Block Triangular Orderings and Factors for Sparse Matrices in LP [1]

Roger Fletcher

Department of Mathematics and Computer Science, University of Dundee, Dundee DD1 4HN, Scotland, UK.

Numerical Analysis Report NA/177, August 1997

Abstract

Sparse matrix methods for factorizing a nonsingular matrix are considered. The possibility of using the little known technique of implicit LU factors is explored, particularly in the context of simplex-like methods for Linear Programming. The concept of a spike-preserving ordering is introduced and a new method for calculating such an ordering is described, based on the recursive use of Tarjan's algorithm for block triangularization. Experiments are described in which the new method is compared to that based on the use of Markowitz orderings.

1 Introduction

The research in this paper arises in the context of sparse matrix methods for factorizing a nonsingular $n \times n$ matrix A. Such methods are relevant to many fields of study, but here there is a particular interest in developing methods for use with simplex-like methods for Linear Programming (LP) and related calculations, in which column updates of the matrix A are carried out on each iteration.

The main aim is to explore the possibilities inherent in a little known but promising factorization technique known as *implicit LU factors*. This can be used whenever a copy of A is available, and is attractive since it takes direct advantage of the sparsity of A, and also enables the upper triangular factor U to be dispensed with. Implicit LU factors are also much more readily updated than regular LU factors. A certain unit lower triangular matrix $\mathbb{L}$ is stored in the method, and much of the paper considers the means by which the sparsity of $\mathbb{L}$ might be optimized. The idea of implicit LU factors is explained in Section 2, and its relationship to the regular LU factors that may be computed by

[1]This paper was presented at the Dundee Biennial Conference on Numerical Analysis, June 1997, and at the 16th International Symposium on Mathematical Programming, Lausanne, August 1997.

Gaussian Elimination (GE) is described. It is also shown how implicit LU factors may be used to solve well-determined systems of linear equations.

An important technique for taking account of sparsity in a nonsingular matrix A is the determination of its irreducible block triangular ordering. Section 3 reviews the steps in this calculation, which is cheap to carry out, and permits the use of a very simple data structure. Block triangular orderings are a useful first step towards exploiting sparsity in the $\mathbb{L}$ factor. However it is often the case that only a limited amount is gained by this ordering, and the most challenging question is that of how sparsity within the diagonal blocks might be exploited. It is argued in Section 4 that a row skyline data structure for $\mathbb{L}$ is advantageous, and ordering methods for concentrating the non-zeros into isolated spikes are considered. A new concept of spike-preserving orderings is introduced which gives a sufficient condition for the spike pattern of $\mathbb{L}$ to be the same as that of A.

A new algorithm for spike generation is described in Section 5, based on the spike-preserving property, followed by an outline of the SPK1 algorithm, which also generates orderings with this property. The relative merits and demerits of these two algorithms are considered. Some illustrations of the successful use of the algorithms are shown. However it is pointed out that both algorithms permit of further improvement.

Finally some experiments are described to monitor the total spike length on a range of LP problems, and the results are compared with the fill-in in regular LU factors that occurs when using Markowitz pivoting. The issue of updating the factors is then addressed and the likelihood of obtaining a satisfactory sparse update scheme for LP is assessed.

2 Implicit LU factors

In this section the idea of implicit LU factors of an $n\times n$ nonsingular matrix A is explained and its relationship to the regular LU factors that may be computed by GE is described. It is also shown how implicit LU factors may be used to solve well-determined systems of linear equations. The issues that arise in assessing the suitability of implicit LU factors for sparse matrix calculations are discussed.

First the computation and use of regular LU factors by GE is reviewed, using a compact matrix notation. The details of the computation are well-known (e.g Wilkinson [10]) and can be expressed in the form

$$A^{(k+1)} = M^{(k)}A^{(k)} \qquad k = 1,2,\dots,n-1 \tag{2.1}$$

where $A^{(1)} = A$. $M^{(k)}$ differs from a unit matrix only in that $(M^{(k)})_{jk} = -m_{jk}$ for $j = k+1,\dots,n$, where

$$m_{jk} = a^{(k)}_{jk}/a^{(k)}_{kk} \qquad j = k+1,\dots,n \tag{2.2}$$

are the so-called *multipliers* on stage k. The operation in (2.1) eliminates the sub-diagonal elements in column k of $A^{(k)}$ so that the generic form of $M^{(k)}$ and $A^{(k)}$ is typified for

$n = 6$ and $k = 3$ by

$$M^{(k)} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & -m_{43} & 1 & & \\ & & -m_{53} & & 1 & \\ & & -m_{63} & & & 1 \end{bmatrix} \quad \text{and} \quad A^{(k)} = \begin{bmatrix} a_{11}^{(k)} & a_{12}^{(k)} & a_{13}^{(k)} & a_{14}^{(k)} & a_{15}^{(k)} & a_{16}^{(k)} \\ & a_{22}^{(k)} & a_{23}^{(k)} & a_{24}^{(k)} & a_{25}^{(k)} & a_{26}^{(k)} \\ & & a_{33}^{(k)} & a_{34}^{(k)} & a_{35}^{(k)} & a_{36}^{(k)} \\ & & a_{43}^{(k)} & a_{44}^{(k)} & a_{45}^{(k)} & a_{46}^{(k)} \\ & & a_{53}^{(k)} & a_{54}^{(k)} & a_{55}^{(k)} & a_{56}^{(k)} \\ & & a_{63}^{(k)} & a_{64}^{(k)} & a_{65}^{(k)} & a_{66}^{(k)} \end{bmatrix}.$$

It is noted that the inverse of $M^{(k)}$ is obtained simply by changing the signs of the subdiagonal elements of $M^{(k)}$. Then it is readily verified that $A = LU$ where

$$L = M^{(1)^{-1}} M^{(2)^{-1}} \dots M^{(n-1)^{-1}} \quad \text{and} \quad U = A^{(n)}. \tag{2.3}$$

Equivalently L is obtained simply by adding the multipliers m_{ij} into the corresponding sub-diagonal elements of a unit matrix, for all $i > j$. A row interchange may be carried out at the start of each stage (partial pivoting) to ensure that the pivot $a_{kk}^{(k)}$ is non-zero, and commonly the largest modulus pivot is chosen. This ensures that the elements of L are bounded in modulus by 1. Other pivoting strategies are also possible in which both row and column interchanges are allowed, and these can be particularly useful in a sparse matrix context. However to keep the notation simple, it is assumed that any interchanges are subsumed into the original matrix A.

The value of LU factors is that they readily enable a nonsingular linear system to be solved. To solve a system $A\mathbf{x} = \mathbf{b}$, the systems

$$L\mathbf{y} = \mathbf{b} \quad \text{and} \quad U\mathbf{x} = \mathbf{y} \tag{2.4}$$

are solved in turn. These are triangular systems and may be solved by forward and backward substitution respectively. Likewise to solve $A^T\mathbf{x} = \mathbf{b}$, the systems

$$U^T\mathbf{y} = \mathbf{b} \quad \text{and} \quad L^T\mathbf{x} = \mathbf{y} \tag{2.5}$$

are solved in a similar way.

