation that generates *loop-free* FORTRAN codes tailored to a given sparse matrix.[21] In this fashion, all the overhead operations related to indirect access of elements in the data structure are avoided. The *symbolic factorization* step involves the

compilation of the resulting FORTRAN code. The drawback hidden in the *loop-free* approach is the prohibitive memory requirement necessary to store the resulting FORTRAN code, even for moderately sized matrices.[10]

The *interpretative* approach[3] follows a natural extension of the *loop-free* code idea. In the procedure SGE, described in Pseudo-code 2, each update operation on the super-diagonal elements of factor U in the symmetric Gaussian elimination procedure is:

$$u_{ij} := u_{ij} - u_{qj}u_{qi}/u_{qq}. \tag{4.2}$$

Instead of creating an actual line of FORTRAN code that executes this operation, we collect the locations of elements u_{ij}, u_{qj}, u_{qi} and u_{qq} into an array of pointers called the *operation list*. Operations on the diagonal elements are stored in a similar manner. This operation list is used as data by a simple FORTRAN subroutine that executes the arithmetic computations as indicated by the list pointers. This scheme reduces the storage requirement incurred in the *loop-free* approach, still making use of some indirect addressing to access matrix elements, but eliminating a substantial portion of the overhead operations. Still, the *operation list* requirements can grow dramatically fast with the size of the problem. We describe in Section 6 a technique that effectively reduces the storage requirements by performing the LU factorization with the aid of a *dense window* data structure.

As exemplified by Marsten et al.[30] and Monma and Morton,[32] the use of such *interpretative* scheme is not essential for a successful implementation of interior point algorithms. Gay[13] compares three different Gaussian elimination implementations, including two versions of the *interpretative* approach. This study observes that a scheme designed without the use of an *operation list* performs within a factor of two of the optimal floating-point operation rate, leaving little room for improvement. His numerical experiments report speed-ups of up to 22% when an *interpretative* scheme is used, at the expense of large memory use. We argue that by incorporating a *dense window* data structure we benefit from the improved performance offered by the use of an *operation list* without massive memory use.

In our implementation, the *symbolic factorization* step builds a similar operation list. As illustrated in Figure 10, we add to data structure {**diag, iaat, jaat, aat**} arrays that describe the list of operations performed during Gaussian elimination. Referring to (4.2), u_{qq} is directly accessible in FORTRAN array **diag**, u_{qi} is identified by scanning the nonzero elements of the current pivot row q. The *operation list* is built in two levels of pointers, indicating the operations necessary to merge rows q and i corresponding to lines 7–11 in Pseudo-code 3. The FORTRAN arrays used in the representation

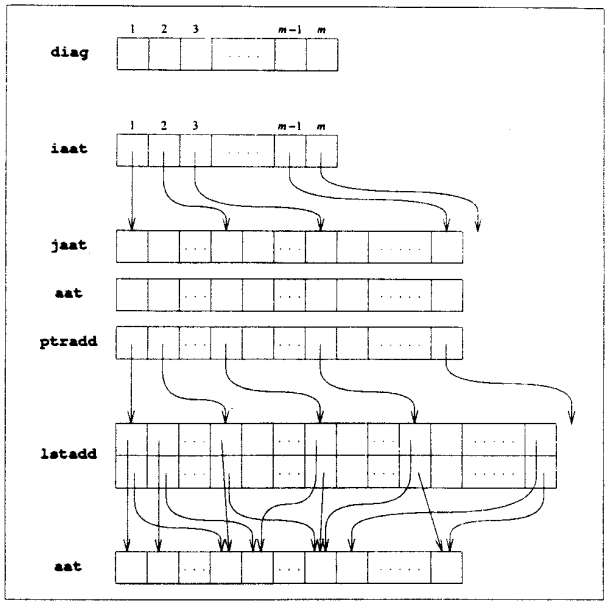

Figure 10. Data structure for factorization operation list.

of the *operation list* are the following:

ptradd contains pointers to the first position of the *operation list* involving each element of U. Each **ptradd**(i) element corresponds to a super-diagonal element of U stored in **aat**(i), and can be scanned with the aid of the pointers contained in array **iaat**.

lstadd contains pointers to the elements involved in each operation in the Gaussian elimination procedure. **lstadd** is a two-dimensional array, where each position points to two super-diagonal elements of U stored in **aat**. The first dimension contains the index of array **aat** storing the value of u_{ij}, and the second dimension contains the same information for the value of u_{qj}.

PSEUDO-CODE 4—INTERPRETATIVE SPARSE SYMMETRIC GAUSSIAN ELIMINATION

```
procedure ISSGE(m, diag, iaat, jaat, aat, ptradd, lstadd)
1  for q = 1, ..., m − 1 →
2      for i = iaat(q), ..., iaat(q + 1) − 1 →
3      l := jaat(i);
4      diag(l) := diag(l) − aat(i)²/diag(q);
5      for j = ptradd(i), ..., ptradd(i + 1) − 1 →
6        aat(lstadd(1, j))
           := aat(lstadd(1, j)) − aat(i) ×
           aat(lstadd(2, j))/diag(q);
7      rof
8    rof
9  rof
end ISSGE;
```

Procedure ISSGE in Pseudo-code 4 encapsulates the *interpretative* Gaussian elimination algorithm. The *numerical factorization* stage consists mostly of scanning the *operation list* (inner-loop in lines 5–7) and performing the Gaussian elimination computations updating the super-diagonal elements of factor U. Gay[13] observes that **lstadd** can be reduced to a one-dimensional array. In a modified implementation, procedure ISSGE scans the *operation list* for the location in array **aat** storing the value of u_{ij} while the location of u_{qj} is obtained by concurrently scanning the nonzero elements of the pivot row.

5. Building and Updating $AD_k^2A^T$

The solution of a system of linear equations dominates the theoretical complexity of Karmarkar's algorithm. However, in practice, forming the $AD_k^2A^T$ matrix at each iteration k of the linear programming algorithm consumes a significant portion of the total computational effort. In this section, we describe a procedure that reduces the effort of building this matrix at each iteration by performing some of the computations in the preparatory stage of the algorithm. In addition, further reduction in the computational effort can be achieved by approximating $AD_k^2A^T$, and updating it from iteration to iteration.

As before, we define

$$B_k = AD_k^2A^T. \tag{5.1}$$

Each super-diagonal element of B_k is expressed as

$$B_k(i, j) = \sum_{l=1}^{n} A(i, l) \times A(j, l) \times D_k^2(l, l) \quad \text{for } 1 \leq i < j \leq n. \tag{5.2}$$

Since in (5.2) only the scaling matrix D_k changes from iteration to iteration, we compute every nonzero outer-product $A(i, l) \times A(j, l)$ once in the beginning of the linear programming procedure.

As depicted in Figure 11, the following arrays are added to data structure {**diag, iaat, jaat, aat**} with the information associated with the outer-products:

ptrprd contains pointers to the first element of outer-product list involving elements in each column of A.

lstprd contains the computed outer-products sorted in increasing order of the corresponding column index l.

lstdst for each entry in **lstprd**, **lstdst** contains a pointer to the location of element $B_k(i, j)$ in array **aat**.

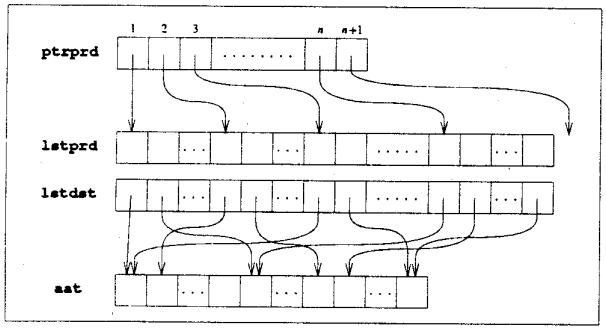

Figure 11. Data structure for outer-product list.