The computation of regular LU factors is well understood, and has come to be regarded as the method of choice for the direct solution of a nonsingular linear system. However there is an alternative approach referred to as *implicit LU factors*, which is equally effective for dense systems, and about which there are many relevant papers, yet which is not at all well known. This method is reviewed by Fletcher [3], where many other references are given. It is shown in [3] that implicit LU factors enable systems with both A and A^T to be solved, and it may be that this feature has not been fully recognised in previous work, most of which concentrates on the suitability of the factors for use in an iterative method.

In the method of implicit LU factors, a unit lower triangular matrix $\mathbb{L}$ and a diagonal matrix $D = \text{diag}(d_1, d_2, \dots, d_n)$ are computed. $\mathbb{L}$ is the *inverse* of the matrix L in the

regular LU factors, and D contains the diagonal elements (pivots) of U. The rest of the U matrix is *not* stored, but is implicitly determined by the equation

$$U = \mathbb{L}A, \tag{2.6}$$

through $\mathbb{L}$ and the original copy of A. There is also an analogous form of the method in which L is implicit and the inverse of U is kept. However the form in which U is implicit is more suitable for our purposes.

The computation of $\mathbb{L}$ is via the product

$$\mathbb{L} = M^{(n-1)} \dots M^{(2)} M^{(1)} = L^{-1}, \tag{2.7}$$

using the notation introduced above. On an intermediate stage k of the calculation we have

$$\mathbb{L}^{(k)} = M^{(k-1)} \dots M^{(2)} M^{(1)} = M^{(k-1)} \mathbb{L}^{(k-1)}. \tag{2.8}$$

and this recurrence is used to calculate $\mathbb{L}^{(k)}$ from $\mathbb{L}^{(k-1)}$. Initially $\mathbb{L}^{(1)} = I$ and $A^{(1)} = A$. It follows from (2.1) that the matrix $A^{(k)}$ in Gaussian elimination is *implicitly* defined by

$$A^{(k)} = \mathbb{L}^{(k)} A, \tag{2.9}$$

so we can generate column k of $A^{(k)}$ by computing $\mathbb{L}^{(k)}\mathbf{a}_k$ where $\mathbf{a}_k$ denotes column k of A. This enables the multipliers for stage k of GE to be calculated as in (2.2). This determines $M^{(k)}$ and hence $\mathbb{L}^{(k+1)} = M^{(k)}\mathbb{L}^{(k)}$, and enables the calculation to move forward to the next stage. It is clear that the information to do partial pivoting is readily available, and it is convenient to store the pivot element d_k for use in the solution algorithms.

It is not immediately obvious how to use implicit LU factors to solve systems, as the matrix U in (2.4) and (2.5) is no longer available. However it is shown in [3] how the implicit definition of U may be exploited to determine algorithms for solving systems. To solve $A\mathbf{x} = \mathbf{b}$ we may use the algorithm

$$\begin{array}{l} \mathbf{b}^{(n)} = \mathbf{b} \\ \texttt{for } \ i = n, n-1, \dots, 1 \\ \quad x_i = \mathbf{l}_i^T \mathbf{b}^{(i)} / d_i \\ \quad \mathbf{b}^{(i-1)} = \mathbf{b}^{(i)} - \mathbf{a}_i x_i \\ \texttt{end}, \end{array} \tag{2.10}$$

where $\mathbf{l}_i^T$ denotes row i of $\mathbb{L}$. To solve $A^T\mathbf{x} = \mathbf{b}$ we may use

$$\begin{array}{l} \mathbf{x}^{(0)} = \mathbf{0} \\ \texttt{for } \ i = 1, 2, \dots, n \\ \quad y_i = (b_i - \mathbf{a}_i^T \mathbf{x}^{(i-1)}) / d_i \\ \quad \mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \mathbf{l}_i y_i \\ \texttt{end} \\ \mathbf{x} = \mathbf{x}^{(n)}. \end{array} \tag{2.11}$$

The major cost of (2.10) is a saxpy operation with $\mathbf{a}_i$ and a dot operation with $\mathbf{l}_i$ at each stage, whereas for (2.11) it is a saxpy operation with $\mathbf{l}_i$ and a dot operation with $\mathbf{a}_i$. This indicates that a preferred storage scheme for implicit LU factors is one in which $\mathbb{L}$ is stored explicitly by rows, and A by columns. It is readily observed that these algorithms are totally different in structure to (2.4) and (2.5), and this indicates that the implicit LU approach is more than just a minor variation on the regular LU algorithm.

For dense A, it is possible for $\mathbb{L}$ and D to overwrite the lower triangle of A as the calculation proceeds, in which case the implicit LU algorithm requires $\frac{1}{3}n^3 + O(n^2)$ multiplications, as for GE. Also it will be seen below that only the strict upper triangle of A is required when solving systems with implicit LU factors. Likewise, only the strict upper triangle of A is required to recover U from (2.6). Thus the lower triangle of A can be replaced by $\mathbb{L}$, D and its strict upper triangle, without any essential loss of information (ignoring round-off). However for sparse A this property is unlikely to be advantageous. Also, in the dense case, the solution algorithms (2.10) and (2.11) each require $n^2 + O(n)$ multiplications if columns of the upper triangle of A are used. This is the same as would be required by the combined forward and backward substitutions of (2.4) and (2.5). A backward error analysis is also available [3] in the dense case which indicates that the numerical stability properties of the implicit LU approach is similar to that of the regular LU approach, if partial pivoting is used in each case.

Another feature of the implicit LU scheme is that it readily permits of *updating* the factors when one column of A is replaced by another, as in the simplex method of linear programming. The idea (again see [3]) is to modify the Fletcher–Matthews method to update implicit LU factors rather than regular LU factors. In fact, in the dense case, it is more efficient to update implicit factors on account of the fact that U need not be updated. I have been using this updating scheme (the L–method in [3]) as the basis of a dense code for linear and quadratic programming, and the scheme has exhibited excellent numerical stability and reliability.

A particular advantage of the implicit LU approach is that there are potential savings when A is a sparse matrix, both in the calculation of the factors and in their use in the solution algorithms. However a potential difficulty is that the matrix $\mathbb{L}$ is the *inverse* of the matrix L in the regular case. Thus, although is is quite likely that L may be sparse in this context, this is not necessarily true of $\mathbb{L}$. The main thrust of this paper is to seek for orderings of A in which as much sparsity as possible is retained in the matrix $\mathbb{L}$.

In the context of sparse matrices, it is the case that column pivoting as well as row pivoting becomes important. In the case of regular LU factors, the Markowitz scheme (see Duff, Erisman and Reid [1]) has proved to be very effective in controlling fill–in in the LU factors. It is worth pointing out that for implicit factors, column pivoting is also a practical possibility. First of all there is the possibility of choosing a column ordering which is attractive from the point of view of sparsity control. Moreover, to examine the value of an element in the active part of the matrix $A^{(k)}$ as a potential pivot requires only a single scalar product to be calculated. If the sparsity pattern of $A^{(k)}$ is known, then savings can be made by examining only those elements that are known to be non-zero,

and are likely to be favourable in regard to fill-in. Thus is is quite feasible in practice to examine elements from a number of columns if the matrices involved are sparse.

3 Block Triangular Orderings

An important technique for taking account of sparsity in a nonsingular matrix A is the determination of its irreducible block triangular ordering. This section reviews the steps in this calculation, which is cheap to carry out, and permits the use of a very simple data structure.

There exists some inherent sparsity in the triangular factor $\mathbb{L}$ if the matrix A can be transformed to a block upper triangular matrix by row and column permutations. For example, to take an extreme case, if A is an upper triangular matrix then $\mathbb{L} = I$ has optimum sparsity. If A is block upper triangular, then $\mathbb{L}$ has zero elements in locations that lie below the block upper triangular profile. Thus it is important to discover the extent to which A can be transformed to block upper triangular form. Fortunately there are efficient algorithms that enable this to be done, and these are reviewed in this section. In the subsequent section we examine the extent to which sparsity in $\mathbb{L}$ can be induced by reordering within the block triangular profile.

There are two stages in finding an irreducible block triangular ordering of a square matrix. The first of the consists in finding a *transversal*, which is an ordering in which there are non-zero elements on the diagonal of the matrix. (For the purposes of this discussion, elements are regarded as being either zero or non-zero, and the actual values of the non-zero elements are disregarded.) It is possible to find a transversal (if one exists) by permuting either rows or columns, and we shall assume that a row permutation PA is sought, where P is a permutation matrix. Many algorithms have been suggested for this task and we follow the recommendations of [1] based on the work of Duff and Gustavson.

The idea is to take the columns in turn, and as far as possible to assign non-zero row elements to lie on the diagonal in an arbitrary manner. Usually there comes a stage at which all the non-zero elements in the new column have already been assigned. In this case a reassignment of the current partial transversal is sought. A non-zero element of the new column is considered as a transversal element, in which case a transversal element in a previous column must be reassigned. If there is an unassigned element in this column then this can be used to extend the transversal. If not, a deeper search for a reassignment must be made. In general a depth-first search is made of all the possibilities for a reassignment by which to extend the transversal. If no reassignment can be found, then the matrix is said to be *structurally singular*. A consequence is that the new column is linearly dependent on the previous columns for all possible numerical values of the non-zero elements. Otherwise the transversal is extended by one row, and the algorithm proceeds to the next new column.

The second stage in the block triangularization process is to make a symmetric permutation $Q^T PAQ$ of the matrix PA arising from the first stage. (Q is another permutation matrix.) The method of choice is due to Tarjan [9] and is also explained in detail in [1].

A brief explanation will suffice here. The outcome is illustrated by the pair of matrices in Figure 1. The left hand matrix has a transversal but is not irreducible. The right hand matrix shows the optimum reordering given by Tarjan's algorithm. This ordering is essentially unique except for permutations within each block and (in some cases) reordering the blocks along the diagonal.

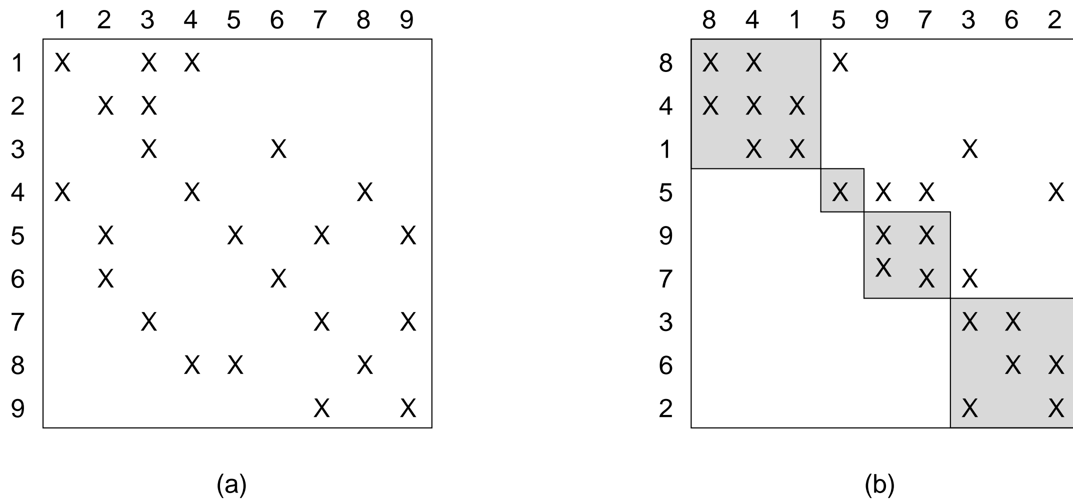

Figure 1: Transversal and irreducible block upper triangular orderings

Tarjan's algorithm builds up information about the ordering by considering the off-diagonal elements one at a time. For example it follows from the (7, 9) element in Figure 1(a) that 7 must either preceed 9 in the final ordering, or must lie in the same block. We write this as $7 \preceq 9$. However from the (9, 7) element in (a) it follows that $9 \preceq 7$. Thus we deduce that 7 and 9 must lie in the same diagonal block. The elements are examined in a depth-first search and a partial ordering is built up based on the elements that have so far been considered. Elements that must lie in the same block are designated by storing along with each element in the ordering, a number which indicates the highest ordered element that must lie in the same block as the current element. This is a key idea which enables the algorithm to be implemented very efficiently.

During the search it may become clear that one group of elements must precede all others in the ordering. For example, when all the elements in columns 1, 4 and 8 have been examined we have the relationships $4 \preceq 1$, $1 \preceq 4$, $8 \preceq 4$ and $4 \preceq 8$ and no others. This tells us that 1, 4 and 8 must occur in the same block, and that there are no other columns that must precede these columns in the final ordering. Thus we are at liberty to move this group, known as a strongly connected component, or *strong component* to the front of the ordering. These rows and columns are then ignored for the rest of the search process. As the algorithm proceeds, further strong components are separated out. These strong components correspond to the successive diagonal blocks in the final block upper triangular matrix.