PSEUDO-CODE 5—BUILDING $AD_k^2A^T$

```
procedure BLDMTR(m, n, D_k, diag, ia, ja, a, iaat,
                 jaat, aat, ptrprd, lstdst, lstprd)
1   for i = 1, ..., m − 1 →
2     diag(i) := 0;
3     for j = iaat(i), ..., iaat(i + 1) − 1 →
      aat(jaat(j)) := 0 rof;
4   rof;
5   diag(m) := 0;
6   for i = 1, ..., n →
7     for j = ia(i), ..., ia(i + 1) − 1 →
8       diag(ja(j)) := diag(ja(j)) + a(j) × a(j) ×
        D_k(i, i)^2;
9     rof;
10    for j = ptrprd(i), ..., ptrprd(i + 1) − 1 →
11      aat(lstdst(j)) := aat(lstdst(j)) + lstprd(j)
        × D_k(i, i)^2;
12    rof
13  rof
end BLDMTR;
```

Using the list of outer-products, B_k is built at each iteration of the linear programming algorithm by procedure BLDMTR outlined in Pseudo-code 5. This procedure produces as output the data structure required as input by procedure BSSGE.

In an effort to reduce the computational complexity of the linear programming algorithm, Karmarkar[25] suggests a scheme where an approximate scaling matrix is determined at each iteration. Using this approximate scaling matrix, the search direction can be obtained through a series of rank-1 updates on the coefficient matrix of the system of linear equation used in the previous iteration. As a consequence, each iteration can be performed in an average of $O(m^{2.5})$ arithmetic operations, compared to the $O(m^3)$ arithmetic operations required to solve a system of linear equations. In practice, we do not implement this scheme, obtaining the search directions by solving system (2.2) directly. However, although not improving on the algorithm's overall complexity, the computational effort of building the system of linear equations (2.2) can also be reduced by using the approximate scaling matrix.

At iteration k, instead of computing the exact scaling matrix

$$D_k = diag(1/v_1^k, \ldots, 1/v_n^k), \tag{5.3}$$

as in line 4 of Pseudo-code 1, we use an approximate scaling matrix $\tilde{D}_k$. Starting with

$$\tilde{D}_1 = D_1, \tag{5.4}$$

we compute the elements of $\tilde{D}_k$ by updating elements in the previous approximate scaling matrix whenever they differ substantially from the corresponding elements in the exact scaling matrix. The updates are performed as follows:

$$\tilde{D}_k(i, i), = \tag{5.5}$$

$$\begin{cases} \tilde{D}_k(i, i), \\ \quad \text{if } | D_k(i, i) - \tilde{D}_{k-1}(i, i) | / | \tilde{D}_{k-1}(i, i) | < \epsilon_u \\ D_k(i, i), \\ \quad \text{if } | D_k(i, i) - \tilde{D}_{k-1}(i, i) | / | \tilde{D}_{k-1}(i, i) | \geq \epsilon_u, \end{cases}$$

for a given $\epsilon_u > 0$. In practice, we use $\epsilon_u = 0.1$. After computing the current approximate scaling matrix according to (5.5), we define

$$\Delta = \tilde{D}_k^2 - \tilde{D}_{k-1}^2. \tag{5.6}$$

Then, we write the following update expression:

$$\tilde{B}_k = A\tilde{D}_k^2 A^T = A\tilde{D}_{k-1}^2 A^T + A\Delta A^T, \tag{5.7}$$

where $\tilde{B}_k$ denotes the matrix replacing B_k in system (2.2).

This approach computes the $\tilde{B}_k$ matrix by adding the second term of the right hand side of (5.7) to the matrix used in the previous iteration. As the algorithm progresses, the v^k iterates converge to the optimal dual slack values. The components of v^k converging to nonzero values display small variation towards the last iterations of the algorithm. The corresponding diagonal elements of Δ will be zero, excluding the associated columns of the coefficient matrix A from the update operation described in (5.7). The practical effect of this strategy is illustrated in Figures 12–14, where we display, as the algorithm progresses, the proportion of excluded columns and the CPU time required to build the $\tilde{B}_k$ matrix (relative to the CPU time to build the matrix from scratch).

In Figure 12, we examine the behavior of solving phase II of problem *Bandm* from the NETLIB collection. After removing trivial rows and columns, this test problem has 246 rows and 401 columns. The optimal solution found by Algorithm I displays a small degree of dual degeneracy, with 157 nonzero dual slack values. Since this does not represent a substantial portion of the total number of columns, we observe modest savings in building matrix $\tilde{B}_k$. A more dramatic effect is

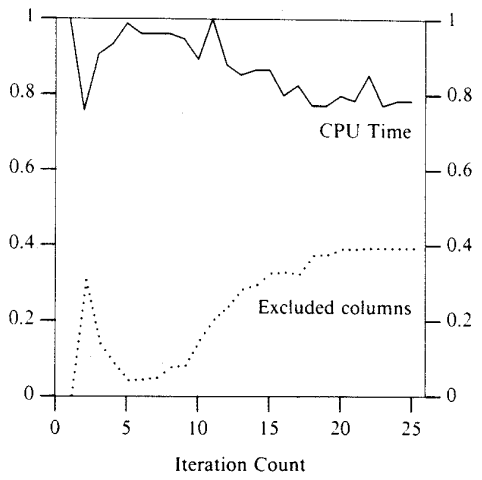

Figure 12. Excluding columns in updating $A\tilde{D}_k^2 A^T$ (phase II of problem *Bandm*).

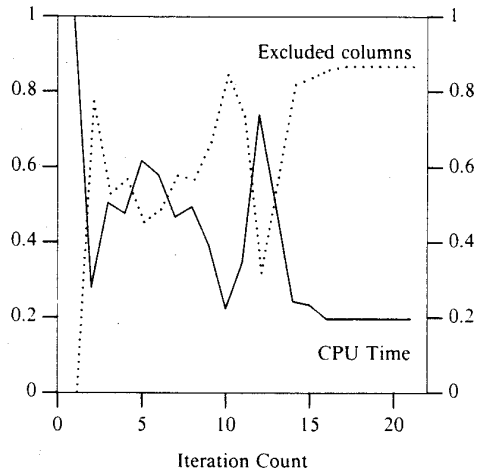

Figure 13. Excluding columns in updating $A\tilde{D}_k^2 A^T$ (phase II of problem *Scsd6*).

displayed in Figures 13 and 14. Problem *Scsd6*, also from the NETLIB collection, has 147 rows and 1350 columns, with 1168 nonzero dual slack values in the optimal solution found by Algorithm I. As the algorithm progresses, updating $\tilde{B}_k$ requires substantially less effort by virtue of the large portion of columns excluded from the update operation. Also, for problems with a nonempty dual feasible interior, the Phase I stage of the algorithm approaches a dual point where all dual slacks are positive. For problem *Bandm*. Figure 14 shows the increasing number of excluded columns and decreasing CPU time required to build $\tilde{B}_k$.

At each iteration k, Algorithm I computes a tentative primal solution, as indicated in line 9 of Pseudo-code 1, by substituting $\tilde{D}_k$ for D_k. An interesting observed property of this approximation scheme

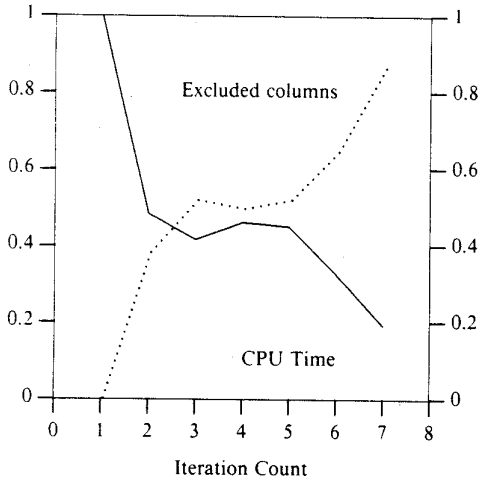

Figure 14. Excluding columns in updating $A\tilde{D}_k^2A^T$ (phase I of problem *Bandm*).