Tarjan's algorithm requires each non-zero element in the matrix to be accessed only once, and this makes the algorithm very efficient. Also a very simple compact sparse matrix data structure is suitable for implementation. In this there is a real vector which stores all the values of the non-zero elements, grouped by column, and an integer vector of the same length which stores the corresponding row indices. Also a shorter (usually) integer vector holds pointers to the start of each column group in the other vectors. In some implementations the lengths of each column group is also kept, which allows gaps between the column groups to be present. In the example of Figure 1(a), the vector of row indices is

$$[1 \quad 4 \quad 2 \quad 5 \quad 6 \quad 1 \quad 2 \quad 3 \quad 7 \quad 1 \quad 4 \quad 8 \quad 5 \quad 8 \quad 3 \quad 6 \quad 5 \quad 7 \quad 9 \quad 4 \quad 8 \quad 5 \quad 7 \quad 9]$$

and the vector of pointers is

$$[1 \quad 3 \quad 6 \quad 10 \quad 13 \quad 15 \quad 17 \quad 20 \quad 22 \quad 25].$$

Tarjan's algorithm keeps an extra vector of pointers to indicate which elements in each column grouping have currently been examined. There is no need to examine the elements within each group in any particular order. The same is true for the implementation of the transversal search algorithm.

In a Linear Programming context, the matrix A will usually contain a substantial numbers of unit columns, that is columns having a single non-zero element whose value is unity. These arise due to slack or artificial variables in the simplex method basis, or to active simple bounds in an active set method. It is more efficient to treat these columns in a special way, particularly for the solution algorithms (see the discussion in Section 4 of [3]). Thus the generic structure of A is deemed to be

$$A = \begin{bmatrix} I & A_1 \\ 0 & A_2 \end{bmatrix} \tag{3.1}$$

in which I is the $m_1 \times m_1$ unit matrix, A_2 is an $m_2 \times m_2$ matrix, and there are m_1 unit columns. The subdiagonal elements of $\mathbb{L}$ are known to be zero in columns $1, \ldots, m_1$. The sub-block A_1 is part of U and does not need extra storage if implicit LU factors are used. A consequence of this data structure is that the transversal search and Tarjan's algorithm need only be applied to the sub-block A_2.

4 Skyline and Spike Orderings

In this section we show how block triangular orderings are a useful first step towards exploiting sparsity in the $\mathbb{L}$ factor. The potential for exploiting sparsity within the diagonal blocks is then considered. It is argued that a row skyline data structure for $\mathbb{L}$ is advantageous, and ordering methods for concentrating the non-zeros of A into isolated spikes are considered. The concept of a *spike-preserving* matrix is introduced to ensure that there is no fill-in in $\mathbb{L}$ outwith the spikes of A.

The outcome of the Tarjan algorithm is an block upper triangular ordering

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1p} \\ & A_{22} & \dots & A_{2p} \\ & & \ddots & \vdots \\ & & & A_{pp} \end{bmatrix} \qquad (4.1)$$

in which the permutations have been subsumed into A for convenience. We first review how this ordering might be used to solve systems based on using regular LU factors. The ordering can be used to solve a system $A\mathbf{x} = \mathbf{b}$ by a block back substitution process

$$\mathbf{x}_i = A_{ii}^{-1}(\mathbf{b}_i - \sum_{j=i+1}^{p} A_{ij}\mathbf{x}_j) \qquad i = p, \dots, 2, 1. \qquad (4.2)$$

Likewise $A^T\mathbf{x} = \mathbf{b}$ can be solved by a block forward substitution process

$$\mathbf{x}_i = A_{ii}^{-T}(\mathbf{b}_i - \sum_{j=1}^{i-1} A_{ji}^T\mathbf{x}_j) \qquad i = 1, 2, \dots, p. \qquad (4.3)$$

These calculations only need LU factors of the diagonal blocks in (4.1) and so can be very efficient if the sizes of the diagonal blocks are small. More often however there are one or more quite large irreducible blocks, which nevertheless contain quite a lot of sparsity. Thus it can be important to get *sparse LU factors* within each irreducible block. This requires a quite complex data structure. Also it is quite common that A contains one huge (but sparse) irreducible block (see for example the `israel` example below), so that relatively little is gained from (4.1). Thus many codes use Markowitz pivoting on the full matrix A (perhaps taking advantage of unit columns as in (3.1)) and ignore the possibility of making use of the block triangular ordering.

We now consider how the block upper triangular ordering (4.1) might be used advantageously with implicit LU factors. It is clear that the matrix $\mathbb{L}$ (for which $\mathbb{L}A = U$) is just a block diagonal matrix of submatrices $\mathbb{L}_{ii}$ for which $\mathbb{L}_{ii}A_{ii} = U_{ii}$. Thus if the dimension of the diagonal blocks is small, then this ordering induces considerable sparsity in the matrix $\mathbb{L}$. In the implicit LU method it is only necessary to store the matrices $\mathbb{L}_{ii}$, together with the pivots (diagonal elements of U). As we have seen, the solution algorithms (2.10) and (2.11) make dot and saxpy operations with *columns of* A and *rows of* $\mathbb{L}$. The simple data structure described at the end of the previous section is very suitable for making operations with columns of A. As regards $\mathbb{L}$, a suitable data structure would be a *row skyline* ordering in which the elements of $\mathbb{L}$ are stored by contiguous elements in rows, as illustrated in Figure 2. The shaded part of the diagram corresponds to the block upper triangle of A and the *row spikes* below the diagonal correspond to the non-zero parts of the rows of $\mathbb{L}$.

Methods exist in which the irreducible diagonal blocks of A may each be ordered so as to minimize (or at least diminish) the length of the row spikes. A question of some

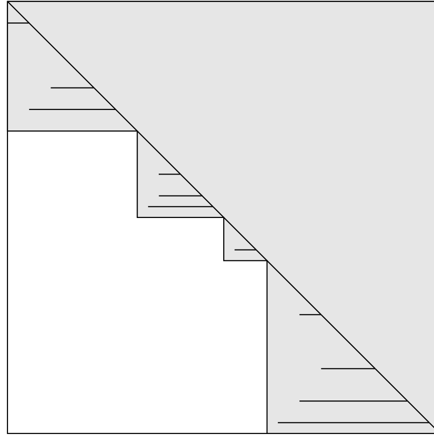

Figure 2: Skyline (row spike) ordering for a block upper triangular matrix.

interest is the extent to which $\mathbb{L}$ fills in for these orderings. To assess the potential for minimizing the number of spikes, some special cases of irreducible matrices are illustrated in Figure 3. The positions in $\mathbb{L}$ in which fill-in occurs are denoted by the shading in the figures.

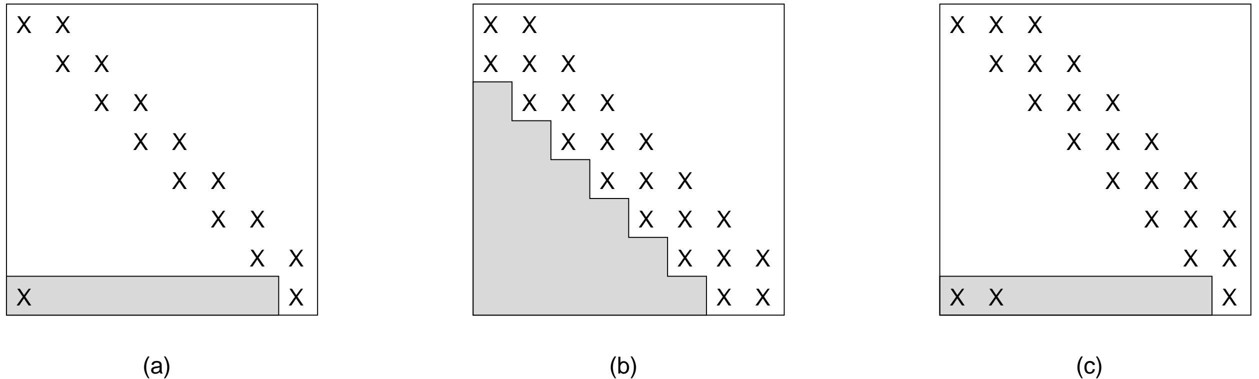

Figure 3: Fill-in for implicit LU factors

It can be seen in Figure 3(a) that the matrix A has only one spike, and that L and $\mathbb{L}$ fill in only within this spike. However the tridiagonal matrix in Figure 3(b) has $n-1$ spikes of length 1, and L does not fill in at all, but there is total fill-in below the diagonal for $\mathbb{L}$. Fortunately this can be avoided by reordering the matrix (numerical stability permitting), and a cyclic row permutation as in Figure 3(c) gives much less fill-in in $\mathbb{L}$.

A sufficient condition for an ordering of A not to cause fill-in in $\mathbb{L}$ outwith the spikes of A is provided by the following recursive definition. In the context of implicit LU factors, we shall say that a matrix A is *spike-preserving* if it has nonsingular LU factors, and either

(i) A is fully dense (or we choose to regard it as such), or

(ii)

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{4.4}$$

in which A_{11} is a block upper triangular matrix whose diagonal blocks are spike-preserving, and in which the first element of each row of A_{21} is non-zero.

Clearly a dense matrix is spike-preserving, and likewise a block upper triangular matrix is spike-preserving if its diagonal blocks are spike preserving. Since the sub-diagonal rows in (4.4) corresponding to $[\,A_{21} \quad A_{22}\,]$ are full length spikes, it follows that (4.4) is spike-preserving. Illustrations of spike-preserving matrices A (assuming the existence of nonsingular LU factors) are given in Section 5.

Because $\mathbb{L}$ is an inverse matrix, it is likely that there will be fill-in in the interior of each spike row, but we shall see that it is possible to keep down the number of the spikes and the length of the spikes by a careful use of ordering techniques within the irreducible blocks. This is the subject of the next section of this paper. Because the spike rows are likely to be dense in their interior, the elements can be stored in consecutive locations and only the first location and the length of each spike need be stored. This gives a very simple data structure. Also no special attention need be paid to the irreducible block structure, as would be required by (4.2) and (4.3). The location of the spikes is all that is needed to implement the solution algorithms (2.10) and (2.11).

Various algorithms have been suggested for spike generation, notably the P^3 and P^4 algorithms of Hellerman and Rarick [5] and the P^5 algorithm of Erisman, Grimes, Lewis and Poole [2]. A more simple algorithm is the SPK1 algorithm of Stadtherr and Wood [7] (see Fletcher and Hall [4]), and practical experience [4] indicates that on average this is equally effective in controlling fill-in, whilst being easier to code and cheaper to run. In this paper we suggest another spike generation algorithm based on tearing and the recursive use of Tarjan's algorithm. Such an approach has been used before by Lin and Mah [6], but their approach is rather cumbersome and no practical experience on large problems is presented.

A complication with spike generation within an irreducible block is that the issue of numerical stability cannot be ignored. Thus it is possible that a near-zero pivot determined by the ordering algorithm may cause significant growth in the factors and hence cause large numerical errors due to round-off. In this case it is necessary to reclassify the troublesome row as a spike in some way and treat it at a later stage in the calculation. This not only increases the total length of the spike rows, but also requires that the numerical pivoting is carried out along with the ordering, rather than afterwards. Threshold pivoting can reduce the disruption caused by numerical pivoting but does not eliminate the need to take account of it.

In Section 6 practical experience is presented on a range of smallish to medium sized problems derived from LP calculations. This compares the new recursive ordering and the

SPK1 ordering using implicit LU factors, with the commonly used Markowitz algorithm for regular LU factors. It is assumed that the effects of numerical instability can be ignored in all three methods. Although this is unrealistic in practice, for the reason stated in the previous paragraph, it is felt that the results do give at least an approximate measure of the relative merit of the algorithms in question.

5 A Recursive Tarjan Algorithm

In this section a new algorithm for spike generation is described, followed by an outline of the SPK1 algorithm. The relative merits and demerits of the two algorithms are considered. Some illustrations of the successful use of the algorithms are shown. However it is pointed out that both algorithms permit of further improvement.

A spike generation algorithm is suggested by the recursive definition of a spike-preserving matrix given in (4.4). Assume that we are presented with an irreducible structurally nonsingular matrix having some sparsity. Note that every row and column must have at least two non-zero elements. We proceed as follows.

1. Pick a *tear-column* and permute it to column 1.

2. Permute one of its non-zero elements to the top and the others (k say) to the bottom.

3. Permute k columns to the right hand side.

4. Find the irreducible block structure of the resulting interior block and apply the entire process recursively to each diagonal block that is obtained.

The recursion returns to the previous level whenever the current block is fully dense. The algorithm is illustrated in Figure 4.

The decision as to which columns to choose at steps 1 and 3 has an important bearing on the outcome. The k rows in Figure 4 will appear as spikes in the final ordered matrix. To minimize the number of spikes that are generated we therefore choose a tear-column with *minimum column count* (that is as few non-zero elements as possible). In step 2, the decision as to which row to move to the top can be taken entirely on the basis of pivot size, as sparsity within the spikes is unimportant (since the spikes fill in when the $\mathbb{L}$ factor is calculated).

How tie-breaks are resolved in the selection of the tear-column is of particular importance. Very often the minimum column count is 2 (that is $k = 1$), and there are many columns which share this column count. We would like to choose a column that gives the best chance of the interior block decomposing into much smaller blocks. To this end we aim to minimize the number of non-zeros in the interior block, which can be done by maximizing the number of entries in the border (shaded part of Figure 4). In fact we do this by adding up row counts and column counts for the choices under consideration, which can cause some double counting.

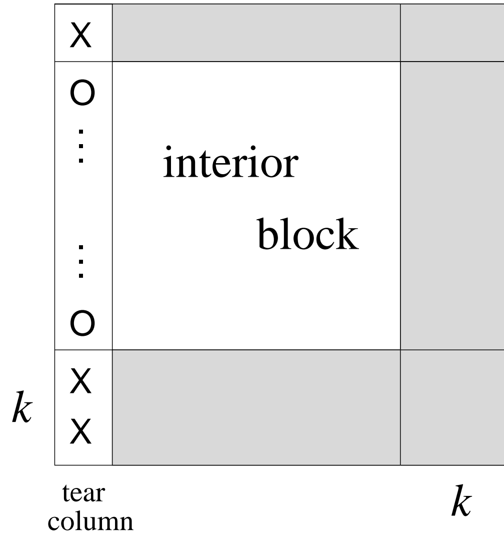

Figure 4: Spike generation by recursion.