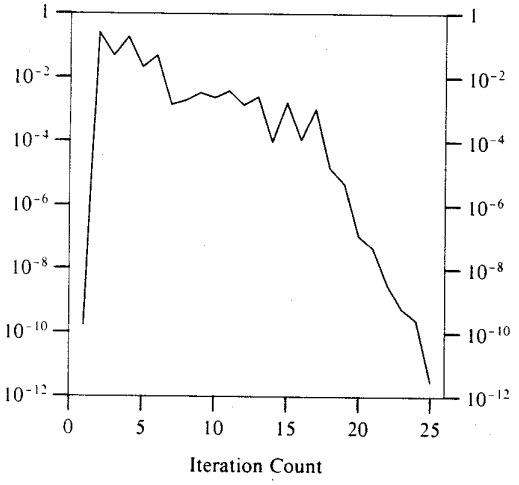

Figure 15. Solution accuracy for system of linear equations (phase II of problem *Bandm*).

is that as the algorithm converges, the solution to the approximate system

$$(A\tilde{D}_k^2A^T)\tilde{d}_y = b \tag{5.8}$$

approaches the solution to the exact system

$$(AD_k^2A^T)d_y = b. \tag{5.9}$$

An example of this is illustrated in Figure 15 for problem *Bandm*, where we plot

$$\| (AD_k^2A^T)\tilde{d}_y - b \| \tag{5.10}$$

as a function of iteration k.

PSEUDO-CODE 6—UPDATING THE $AD_k^2A^T$

```
procedure UPDMTR(m, n, Δ, ia, ja, diag, a, iaat, jaat,
                 aat, ptrprd, lstdst, lstprd)
1   for i = 1, ..., n →
2     if Δ(i, i) ≠ 0 →
3       for j = ia(i), ..., ia(i + 1) − 1 →
4         diag(ja(j)) := diag(ja(j)) + a(j) × a(j)
          × Δ(i, i);
5       rof;
6       for j = ptrprd(i), ..., ptrprd(i + 1) − 1 →
7         aat(lstdst(j)) := aat(lstdst(j)) +
          lstprd(j) × Δ(i, i);
8       rof
9     fi
10  rof
end UPDMTR;
```

We present procedure UPDMTR in Pseudo-code 6, which updates matrix $\tilde{B}_k$ at each iteration. Assume that diagonal matrix Δ was computed according to (5.6), and the matrix approximation for the previous iteration is initially stored in data structure {**diag, iaat, jaat, aat**}. Then, for each column i of the coefficient matrix A, lines 2–9 are executed whenever the update matrix Δ diagonal element is nonzero, replicating Pseudo-code 5 modified to update the diagonal and off-diagonal elements of $\tilde{B}_k$.

6. Using a Dense Window

In the *interpretative* procedure for Gaussian elimination described in Section 4, we observe a drawback. The operation list compiled during the *symbolic factorization* stage can grow dramatically with the size of the coefficient matrix. One possible way to overcome this deficiency arises from a simple observation. As indicated in Figure 16, after the matrix is ordered towards

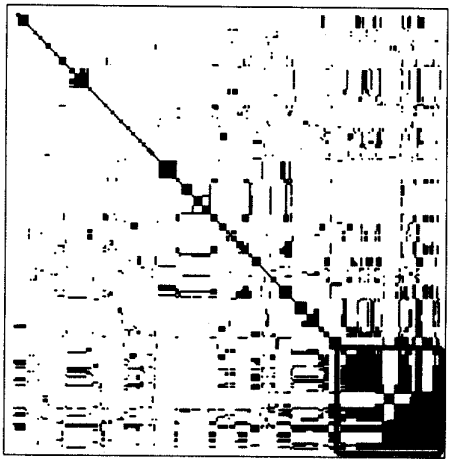

Figure 16. Nonzero pattern of LU factors with a dense window.

sparsity, the nonzero elements of the corresponding LU factors are clustered in the lower right corner forming a dense submatrix. The representation of the LU factors during Gaussian elimination can be split in two parts. Unlike the sparse portion which is represented as usual, the dense sub-matrix is stored in a dense array, including positions for zero elements. Consequently, positions in this *dense window* can be accessed directly without information on the nonzero pattern of the LU factors.

The data structure storing the LU factors including a dense window consists the data structure {**diag**, **iaat**, **jaat**, **aat**} with extensions. During the *symbolic factorization* stage, we select a row m^d where the dense representation for the LU factors is started. As displayed in Figure 17, we modify array **aat** and add two auxiliary pointer arrays as follows:

aat contains the super-diagonal nonzero elements of U stored row-wise, for rows with sparse representations. For rows in the *dense window*, the super-diagonal elements of U are stored row-wise as dense arrays. After the numerical factorization scheme, **aat** reverts to the full sparse representation.

fstdns contains pointers to the first off-diagonal entry in each row with a column index in the *dense window*. This assumes that **jaat** is ordered, within each row, by column index.

dnsptr contains pointers to the first super-diagonal elements of each row with dense representation. More specifically, during Gaussian elimination, element u_{ij} is stored in **aat(dnsptr**(i) + $j - i$). These pointers are kept for programming convenience only, since they could be recomputed as needed based on the last pointer for the sparse portion of the representation.

PSEUDO-CODE 7—INTERPRETATIVE GAUSSIAN ELIMINATION WITH DENSE WINDOW

```
procedure DISSGE(m, m^d, diag, iaat, jaat, aat, ptradd,
                                    1stadd, fstdns, dnsptr);
1   for q = 1, ..., m^d - 1 →
2     for i = iaat(q), ..., fstdns(q) - 1 →
3       l := jaat(i);
4       diag(l) := diag(l) - aat(i)^2/diag(q);
5       for j = ptradd(i), ..., ptradd(i + 1) - 1 →
6         aat(1stadd(1, j))
            := aat(1stadd(1, j)) - aat(i) ×
          aat(1stadd(2, j))/diag(q);
7       rof;
8     rof;
9     for i = fstdns(q), ..., iaat(q + 1) - 1 →
10      l := jaat(i);
11      diag(l) := diag(l) - aat(i)^2/diag(q);
12      for j = i + 1, ..., iaat(q + 1) - 1 →
13        p^l := dnsptr(l) + jaat(j) - l;
14        aat(p^l) := aat(p^l) - aat(j) × aat(i)/diag(q);
15      rof;
16    rof;
17  rof;
18  for q = m^d, ..., m - 1 →
19    for i = iaat(q), ..., iaat(q + 1) - 1 →
20      p^i := dnsptr(q) + jaat(i) - q;
21      l := jaat(i);
22      diag(l) := diag(l) - aat(p^i)^2/diag(q);
23      for j = i + 1, ..., iaat(q + 1) - 1 →
24        p^j := dnsptr(q) + jaat(j) - q;
25        p^l := dnsptr(l) + jaat(j) - l;
26        aat(p^l) := aat(p^l) - aat(p^j) × aat(p^i)/diag(q);
27      rof;
28      aat(i) = aat(p^i);
29    rof
30  rof
end DISSGE;
```

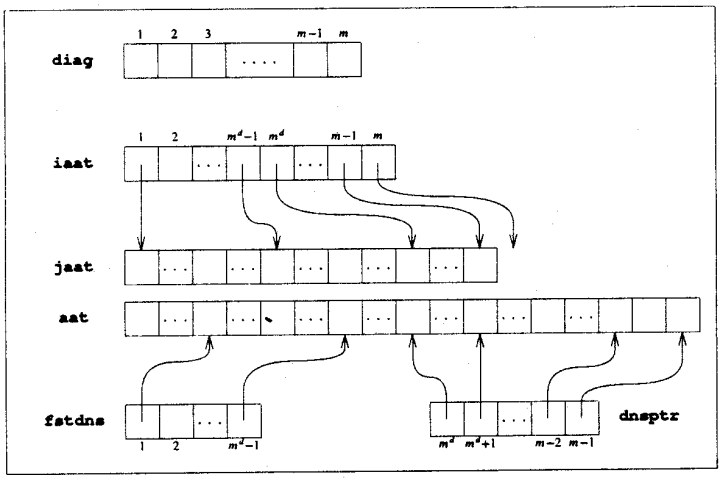

Figure 17. Data structure for LU factors with a dense window.

Based on the data structure described above, we extend procedure ISSGE so that the *operation list* will pertain only to elements stored in the sparse portion of the representation. Procedure DISSGE in Pseudo-code 7 incorporates the *dense window* representation during Gaussian elimination. The data structure described in Figure 17 is supplied as input, initialized with the super-diagonal values of matrix $AD_k^2A^T$ for the current iteration of the linear programming algorithm. Partitioned into two main sections, the code in lines 1–17 perform elementary operations based on pivot rows with sparse representation. The remainder of the code (lines 18–30) performs similar operations based on pivot rows with dense representation. This procedure supplies, as output, the U factor stored in the original data structure {**diag, iaat, jaat, aat**}.