Finally we have to decide on which k columns to move to the right hand side (step 3). In the experiments reported in the next section, *symmetric permutations* of rows and columns are made in the algorithm. Thus the decision of step 1 completely determines the outcome of step 3. This strategy is easy to implement and has the advantage that it retains a transversal in the interior block. We refer to this as the Symmetric Recursive Tarjan (SRT) algorithm. Although the SRT algorithm often works well, it can be substantially inefficient in the worst case. For example if the original matrix is an $n \times n$ tridiagonal matrix, then the resulting ordered matrix has $\sim \frac{1}{2}n$ spikes, whereas the ordering shown in Figure 3(c) obtained by a non-symmetric permutation has only 1 spike. It will certainly be necessary to review this decision in future work.

It is interesting to consider the alternative of using the SPK1 algorithm ([7], [4]) in place of the SRT algorithm. The SPK1 algorithm maintains an *active sub-matrix* which is initialized to the original matrix, but in general may have fewer rows than columns. An iteration of the algorithm proceeds as follows.

1. Pick a *tear-column* in the active sub-matrix with minimum column count.

2. Permute all rows that intersect the tear-column to the top of the active sub-matrix, and remove all these rows from the active sub-matrix.

3. Update the column counts for the active sub-matrix.

4. Permute any columns with zero column count in the active submatrix (including the tear-column) to the left of the active sub-matrix and remove these columns from the active sub-matrix.

Ties in step 1 are resolved by maximizing the number of non-zero elements in the rows that will be removed in step 2. The outcome of this process is a block upper Hessenberg matrix in which the blocks are rectangular. When the matrix is factorized, blocks with more rows (r) than columns (c) contribute $r - c$ spikes which are moved to the bottom of the matrix. Likewise blocks with more columns than rows contribute $c - r$ columns which are moved to the right of the matrix. It is easy to construct examples in which the SRT algorithm generates a shorter total spike length than the SPK1 algorithm, and vice-versa.

An example that illustrates the deficiencies of both algorithms is provided by Figure 5.

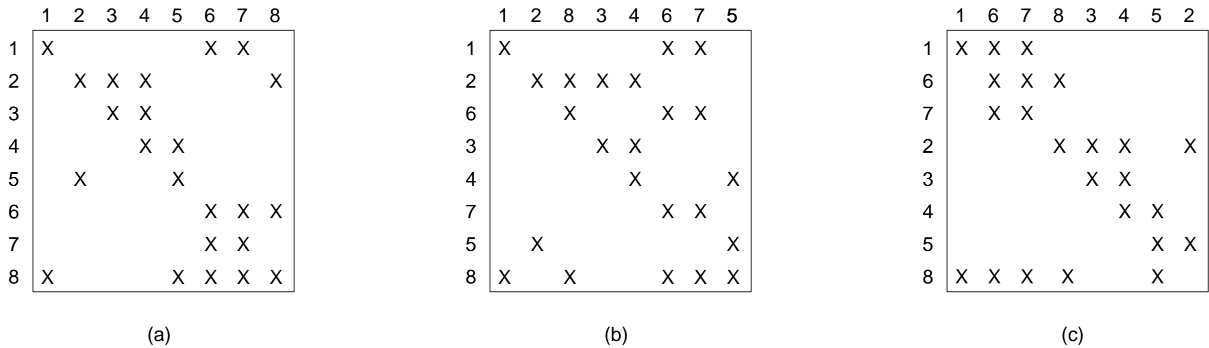

Figure 5: Deficiencies of spike generation algorithms.

The original matrix in (a) is irreducible, and column 1 is the tear-column in both algorithms, giving rise to a spike of length 7. The SRT algorithm then decomposes the inner 6×6 block, as shown in (a), into a 4×4 and a 2×2 block. The latter is fully dense and the former decomposes further on recursion to give an additional spike of length 3. Thus the total spike length for the SRT algorithm is 11. The SPK1 algorithm chooses column 2 as the tear-column for the 6×7 active sub-matrix resulting from the first stage, giving another spike of length 5. The rest of the algorithm provides no further spikes, the final ordering is shown in (b), and the total spike length is 12. The optimum ordering for the matrix is shown in (c) and has a total spike length of 8. Note that the optimum ordering does *not* have a transversal element in the long spike row. The SRT algorithm fails to achieve the optimum result because it does not allow column 8 to be used to simplify the decomposition at the second stage (this would require a non-symmetric permutation of the original matrix). The SPK1 algorithm does not pick the best tear-column at stage two because it fails to recognise that removing column 6 will also remove column 7 and so create a strong component with a spike length of only 1. It also puts the second spike in an inferior position compared to (a). However it does generate fewer spikes than (a) and in fact a detailed examination of the results in the next section indicates that SPK1 usually generates *fewer* spikes than SRT, even when the total spike length is greater.

Some idea of the effectiveness of the methods is gained by looking at sparsity patterns

for different test problems. These are described in more detail in the next section. The left hand diagram of Figure 6 shows the effect of applying Tarjan's algorithm to the 70×70 matrix arising from the `israel` test problem. This illustrates, as is often the case, that the block upper triangular form contains a large irreducible block (here 53×53), and so does not take full account of the sparsity in the matrix. In this case there are two quite dense rows in the block and there exists a tear-column with two non-zeros which removes these rows from the matrix. (This is column 13 counting from the right of the entire matrix.) The right hand pattern shows the effect of the resulting symmetric permutation. Tarjan's algorithm is able to decompose the interior block into an 11×11 block and a 4×4 block, giving a significant improvement in the spike length. Further recursion gives more improvement, albeit to a lesser extent, as shown in the right hand diagram.

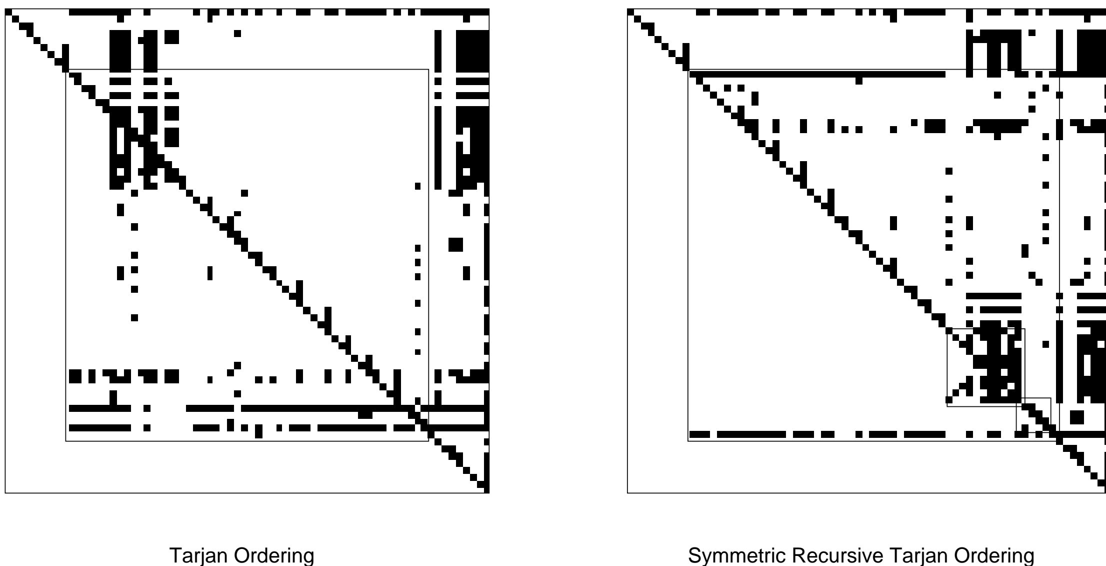

Figure 6: Sparsity patterns for the `israel` test problem.

Another illustration is provided by the 66×66 matrix arising from the `qap4` test problem as shown in Figure 7. The first application of Tarjan's algorithm decomposes the matrix into blocks of dimension 5 and 32. Using the SRT algorithm, the larger block is decomposed into a spike plus a block of dimension 21. Further recursion decomposes this into a spike and a block of length 10. Finally this block decomposes into a single spike and a diagonal interior block. Thus 4 spikes in all are created and the total spike length is 65. All these cases use symmetric permutations when moving the tear-column and creating the spikes, so the initial transversal is maintained, as can be seen in the diagram.

The right hand diagram of Figure 7 illustrates the use of the SPK1 ordering. It tears the 32×32 block in the same way as the SRT algorithm and then picks out some 1×1 blocks on the diagonal. However, it puts the next spike in an inferior position as

compared to the SRT algorithm. Nonetheless, because of the wider choice of tear-column, it is able to avoid creating any further spikes, and the outcome is a total spike length of 60 elements, which improves on the SRT algorithm. However if a non-symmetric permutation were allowed then the superior spike placement of the recursive Tarjan algorithm would achieve a total spike length of 51. Note that the SPK1 algorithm has not maintained the transversal.

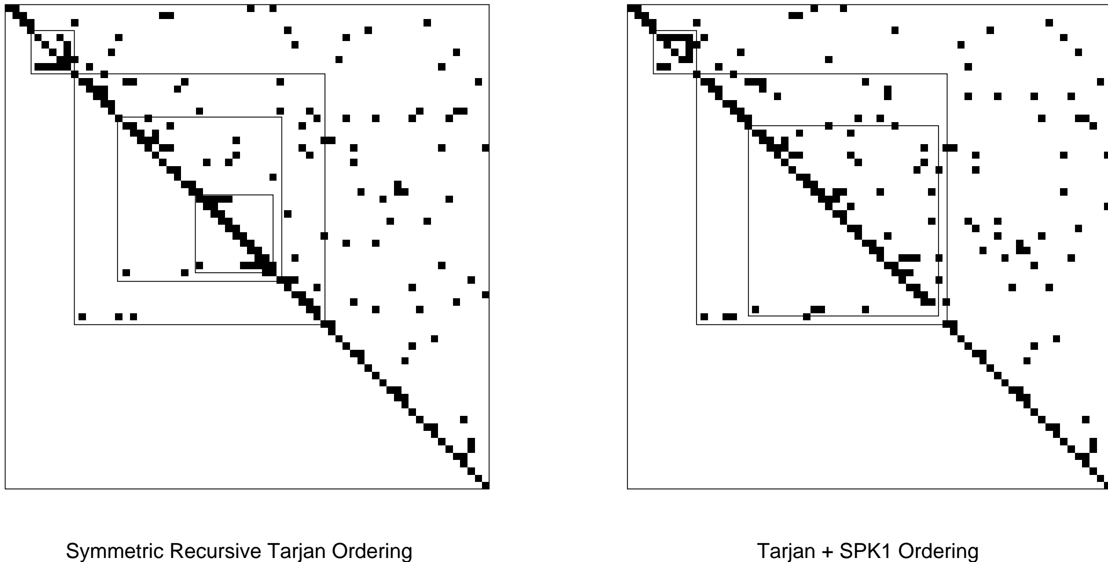

Figure 7: Spike-preserving orderings for the `qap4` test problem.

Clearly it will be important to investigate the use of non-symmetric permutations in the recursive Tarjan algorithm, but it is by no means clear how best to do this, although various possibilities suggest themselves. Also the merits of P^4-like algorithms may need to be reassessed. These topics will be the subject of future research.

6 Practical Experience and Conclusions

In this section we describe some experiments to monitor the total spike length on a range of LP problems and compare the results with the fill-in obtained in regular LU factors using Markowitz pivoting. The issue of updating the factors is then addressed and the likelihood of obtaining a satisfactory sparse update scheme for LP is assessed.

The ability of the methods to maintain a sparse structure (albeit without taking pivoting for numerical stability into account) is tested on a range of smallish to medium sized LP problems from the `netlib` test set. The final active set arising from an active set method is used to determine the matrix A to be factorized. A feature of LP problems is that a not insignificant number of simple bounds are usually active at the solution. These contribute unit columns to A, and it important to take maximum advantage of

their presence. The implicit LU method can readily do this by expressing A in the form given by (3.1) in which there are m_1 unit columns, and A_2 is an $m_2 \times m_2$ matrix. Only the implicit LU factors of A_2 need be available and the solution algorithms (2.10) and (2.11) adapt readily to this form without needing to partition the non-unit columns of A explicitly.

The dimensions of the test problems are given in Table 1, and the columns headed $A_1 + A_2$ and A_2 give the numbers of non-zero elements in the corresponding parts of A. The final column gives the average number of elements per row in the matrix A_2 and provides a measure of the density of A.

	n	m_1	m_2	$A_1 + A_2$	A_2	e/row
share2b	79	25	54	355	235	4.4
qap4	88	22	66	264	205	3.1
adlittle	97	51	46	346	185	4.0
israel	142	72	70	962	583	8.3
share1b	225	131	94	947	420	4.5
qap5	225	77	148	740	487	3.3
brandy	249	87	162	2040	1136	7.0
e226	282	140	142	1943	816	5.7
koc148	308	208	100	664	240	2.4
capri	353	127	226	1517	945	4.2
stair	467	117	350	3833	3560	10.2
bandm	472	167	305	2494	1891	6.2
etamacro	688	311	377	2216	1093	2.9
bp822	1571	923	648	7846	3206	5.0
qap8	1632	890	742	5936	2996	4.0
pilot	3652	2313	1339	41198	17562	13.1

Table 1. LP Test Problems (Final Active Set)

The methods are compared against the Markowitz ordering for calculating sparse LU factors which is widely regarded as the best practical method for limiting fill-in. As with the SRT and SPK1 methods, the Markowitz method has been coded without regard to possible numerical instability. It is assumed that the Markowitz method stores factors of A rather than A_2, but takes account of the unit columns by not storing their unit elements. This would seem to be the most efficient way to proceed (just storing factors of A_2 would require A to be accessed in the solves as in (4.2) and (4.3), and would increase the time required for a solve).