Similar to the behavior of the *operation list*, in the methodology introduced in Section 5 for building the B_k matrix, the size of the list of outer products can grow too fast with the size of the linear program. One solution to this problem is to limit the size of arrays **lstprd** and **lstdst** by the use of the same *dense window* used in the *interpretative* Gaussian elimination. In the preparatory stage of Algorithm I, we compute only outer-products $A(i, l) \times A(j, l)$ associated with elements $B_k(i, j)$ in rows with sparse representation. These outer-products are stored in array **lstprd** as described in Section 5. The outer-products associated with elements in rows with dense representations are computed when building B_k in each iteration of the linear programming algorithm. In this situation, element $B_k(i, j)$ associated with outer-product $A(i, l) \times A(j, l)$ is stored in the dense portion of array **aat**, in a position determined solely by row indices i and j.

When computing the outer-products, we scan the nonzero elements of each column of A, using only the elements in rows with dense representation. Assuming that the column-wise representation of A is ordered according to row indices, we modify data structure {**ia, ja, a**} to include information linking the sparse representation of the linear programming coefficient matrix with the dense window representation used during Gaussian elimination. Figure 18 depicts the modified column-wise representation of A. It includes an array of pointers **iadns** that indicates the location of the first element in each column of A corresponding to a row with dense representation.

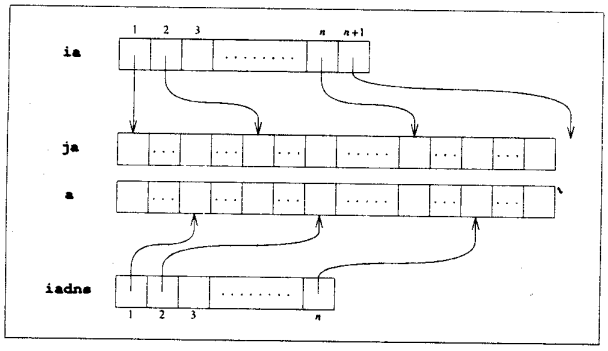

Figure 18. Data structure for coefficient matrix with dense window.

PSEUDO-CODE 8—UPDATING $AD_k^2A^T$ WITH DENSE WINDOW REPRESENTATION

```
procedure DNSMTR(m, n, Δ, diag, ia, ja, a, iaat, jaat, aat,
                                          ptrprd, lstdst, lstprd)
1   for i = 1, ..., n →
2     if Δ(i, i) ≠ 0 →
3       for j = ia(i), ..., ia(i + 1) − 1 →
4         diag(ja(j)) := diag(ja(j)) + a(j) × a(j) ×
          Δ(i, i);
5       rof;
6       for j = ptrprd(i), ..., ptrprd(i + 1) − 1 →
7         aat(lstdst(j)) := aat(lstdst(j)) + lstprd(j) ×
          Δ(i, i);
8       rof
9       for j1 = iadns(i), ..., ia(i + 1) − 1;
10        l := ja(j1);
11        for j2 = j1 + 1, ia(i + 1) − 1;
12          p := dnsptr(l) + ja(j2) − l;
13          aat(p) := aat(p) + a(j1) × a(j2) × Δ(i, i);
14        rof
15      rof
16    fi
17  rof
end DNSMTR;
```

Procedure DNSMTR described in Pseudo-code 8 incorporates the dense window approach to procedure UPDMTR. As before, we assume that data structure {**diag, iaat, jaat, aat**} contains $\tilde{B}_{k-1}$. The $\tilde{B}_k$ matrix is computed by updating, if necessary, the contribution of each column of A.

7. Preprocessing of the Linear Programming Coefficient Matrix

Preprocessing operations are intended to eliminate easily detectable redundant rows and dominated columns (redundant rows in the dual problem) and to reduce the density of the coefficient matrix. Linear programming input data is frequently presented by the user in a form that is ill-suited for direct solution. Often, there are columns or rows where all elements are zero, redundant constraints, obvious infeasibilities, null or fixed-value variables and dominated columns.

In the preprocessing phase of our code we identify several of the above conditions that are detectable by a

simple inspection of the input data. There is a twofold justification for the relevance of this procedure implemented in subroutine CLEAN. On one hand, by eliminating some rows and columns, the solution procedure will effectively deal with a smaller problem. In addition, we eliminate some numerical stumbling blocks with the removal of null variables and redundant constraints. The procedure follows the outline below:

(i) Identify and remove all rows that have all coefficient entries with the same sign and zero right hand side, setting all variables that appear in these constraints to zero.
(ii) Identify any row that has all coefficient entries with the same sign and right hand side of the opposite sign. If any is found, declare the problem infeasible.
(iii) For all rows with only one entry, set the corresponding variable to the right hand side value or declare the problem infeasible if the value is negative. Update the right hand side whenever a fixed variable is found.

The above process is repeated until no further change occurs.

In practice, it is usually possible to reformulate a given linear program as an equivalent problem with a coefficient matrix of reduced density. The procedure described below eliminates nonzero elements in the linear programming coefficient matrix A by means of elementary row operations. Any linear programming algorithm can potentially profit from the reduced density in coefficient matrix. In the case of Karmarkar's algorithm, the benefits of a sparser coefficient matrix are immediate. As long as the lower density level is replicated in matrix $AD_k^2A^T$, the computational effort per iteration will be reduced and no major change is expected in the total number of iteration or convergence properties. On the other hand, for strictly pivoting algorithms, we cannot easily evaluate the effect of reduced density. The cost of updating the basis inverse is likely to be reduced. However, the total number of iterations can change in any direction, as we cannot predict the behavior of other operations such as *pricing*.

Given a system of linear equations, finding an equivalent system with minimum number of nonzero elements in its coefficient matrix is referred to as the *Sparseness Problem*. Unless some simplifying assumptions are added, this problem is NP-Hard.[24] We describe below a local sparsity-increasing heuristic, implemented in subroutine SPARSE. In each step of SPARSE, the heuristic selects a set of row operations that yields the maximal local reduction in the number of nonzero elements in the linear programming coefficient matrix, producing an equivalent linear program. After each step, subroutine CLEAN is applied to the resulting linear program attempting to eliminate newly found null or fixed-value variables.

The main loop of SPARSE is repeated until only an insignificant number of nonzero elements of A can be canceled. In our code, we consider the reduction to be insignificant if less than one percent of the nonzero elements are canceled. The main loop consists of four phases:

(i) Build Pivot Table (*PT*).
(ii) Sort Pivot Table.
(iii) Perform elementary row operations according to sorted *PT*.
(iv) Clean the resulting matrix.

In phase (i), for every pair of rows (i, j) of the coefficient matrix A, the procedure scans the nonzero elements of row j and identifies the pivot element that cancels the greatest number of nonzero elements in row i. This operation is called the *maximal elementary row operation* of row j on row i. The column index of the pivot element that corresponds to the maximal elementary row operation is c_p and the number of canceled elements is $\#c$. An elementary row operation can be compactly represented by the tuple $[i, j, c_p, \#c]$. The list of all maximal elementary row operations is called the Pivot Table (*PT*). In phase (ii), *PT* is sorted in decreasing order of the number cancellations $\#c$.

In phase (iii), the elementary row operations are performed in the order indicated by the sorted *PT*. This is achieved by executing procedure CANCEL, described in Pseudo-code 9.

Finally, in phase (iv), procedure CLEAN is applied to the resulting linear program, eliminating null and fixed-value variables uncovered by the elementary row operations.

PSEUDO-CODE 9—PERFORM ELEMENTARY ROW OPERATION

```
procedure CANCEL(n, A, list PT)
1   for k = 1, ..., m → marked(k) = false rof;
2   do PT ≠ [] →
3     [i, j, c_p, #c] := PT(1);
4     PT := PT[2..];
5     if not marked(i) and not marked(j) →
6       for k = 1, ..., n →
7         a_ik := a_ik − (a_ic_p / a_jc_p) a_jk;
8       rof;
9       marked(i) := true
10    fi
11  od
end CANCEL;
```

Figures 19–23 illustrate the effect of input matrix preprocessing for a corporate level production planning

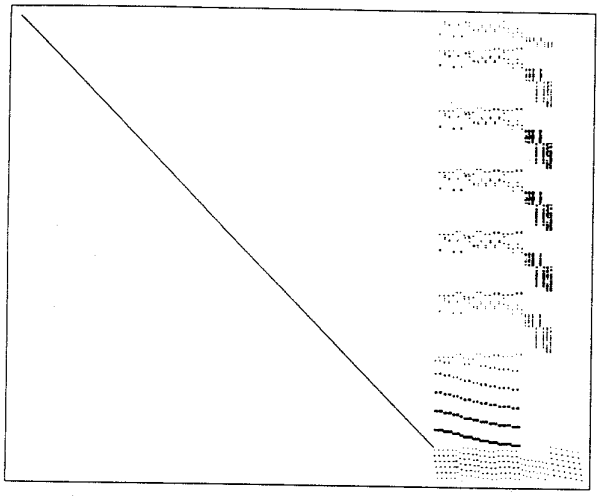

Figure 19. Nonzero pattern of coefficient matrix A before CLEAN.

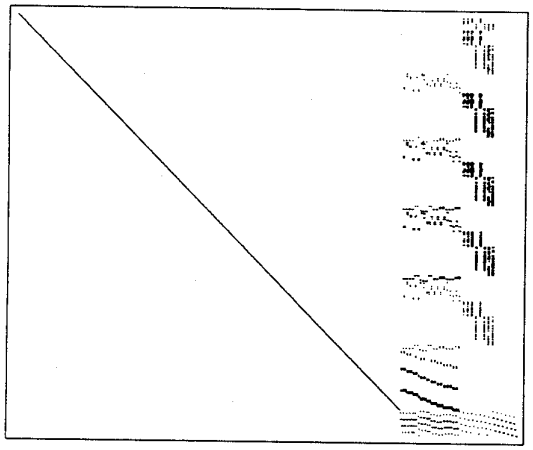

Figure 20. Nonzero pattern of coefficient matrix A after CLEAN nonz(A) = 2092.

model for the semiconductor industry.[27] Figure 19 depicts the input matrix before preprocessing. Figure 20 shows the resulting matrix after CLEAN is carried out. Several fixed variables are identified and the matrix dimension is reduced. At this stage there are 2092 nonzero elements in A. Figure 21 shows the resulting equivalent input matrix after SPARSE is applied. The number of nonzero elements has fallen to 1728, representing a reduction of 17.4%. Figures 22 and 23 show the difference between LU factors when SPARSE is not activated (Figure 22) and when it is (Figure 23). The number of nonzero entries in L falls from 11933 to 7896, a 33.8% reduction. Tables I and II illustrate applying SPARSE on a set of linear programming test problems that includes several linear programs publicly available through NETLIB.[12]

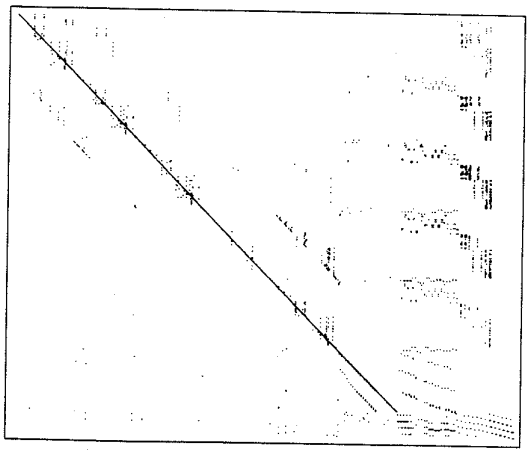

Figure 21. Nonzero pattern of coefficient matrix A after SPARSE nonz(A) = 1728.

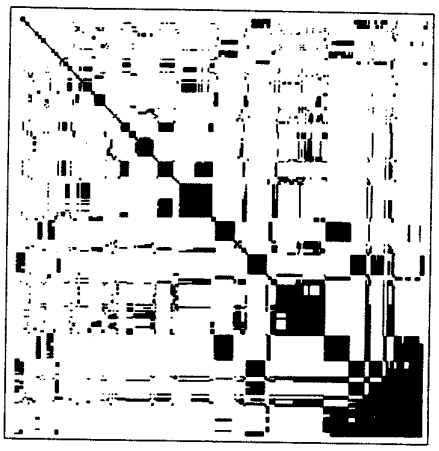

Figure 22. Nonzero pattern of LU factors before SPARSE nonz(L) = 11933.

Figure 23. Nonzero pattern of LU factors after SPARSE nonz(L) = 7896.

TABLE I
Density Reduction

Problem	nonz(A)	nonz($s(A)$)	% Reduction	nonz(L)	nonz($s(L)$)	% Reduction
Afiro	102	102	0.00	107	107	0.00
AdLittle	417	393	5.76	404	383	5.20
Scagr7	457	434	5.03	734	731	0.41
Sc205	663	661	0.30	1182	1211	−2.45
Dsg	691	630	8.83	1226	940	23.33
Share2b	777	738	5.02	1026	1014	1.17
Share1b	1148	1060	7.67	1425	1287	9.67
MulTest	1283	1275	0.62	717	717	0.00
Scorpion	1393	1300	6.68	2324	2297	1.16
Scagr25	1717	1604	6.58	2948	2945	0.10
Sctap1	1872	1420	24.15	2667	1985	25.57
Brandy	1927	1811	6.02	2850	2797	1.86
Beaconfd	2062	840	59.26	1727	1107	35.90
Agg	2092	1728	17.40	11933	7896	33.83
Scsd1	2388	2388	0.00	1392	1391	0.07
Israel	2443	1705	30.21	13744	12921	5.99
E226	2558	2257	11.77	3414	3223	5.59
Scfxm1	2655	2478	6.67	4963	4541	8.50
Scrs8	3147	2948	6.32	5134	5161	0.52
Ship04s	3899	3891	0.21	3134	3130	0.13
Scsd6	4316	4316	0.00	2545	2545	0.00
Forplan	4608	2202	52.21	3688	2546	44.85
Agg2	4728	3140	33.59	21535	18306	14.99
Agg3	4744	3156	33.47	21535	18306	14.99
Ship08s	4929	4911	0.37	4112	4090	0.54
Pilot4	5248	5118	2.48	14115	14414	−2.12
Scfxm2	5315	4961	6.66	9791	9163	6.41
Ship04l	5771	5763	0.14	4158	4380	−5.07
Ship12s	5983	5965	0.30	5063	5079	−0.32
Fpk010	6021	5784	3.94	854	854	0.00
Sctap2	7334	5548	24.35	14870	12375	16.78
Scfxm3	7975	7444	6.66	14619	13791	5.66
Scsd8	8584	8584	0.00	5879	5879	0.00
Czprob	8993	8942	0.57	7059	7007	0.74
Ship08l	9480	9462	0.19	7128	7106	0.31
Sctap3	9734	7394	24.04	19469	16051	17.56
25fv47	10566	10016	5.21	34291	32190	6.13
Ship12l	12667	12649	0.14	9501	9493	0.08

Table I shows the reduction in nonzero entries of A and L obtained by using SPARSE. Columns 1–3 show, respectively, the number of nonzero elements in A after CLEAN but prior to SPARSE, after SPARSE and the corresponding percentage reduction. Columns 4–6 show the same for the Cholesky factor L.

In Table II IBM 3090 CPU times are shown. Column 1 lists CPU times for Algorithm I without SPARSE. Column 2 are the CPU times for running SPARSE alone. Column 3 shows the times for running Algorithm I (excluding the time taken by SPARSE). Column 4 is the percentage reduction in CPU time for optimization alone. Column 5 is the sum of columns 2 and 3, i.e., the total CPU time for Algorithm I with SPARSE. Column 6 is the total percentage reduction in CPU time. All CPU times are measured with the DATETM utility. The FORTRAN subroutines were compiled on the FORTVS compiler with options OPT(3), NOSYM and NOSDUMP.

A total of 38 test problems were run. In 34 there were positive reductions in the density of A. In 24 there was a greater than 5% reduction. In 10 the reduction was greater than 10%. The density reduction was greater than 20% in 8 problems; greater than 30% in 5 and greater than 50% in 2. The maximum reduction measured was 59.2%. In 28 cases there was a positive reduction in the density of L. In 17 cases the reduction was greater than 5%; in 9 cases it was greater than 10%; in 5 cases it was more than 20%; in 3 cases it was more than 30% and in one case it was 44.85%. However, in

TABLE II
Solution CPU Times (IBM 3090 sec)

Problem	No sparse	Sparse	Opt	% Reduction	Total	% Reduction
Afiro	0.03	0.01	0.02	33.33	0.03	0.00
AdLittle	0.12	0.01	0.12	0.00	0.13	−8.33
Scagr7	0.17	0.02	0.18	−5.56	0.20	−15.00
Sc205	0.29	0.01	0.29	0.00	0.30	−3.33
Dsg	0.32	0.03	0.23	28.13	0.26	18.75
Share2b	0.29	0.03	0.29	0.00	0.32	−9.38
Share1b	0.59	0.05	0.52	11.86	0.57	3.51
MulTest	0.30	0.03	0.31	−3.23	0.34	−11.76
Scorpion	0.51	0.03	0.50	1.96	0.53	−3.92
Scagr25	0.69	0.06	0.68	1.45	0.74	−6.76
Sctap1	0.85	0.09	0.62	27.06	0.71	19.72
Brandy	1.51	0.08	1.36	9.93	1.44	4.86
Beaconfd	0.69	0.32	0.36	47.83	0.68	1.47
Agg	7.86	0.23	3.76	52.16	3.99	49.24
Scsd1	0.41	0.02	0.41	0.00	0.43	−4.65
Israel[a]	17.36	0.29	14.65	15.61	14.94	13.94
E226	1.55	0.19	1.43	7.74	1.62	−4.32
Scfxm1	1.97	0.13	1.76	10.66	1.89	4.06
Scrs8	2.28	0.09	2.61	−14.47	2.70	−15.56
Ship04s	1.27	0.04	1.32	−3.94	1.36	−6.62
Scsd6	0.84	0.03	0.83	1.20	0.86	−2.33
Forplan	1.90	0.18	1.18	37.89	1.36	28.42
Agg2	11.49	0.49	8.78	23.59	9.27	19.32
Agg3	12.48	0.49	9.05	27.48	9.54	23.56
Ship08s	1.54	0.05	1.55	−0.65	1.60	−3.75
Pilot4	14.24	0.13	14.24	0.00	14.37	−0.90
Scfxm2	4.24	0.25	3.86	8.96	4.11	3.07
Ship04l;	2.27	0.06	2.27	0.00	2.33	−2.58
Ship12s	1.87	0.05	1.89	1.07	1.94	−3.61
Fpk010	0.83	0.32	0.74	10.84	1.06	−21.70
Sctap2	6.70	0.35	5.16	22.99	5.51	17.76
Scfxm3	6.45	0.38	6.02	6.67	6.40	0.78
Scsd8	1.66	0.07	1.67	−0.60	1.74	−4.60
Czprob	9.55	0.20	9.39	1.68	9.59	−0.42
Ship08l	3.99	0.10	3.88	2.76	3.98	0.25
Sctap3	9.33	0.54	7.39	20.79	7.93	15.01
25fv47	45.42	0.51	40.00	11.93	40.51	10.81
Ship12l	4.89	0.12	4.87	0.41	4.99	−2.00

[a] Solution times for *Israel* are for implementation using direct factorization.

five cases there were small increases (at most 5.07%) in the density of L despite reductions in the density of the corresponding A matrix.

The maximum CPU time for SPARSE was 0.54 seconds on the IBM 3090. There were 17 cases of decrease in total CPU time and 20 cases of increased time. In 10 cases the decrease was greater than 10%; in 3 it was more than 20% and in 1 greater than 40%. The maximum decrease was 49.2%. In 8 cases the increase in CPU time was greater than 5%; in 4 cases it was greater than 10% and in only a single case was it greater than 20%. This case showed an increase of 21.7%. As a whole, for the 38 problems tested, the use of SPARSE decreased the total CPU time by 10.34%. It should be pointed out that SPARSE did particularly well on specific linear programming models such as *Asg*, *Asg2* and *Asg3*, the before mentioned production planning model and on *Sctap1*, *Sctap2* and *Sctap3*, random staircase structure linear programs.[23]

8. Treating Dense Columns in the Coefficient Matrix

In previous sections, we concentrated on the solution of the system of linear equations (2.1) by direct methods. This seems to be appropriate for most real-world linear programming problems, where the AD^2A^T matrix can be made sparse after an appropriate ordering of the rows of A. However, some linear programming formulations contain a few dense columns in the coefficient matrix A, in spite of low overall density. This will result

in AD^2A^T extremely dense, regardless of the selected row permutation. Consequently, we face prohibitively high computational effort and storage requirements in the Gaussian elimination procedure.

We illustrate this situation with Problem *Israel* from the collection of linear programming test problems available through NETLIB.[12] From the nonzero pattern of the linear programming coefficient matrix presented in Figure 24, we detect the presence of a few dense columns. Figure 25 depicts the nonzero pattern for the AA^T matrix, which is already very dense, in spite of the ordering procedure. Of course, the corresponding LU factors, shown in Figure 26, are even more dense since *fill-in* is created.

To remedy this situation, we turn our attention to the conjugate gradient method applied to the solution of systems of linear equations.[22] According to convergence properties described in [16] and other nonlinear programming texts, the conjugate gradient method

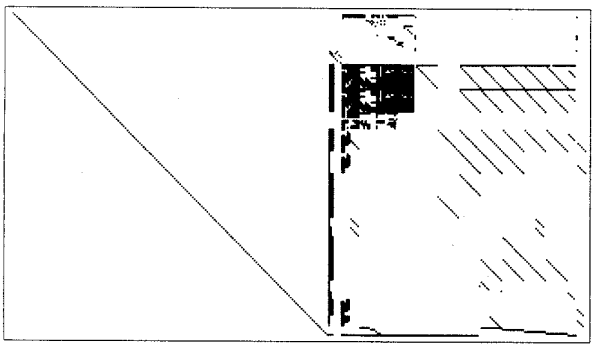

Figure 24. Nonzero pattern of coefficient matrix A—problem *Israel.*

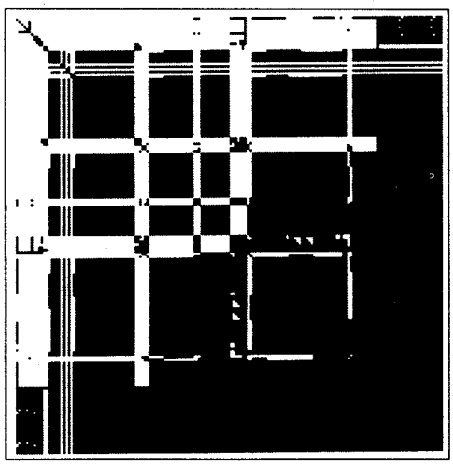

Figure 25. Nonzero pattern of complete AA^T—problem *Israel* (*minimum local fill-in* ordering heuristic).

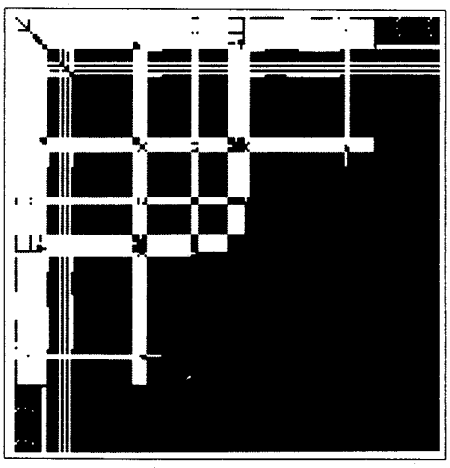

Figure 26. Nonzero pattern of LU factors—problem *Israel* (*minimum local fill-in* ordering heuristic).

solves a system of linear equations in m iterations. In practice, the property of finite termination does not apply, as a consequence of rounding errors. This suggests an analysis of the conjugate gradient method as a genuinely iterative technique.[34] Golub and van Loan[19] presented a convergence analysis of the resulting iterative method, suggesting extremely low convergence for ill-conditioned problems.

One technique to deal with this problem is referred to as *preconditioning.* Following an approach initially discussed in [31], we make use of a hybrid scheme in which we first perform an incomplete factorization of $AD_k^2A^T$. Next, we use the incomplete triangular factors as preconditioners for a conjugate gradient method to solve the system of linear equations defined in (2.1).

The incomplete factorization is obtained by partitioning the coefficient matrix A into two submatrices, based on a threshold value q imposed on the number of nonzero elements in each column. Matrix A_N contains all the columns with less than q nonzero elements, with all the remaining columns in matrix $A_{\bar{N}}$. The selection of q takes into consideration the structure of the coefficient matrix A. It must be such that A_N is of full-rank and $A_NA_N^T$ is sparse. At each iteration k, we compute $\tilde{L}_k$ and $\tilde{L}_k^T$, the Cholesky factors of $A_ND_N^2A_N^T$, where D_N is the principal submatrix of D_k corresponding to the columns that form A_N. As opposed to using the LU factors, we need the symmetry displayed by the Cholesky factors to build the following system of linear equations:

$$Qu = f, \tag{8.1}$$

where

$$Q = \tilde{L}_k^{-1}(AD_k^2A^T)(\tilde{L}_k^T)^{-1}, \tag{8.2}$$

$$u = \tilde{L}_k^T d_y \tag{8.3}$$

and

$$f = \tilde{L}_k^{-1} b. \tag{8.4}$$

The purpose of using preconditioner $\tilde{L}_k$ is to change the eigenvalue structure of $AD_k^2A^T$, hopefully improving the convergence properties of the conjugate gradient method when solving the transformed system (8.1). The selection of this preconditioner is appropriate for linear programming problems where the elimination of a few dense columns in the coefficient matrix results in $A_N D_N^2 A_N^T$ that is much easier to factor via Gaussian elimination than $AD_k^2A^T$. The simple inspection of Figures 27 and 28 will establish this assumption for problem *Israel*. After the removal of a few dense columns from the coefficient matrix, the corresponding AA^T matrix has a favorable nonzero structure. Consequently, after an appropriate ordering of the rows of the coefficient matrix, the Cholesky factors are also sparse.

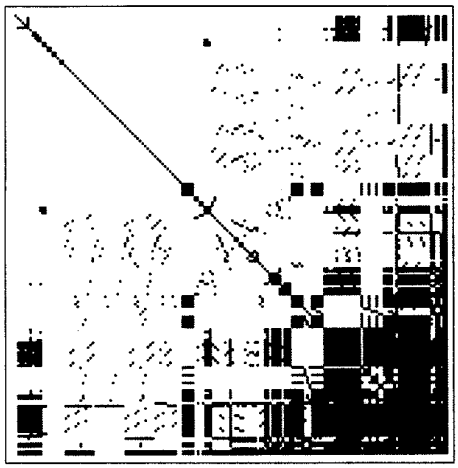

Figure 27. Nonzero pattern of incomplete AA^T—problem *Israel* (*minimum local fill-in* ordering heuristic).

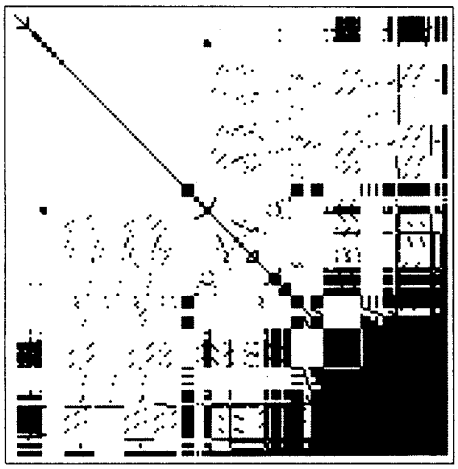

Figure 28. Nonzero pattern of incomplete LU—problem *Israel* (*minimum local fill-in* ordering heuristic).

Various preconditioning strategies have been suggested in the literature.[4,6,26,31,33,37] Their usefulness in the implementation of Karmarkar's algorithm will certainly be the subject of much research in the near future.

Let $\epsilon_{cg} > 0$ be a given termination tolerance. The steps of the conjugate gradient algorithm are outlined is described by Pseudo-code 10.

PSEUDO-CODE 10—CONJUGATE GRADIENT ALGORITHM

procedure CNJGRD(Q, f, ϵ_{cg})
1 $u_0 := f$
2 $r_0 := Qu_0 - f$
3 $p_0 := -r_0$
4 $i := 0$
5 **do** $\| r_i \|_2^2 \geq \epsilon_{cg} \rightarrow$
6 $q_i := Qp_i$
7 $\alpha_i := \| r_i \|_2^2 / p_i^T q_i$
8 $u_{i+1} := u_i + \alpha_i p_i$
9 $r_{i+1} := Qu_{i+1} - f$
10 $\beta_i := \| r_{i+1} \|_2^2 / \| r_i \|_2^2$
11 $p_{i+1} := -r_{i+1} + \beta_i p_i$
12 $i := i + 1$;
13 **od**
end CNJGRD;

9. Summary and Concluding Remarks

In this paper we presented data structures and programming techniques for a direct factorization implementation of the *dual-affine* variant of Karmarkar's algorithm for linear programming. Efficient solution of a sequence of sparse symmetric positive definite systems of linear equations is essential for a successful implementation of most variants of Karmarkar's algorithm. We described an *interpretative* version of Gaussian elimination, tailored to solving such a sequence of systems. This version of Gaussian elimination takes advantage of the strong structural and numerical correlation among the linear systems. In a preprocessing phase of the linear programming algorithm, we perform a *symbolic factorization* step where several data structures are built for use throughout the iterations of the algorithm. These data structures store the problem data and a symbolic representation of the numerical computations that are to be carried out by the algorithm. A *numerical factorization* step accesses this symbolic representation at each iteration. A dense window data structure is used to control the growth of the data structure. Furthermore, we discuss the following techniques required for the efficient implementation of the

interpretative scheme for Gaussian elimination:

- A procedure to build the $AD_k^2A^T$ matrix at each iteration of the linear programming algorithm.
- A scheme in which an approximation of $AD_k^2A^T$ is obtained at each iteration by updating the matrix used in the previous iteration.
- A procedure that identifies and eliminates easily detectable null or fixed variables. As a consequence, the procedure also identifies some trivial infeasibilities or redundancies.
- A scheme designed to increase the sparsity of the input linear programming matrix by carrying out row operations.
- A preconditioned conjugate gradient procedure for handling linear programs with a few dense columns in the coefficient matrix.

Some of the implementation details related to experimental fine tuning are omitted here, but discussed in [1]. Other issues are the subject of future research. These include:

- Obtaining an initial interior feasible solution.
- A variety of termination criteria, including detection of unboundedness and infeasibility.
- Facilities for sensitivity and post-optimality analysis.
- Obtaining an optimal basic solution.
- Computer architectural dependent implementation, including vectorized and concurrent versions.
- Implementation based on a preconditioned conjugate gradient algorithm.
- Data structures and techniques for handling ranges and bounded variables.
- Implementation of primal and primal-dual variants of the algorithm.

ACKNOWLEDGMENTS

This research was partially funded by the Brazilian Council for the Development of Science and Technology—CNPq, and the Management Sciences Staff, US Forest Service—USDA.

REFERENCES

[1] I. ADLER, N. KARMARKAR, M.G.C. RESENDE AND G. VEIGA, 1986. *An Implementation of Karmarkar's Algorithm for Linear Programming*, **Report No. ORC 86-8 (Revised May 1987),** Operations Research Center, University of California, Berkeley, CA.

[2] J. ARONSON, R. BARR, R. HELGASON, J. KENNINGTON, A. LOH AND H. ZAKI, 1985. *The Projective Transformation Algorithm by Karmarkar: A Computational Experiment with Assignment Problems*, **Technical Report 85-OR-3,** Department of Operations Research, Southern Methodist University, Dallas, TX (August).

[3] A. CHANG, 1969. *Application of Sparse Matrix Methods in Electric Power System Analysis*, in R.A. Willoughby (ed.), **Sparse Matrix Proceedings. Report RA1(#11707),** IBM Thomas J. Watson Research Center, Yorktown Heights, NY, pp. 113–121.

[4] P. CONCUS, G.H. GOLUB AND D.P. O'LEARY, 1976. *A Generalized Conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations*, in J.R. Bunch and D.J. Rose (eds.), **Sparse Matrix Computations,** Academic Press, New York, pp. 309–332.

[5] M.L. DECARVALHO, 1987. *On the Work Needed to Factor a Symmetric Positive Definite Matrix*, **Technical Report ORC 87-14**, Operations Research Center, University of California, Berkeley, CA.

[6] J. DOUGLAS, JR. AND T. DUPONT, 1976. *Preconditioned Conjugate Gradient Iteration Applied to Galerkin Methods for a Mildly-Nonlinear Dirichlet Problem*, in J.R. Bunch and D.J. Rose (eds.), **Sparse Matrix Computations,** Academic Press, New York, pp. 333–348.

[7] I.S. DUFF AND J.K. REID, 1974. *A Comparison of Sparsity Orderings for Obtaining a Pivotal Sequence in Gaussian Elimination*, **Journal of the Institute of Mathematics and Its Applications, 14,** 281–291.

[8] I.S. Duff and J.K. Reid, 1982. *MA27—A Set of Fortran Subroutines for Solving Sparse Symmetric Sets of Linear Equations*, **Report AERE R-10533,** Computer Sciences and Systems Division, AERE Harwell, Harwell, England.

[9] I.S. DUFF AND J.K. REID, 1983. *The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations*, **ACM Transactions on Mathematical Software 9,** 302–325.

[10] I.S. DUFF, A.M. ERISMAN AND J.K. REID, 1986. **Direct Methods for Sparse Matrices,** Clarendon Press, Oxford.

[11] S.C. EISENSTAT, M.C. GURSHY, M.H. SCHULTZ AND A.H. SHERMAN, 1982. *The Yale Sparse Matrix Package, I. The Symmetric Codes*, **International Journal for Numerical Methods in Engineering 18,** 1145–1151.

[12] D.M. Gay, 1985. *Electronic Mail Distribution of Linear Programming Test Problems*, **Mathematical Programming Society Committee on Algorithms Newsletter 13,** 10–12.

[13] D.M. GAY, 1988. *Massive Memory Buys Little Speed for Complete, In-Core Sparse Cholesky Factorizations*, **Numerical Analysis Manuscript 88-04**, AT&T Bell Laboratories, Murray Hill, NJ.

[14] J.A. GEORGE AND J.W.H. LIU, 1981. *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, NJ.

[15] J.A. GEORGE AND J.W.H. LIU, 1987. *The Evolution of the Minimum Degree Ordering Algorithm*, **Technical Report CS-87-06,** Department of Computer Science, York University, North York, Canada.

[16] P.E. GILL, W. MURRAY AND M.H. WRIGHT, 1981. **Practical Optimization,** Academic Press, London.

[17] P.E. GILL, W. MURRAY, M.A. SAUNDERS, J.A. TOMLIN AND M.H. WRIGHT, 1986. *On Projected Newton Barrier Methods for Linear Programming and an Equivalence to Karmarkar's Projective Method*, **Mathematical Programming 36,** 183–209.

[18] G.H. GOLUB AND C.F. VAN LOAN, 1979. *Unsymmetric Positive Definite Linear Systems*, **Linear Algebra and Its Applications 28,** 85–98.

[19] G.H. GOLUB AND C.F. VAN LOAN, 1983. **Matrix Computations,** The Johns Hopkins University Press, Baltimore, MD.

[20] C. GONZAGA, 1987. *Search Directions for Interior Linear Programming Methods*, **Memorandum No. UCB/ERL M87/44,** Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA.

[21] F.G. GUSTAVSON, W.M. LINIGER AND R.A. WILLOUGHBY, 1970. *Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations*, **Journal of ACM 17,** 87-109.

[22] M.R. HESTENES AND E. STIEFEL, 1952. *Methods of Conjugate Gradients for Solving Linear Systems*, **Journal of Research of the National Bureau of Standards Section B 49,** 409–436.

[23] J.K. HO AND E. LOUTE, 1981. *A Set of Staircase Linear Programming Test Problems*, **Mathematical Programming 20,** 245–250.

[24] A.J. HOFFMAN AND S.T. MCCORMICK, 1982. *A Fast Algorithm That Makes Matrices Optimally Sparse*, **Technical Report SOL 82-13,** Systems Optimization Laboratory, Stanford University, Stanford, CA.

[25] N. KARMARKAR, 1984. *A New Polynomial-Time Algorithm for Linear Programming*, **Combinatorica 4,** 373–395.

[26] D.S. KERSHAW, 1978. *The Incomplete Cholesky-Conjugate Method for the Iterative Solution of Systems of Linear Equations*, **Journal of Computational Physics Comput 26,** 43–65.

[27] R.C. LEACHMAN, 1986. *Preliminary Design and Development of a Corporate-Level Production Planning System for the Semiconductor Industry*, **ORC Report 86-11,** Operations Research Center, University of California, Berkeley, CA.

[28] I.J. LUSTIG, 1985. *A Practical Approach to Karmarkar's Algorithm*, **Technical Report SOL 85-5,** Systems Optimization Laboratory, Stanford University, Stanford, CA.

[29] H.M. MARKOWITZ, 1957. *The Elimination Form of the Inverse and Its Application to Linear Programming*, **Management Science 3,** 255–269.

[30] R.E. MARSTEN, M.J. SALTZMAN, D.F. SHANNO, G.S. PIERCE AND J.F. BALLINTIJN, 1988. *Implementation of a Dual Affine Interior Point Algorithm for Linear Programming*, **CMI-WPS-88-06,** Center for Management of Information, University of Arizona, Tucson, AZ 85721.

[31] J.A. MEIJERINK AND H.A. VAN DER VORST, 1977. *An Iterative Solution Method for Linear Equation Systems of Which the Coefficient Matrix Is a Symmetric M-Matrix*, **Mathematics of Computation 31,** 148–162.

[32] C.L. MONMA AND A.J. MORTON, 1987. *Computational Experience with a Dual Affine Variant of Karmarkar's Method for Linear Programming*, **Manuscript,** Bell Communications Research.

[33] N. MUNKSGAARD, 1980. *Solving Sparse Symmetric Sets of Linear Equations by Preconditioned Conjugate Gradients*, **ACM Transactions on Mathematical Software 6,** 206–219.

[34] J.K. REID, 1971. *On the Method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations*, in J.K. Reid (ed.), **Large Sparse Sets of Linear Equations,** Academic Press, London, pp. 231–254.

[35] D.J. ROSE, 1970. *Symmetric Elimination on Sparse Positive Definite Systems and Potential Flow Network Problem*, **Ph.D. Thesis,** Harvard University, Cambridge, MA.

[36] D.J. ROSE, 1972. *A Graph-Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations*, in R.C. Reid (ed.), **Graph Theory and Computing,** Academic Press, New York, pp. 183–217.

[37] Y. SAAD, 1985. *Practical Use of Polynomial Preconditionings for the Conjugate Gradient Method*, **SIAM Journal of Scientific C and Statistical Computing 6,** 865–881.

[38] R.E. TARJAN, 1983. *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia.

[39] W.F. TINNEY AND J.W. WALKER, 1967. *Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization*, **Proceedings IEEE 55,** 1801–1809.

[40] M.J. TODD AND B.P. BURRELL, 1986. *An Extension of Karmarkar's Algorithm for Linear Programming Using Dual Variables*, **Algorithmica 1,** 409–424.

[41] J.A TOMLIN, 1985. *An Experimental Approach to Karmarkar's Projective Method for Linear Programming*, **Manuscript,** Ketron, Inc., Mountain View, CA.

[42] J.A. TOMLIN, 1987. *A Note on Comparing Simplex and Interior Methods for Linear Programming*, **Preliminary draft,** Ketron, Inc., Mountain View, CA.

[43] H. YAMASHITA, 1986. *A Polynomially and Quadratically Convergent Method for Linear Programming*, **Manuscript,** Mathematical Systems Institute, Inc., Tokyo, Japan.

[44] M. YANNAKAKIS, 1981. *Computing the Minimum Fill-in Is NP-Complete*, **SIAM Journal on Algebraic and Discrete Methods 2,** 77–79.