The SRT and SPK1 methods are used in the context of implicit LU factors and so require to store the spikes of the factor $\mathbb{L}_2$ corresponding to A_2, and the diagonal matrix D_2 which contains the diagonal of the implicit upper triangular matrix $\mathbb{L}_2 A_2$ (assuming that any permutations have been subsumed into A). Thus the total storage requirement for the factors is the total spike length plus the length of D_2, which is m_2.

The total storage requirements for the factors is shown in Table 2. It can be seen for the smaller problems that the methods using implicit LU factors improve considerably over the Markowitz method, although the latter is better by a factor of 2 or 3 for some of the larger problems.

	Markowitz	SRT	Tarjan + SPK1
share2b	388	102	107
qap4	312	131	126
adlittle	359	75	76
israel	979	154	136
share1b	1013	170	170
qap5	943	736	544
brandy	2314	1053	894
e226	2071	280	380
koc148	685	182	164
capri	1569	330	352
stair	5746	10291	18550
bandm	2732	863	943
etamacro	2282	457	468
bp822	9157	8307	10723
qap8	13709	50731	35027
pilot	62040	117917	126565

Table 2. Storage Requirements for Factors

However these figures do not tell the whole story and it is also necessary to estimate the amount of time required to solve systems with the resulting factors. This can be done in a crude way by counting the number of accesses required by the various algorithms. For regular LU factors, each nontrivial element of the factors is accessed just once, so the same count applies as in Table 2. For implicit factors, elements of A_1 and A_2 are also accessed once, so this count has to be added on. The resulting numbers are shown in Table 3. Now Markowitz wins in all cases, with the margin increasing for the larger problems.

The implication of these numbers for practical problems is difficult to assess because there are so many imponderables on both sides. First of all there is the substantial cost of indexing which has been ignored here. Markowitz is in the nature of a complete pivoting algorithm and needs a very intricate linked list structure to work efficiently. Moreover this structure is inefficient for carrying out solves with A, and it is usual to transform to a more simple data structure at this stage. This all takes time. On the other hand the calculation of the Tarjan orderings is also not negligible, and in some cases quite a lot of recursion takes place (anything from 3 levels on the smaller problems up to 194 levels for `pilot`, although it is easily possible to impose an upper limit on this number).

	Markowitz	SRT	Tarjan + SPK1
share2b	388	457	462
qap4	312	395	390
adlittle	359	421	422
israel	979	1116	1098
share1b	1013	1117	1117
qap5	943	1476	1284
brandy	2314	3093	2934
e226	2071	2223	2323
koc148	685	846	828
capri	1569	1847	1869
stair	5746	14124	22383
bandm	2732	3357	3437
etamacro	2282	2673	2684
bp822	9157	16153	18569
qap8	13709	56667	40963
pilot	62040	159115	167763

Table 3. Number of Accesses in a Solve

Moreover the influence of numerical stability and threshold pivoting on the amount of fill-in also needs to be taken into account. Finally it may well be that an unsymmetric version of the recursive Tarjan algorithm would improve the spike length to some extent. However for a one-off calculation it seems unlikely that an implicit LU based method would improve on the Markowitz method as the problem dimension gets large, although it might not be grossly inferior.

In an LP context, an even more important aspect is the need to update the factors when one column of A is replaced by another, as usually takes place on each iteration of an active set or simplex-like method. The Markowitz approach does not lend itself to updating because the changing fill-in in the LU factors is difficult to handle efficiently. Usually some sort of a *product form* method is employed in which a file of pre-operations on the factors is built up (see for example Suhl [8]). When this file becomes too large, the current matrix is *reinverted* to determine new LU factors.

There is some hope that the implicit LU approach will provide a much more suitable representation for updating the factors in an LP context. Fletcher [3] shows how implicit LU factors may be updated much more efficiently in the dense case than for regular LU factors. Moreover these updates are achieved by a sequence of row operations on the matrix $\mathbb{L}$, and the spike data structure is ideal for this when the spikes are held in dense format. Experience with storing the implicit LU factors of A_2 in dense format, with A in sparse format, has already proved to give a code which is very reliable and efficient. It should speed up very considerably when going to a spike representation of $\mathbb{L}$.

The updating aspect also affects the way in which the irreducible block structure is calculated. Replacing one column of A by another can remove at most one element from the transversal. Renewing the transversal therefore only needs one application of the reassignment search described in Section 3. Moreover, for sparse matrices, only a part of the block structure is usually affected by the update and this also reduces the calculation time.

There are still many uncertainties and it is not yet clear exactly what form an implicit LU based sparse-update method should take. Research into this project is on-going and there is hope that the approach will provide reasonably efficient and easily maintained software for sparse LP calculations.

7 References

[1] Duff I.S., Erisman A.M. and Reid J.K. (1986), *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford.

[2] Erisman A.M., Grimes R.G., Lewis J.G. and Poole W.G.Jr. (1985), A structurally stable modification of Hellerman–Rarick's P^4 algorithm for reordering unsymmetric sparse matrices, *SIAM J. Numer. Anal.*, **22**, 369–385.

[3] Fletcher R. (1997), Dense Factors of Sparse Matrices, in *Approximation Theory and Optimization. Tributes to M.J.D. Powell*, (M.D. Buhmann and A. Iserles, eds.), Cambridge University Press.

[4] Fletcher R. and Hall J.A.J. (1993), Ordering algorithms for irreducible sparse linear systems, *Annals of OR*, **43**, 15–31.

[5] Hellerman E. and Rarick D.C. (1972), The partitioned preassigned pivot procedure (P^4), in *Sparse Matrices and their Applications*, (D.J. Rose and R.A. Willoughby, eds.), Plenum Press, New York.

[6] Lin T.D. and Mah R.S.H. (1977), Hierarchical partition – a new optimal pivoting algorithm, *Math. Programming*, **12**, 260–278.

[7] Stadtherr M.A. and Wood S.E. (1984), Sparse matrix methods for equation-based chemical flowsheeting. I. Reordering phase, *Comp. Chem. Eng.*, **8**, 9–18.

[8] Suhl U.H. (1994), MOPS – Mathematical OPtimization System, *European J. Operational Research*, **72**, 312–322.

[9] Tarjan R.E. (1972), Depth first search and linear graph algorithms, *SIAM J. Comput.*, **1**, 146–160.

[10] Wilkinson J.H. (1965), *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford.