A New Implementation of Sparse Gaussian Elimination

ROBERT SCHREIBER
Stanford University

An implementation of sparse LDL^T and LU factorization and back substitution, based on a new scheme for storing sparse matrices, is presented. The new method appears to be as efficient in terms of work and storage as existing schemes It is more amenable to efficient implementation on fast pipelined scientific computers

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*sparse and very large systems*; G.4 [**Mathematics of Computing**], Mathematical Software—*algorithm analysis*

General Terms· Algorithms, Theory

Additional Key Words and Phrases· Sparse matrix, sparse systems of linear equations

1. INTRODUCTION

Let A be an $n \times n$, irreducible, symmetric positive definite matrix. The system

can be solved by a Cholesky factorization

$$A = LDL^T, \qquad (1.2)$$

with L unit lower triangular and D diagonal, and forward and backward solves

$$Lz = b, \qquad L^T x = D^{-1} z. \qquad (1.3)$$

When only a small fraction of the elements of A are nonzero, A is said to be sparse. When a sparse matrix is factored, "fill-in" occurs: the triangular factors contain nonzeros in positions where A has zeros. We use the term *sparse Gaussian elimination* to refer to methods for triangular factorization and back substitution that take maximum advantage of zeros in the matrix and the factors. We present a new way to implement sparse Gaussian elimination in this paper.

Ordinarily, the rows and columns of A are first permuted so that the fill-in is made small. We are not concerned with the problem of finding such permutations.

Sparse Cholesky factorization is ordinarily implemented as a two-step process. First, the nonzero structure of the factor is computed by a symbolic factorization

Partial support has been provided by the NASA-Ames Research Center, Moffet Field, Calif., under Interchange No NCA2-OR745-002.

Author's address Department of Computer Science, Stanford University, Stanford, CA 94305.

$$A = \begin{bmatrix} x & & \text{sym} & \\ 0 & x & & \\ x & x & x & \\ x & 0 & x & x \end{bmatrix}$$

aa:	a_{11}	31	41	22	32	33	43	44
row.	1	3	4	2	3	3	4	4
colbegin	1	4	6	8				

Fig. 1. Standard column-oriented storage for a sparse matrix.

step. Using a data structure provided by the symbolic step, a numeric factorization computes the (nonzero) elements of L. The elements, including any fill-ins, are stored in predetermined locations.

The usual data structure is this. Elements of the lower triangle of A, including zeros that later fill in, are stored in a one-dimensional array, which we call *aa*. A separate array, *row*, records the row to which the corresponding element of array *aa* belongs: if a_{64} is stored at $aa(12)$, then $row(12) = 6$. Columns of A occupy successive contiguous blocks of *aa* and are sorted by row. Pointers to diagonal elements are stored in an array, *colbegin*. Figure 1 gives an example of this scheme.

This paper proposes a method for implementing sparse elimination which uses a new scheme to store A and L. A tree structure links every column $k < n$ of A to the first column $j > k$ such that $l_{jk} \neq 0$. We say that $j = next(k)$. Then, instead of the row to which a nonzero element belongs (its ***absolute*** row index), a pointer to the location in *aa* of the nonzero in the same row (say row i, $i > k$) and the *next* column is stored: the pointer for l_{ik} points to the storage for l_{ij}. Later it is shown that if $l_{ik} \neq 0$, then $l_{ij} \neq 0$. We call these pointers *relative* row indices. This scheme bears some similarities to a scheme based on the idea of a "representative" column due to George and Liu [10].

The principal advantage of a relative row-index scheme is the efficiency with which a column (k, above) can be added to or subtracted from its *next* column (j, above). The pseudo-ALGOL for such an operation is

```
for $i$ := 1 to $nk$ do
    $aa(j + ptr(k + i)) := aa(j + ptr(k + i)) + aa(k + i)$;
```

assuming column k has nk off-diagonal nonzeros and variables j and k point to the beginnings of columns k and j in *aa*.

When an absolute row-index scheme is employed, the code for adding or subtracting two columns is more complicated. Gustavson [12] and Eisenstat et al. [3] avoid a complicated inner loop by unpacking one of the columns into a temporary array of size n and using a loop similar to the one above. This scheme suffers from a significant drawback: the code accesses this large temporary in a random way, degrading performance on a machine with a cache memory.

The new scheme has several advantages. Because it accesses memory almost sequentially, it makes good use of a cache memory. Efficient implementation on a vector machine is possible. The pointers it uses are all small integers; in a storage-critical situation more of them can be packed into a word. When redundancies in the relative row indices are fully exploited, their number can be reduced. For $k \times k$ grid problems k^2 pointers are required. Previous methods have required at least $12k^2$ row indices [10, 16]. Section 5 gives a detailed analysis of storage requirements for these problems. An analysis of implementation on vector computers is given in Section 6. A discussion of symbolic factorization and a numerical experiment to determine pointer storage requirements for general sparse problems is given in Section 7.

Relative row-index schemes are not useful for solving $Ax = b$ with arbitrary nonsymmetric A. But if A has a symmetric nonzero structure and can be factored as LU (without partial pivoting) so that the structure of U, which is the transpose of that of L, can be precomputed, then a method based on the techniques of this paper is possible.

A scheme for finite-element systems of Eisenstat et al. [4] has several of the characteristics and advantages of the proposed scheme, as do the schemes of Peters [14], and Duff [2].

2. THE NEW SCHEME

Let L be the Cholesky factor of A. We define an n-vertex undirected graph $G = G(L) = (V, E)$, with vertices $V \equiv \{1, 2, \ldots, n\}$ and edges $E \equiv \{(i, j) \mid i > j \text{ and } l_{ij} \neq 0\}$. Define

$$col(j) \equiv \{i > j \mid l_{ij} \neq 0\}, \qquad 1 \leq j < n,$$

$$row(j) \equiv \{k < j \mid l_{jk} \neq 0\}, \qquad 1 < j \leq n,$$

$$next(j) \equiv \min\{i \in col(j)\}, \qquad 1 \leq j \leq n - 1,$$

and

$$N(L) \equiv \{(j, next(j)) \in E \mid 1 \leq j \leq n - 1\}.$$

The edges $(j, next(j))$ play a special role in the scheme: the relative row indices associated with nonzeros in column j of L will point to nonzeros in column $next(j)$ of L.

We now review some graph terminology. Vertices k, j are *adjacent* if $(k, j) \in E$. A *$k - j$ path* in a graph G is a sequence $k = v_0, v_1, \ldots, v_l = j$ of vertices with v_{i-1} adjacent to v_i, $1 \leq i \leq l$. It is *monotone* if $v_i > v_{i-1}$, $1 \leq i \leq l$. We use the words smaller and larger for comparing vertices.

A graph is *strongly connected* if, for every pair k, j of vertices, there is a $k - j$ path. A *tree* is a strongly connected n-vertex graph with $n - 1$ edges. Trees have no cycles: there is a unique path between every pair of vertices. A tree T is *ordered with root n* if, for every vertex j, the $j - n$ path is monotone.

If $G = (V, E)$ and $V_1 \subset V$, then the *subgraph induced by V_1*,

$$G_{V_1} \equiv (V_1, E \cap (V_1 \times V_1)).$$

A *clique* is a subset $V_1 \subset V$ such that

$$E \cap (V_1 \times V_1) = V_1 \times V_1;$$

in other words, every vertex in V_1 is adjacent to every other vertex in V_1.

Since A is irreducible, L is, too. It follows that $G(L)$ is strongly connected.

The fill-in obeys an important law.

PROPOSITION 1. *If there is a $j - k$ path in $G(L)$ through vertices smaller than both j and k, then $(j, k) \in E$.*

PROOF. This is an easy consequence of the corresponding statement about paths in the graph of A [14, Lemma 4]. □

COROLLARY. *If $j = next(k)$ and $l_{ik} \neq 0$, then $l_{ij} \neq 0$.*

The corollary is essential; without it we couldn't necessarily define a relative row index for the nonzeros of a column $k < n$. With it, we know that for every nonzero in column k there is a corresponding nonzero in column $next(k)$:

$$col(k) \subset col(next(k)) \cup \{next(k)\}.$$

PROPOSITION 2. *For each $1 \leq k < n$, $col(k)$ is not empty.*

PROOF: $G(L)$ is strongly connected. Choose any vertex $l > k$ and consider a $k - l$ path in $G(L)$. Let l' be the first vertex on the path larger than k. Since there is a $k - l'$ path in $G(L)$ consisting of vertices smaller than k or l', $(k, l') \in E$, and so $l' \in col(k)$. Q.E.D.

Thus, the definition of *next* makes sense, since the sets whose minima are required are not empty.

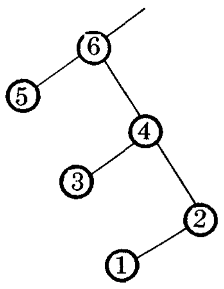
Using relative row indices it is easy to add a multiple of a column to its *next* column. The inner loop of our scheme does this. The inner loop of Cholesky factorization ordinarily subtracts from the pivot column (the jth) a multiple of each column k of L such that $k \in row(j)$. The key idea is to accumulate instead a sum of multiples of columns $k \in row(j)$.

Define the graph $T = (V, N(L))$. We are going to show that T is an ordered tree with root n, and for every $1 \leq j \leq n$, the subgraph $T_{row(j) \cup \{j\}}$ is an ordered tree with root j. Thus, for every $k \in row(j)$ there is a unique $k - j$ path in T that goes through other vertices in $row(j)$. A sum of appropriate multiples of columns of L in $row(j)$ is accumulated by a depth-first traversal of $T_{row(j) \cup \{j\}}$.

Here is an example. Suppose

$$L = \begin{bmatrix} x & & & & & & & \\ x & x & & & & & & \\ 0 & 0 & x & & & 0 & & \\ 0 & x & x & x & & & & \\ 0 & 0 & 0 & 0 & x & & & \\ x & x & x & x & x & x & \cdot & \cdot\ \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & & \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & & \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & & \end{bmatrix}$$

Then $T =$



Since $row(6) = \{1, 2, 3, 4, 5\}$, the algorithm will subtract a multiple of each of those columns from column 6. It can subtract the multiple of column 5 easily since $6 = next(5)$. Next, it can *add* the multiple of column 1 to the multiple of column 2, add the sum (of 1 and 2) to column 4, add column 3 to column 4, and, finally, subtract the 1–4 sum from column 6. The algorithm only uses the operations of adding or subtracting a column to or from its *next* column. Note that it was essential that $T_{row(6)\cup\{6\}}$ be an ordered tree with root 6.

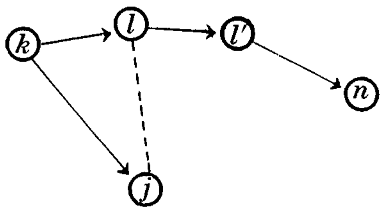
Now the proofs.

PROPOSITION 3. *T is an ordered tree with root n.*

PROOF. Construct T. Start with the single vertex n. Add the preceding vertices $n - 1, n - 2, \ldots, 1$, each with its incident *next* edge. The single vertex n is an ordered tree with root n. Adding a vertex and *next* edge leaves the graph ordered with root n. Q.E.D.

PROPOSITION 4. *For every* $1 \le j \le n$, $T_{row(j)\cup\{j\}}$ *is an ordered tree with root* j.

PROOF. Let $k \in row(j)$. Consider the (unique, monotone) path from k to n in T. Let $l < j$ be on this path. Then the edge (j, k) together with the $k - l$ path is a $j - l$ path through smaller vertices; hence $(l, j) \in E$, and $l \in row(j)$. [This diagram illustrates the argument:



Vertices increase going left to right.] We claim that j is actually *on* the $k - n$ path, showing that a unique monotone $k - j$ path in $T_{row(j)\cup\{j\}}$ exists, which proves the proposition. Suppose not. Then an edge (l, l') on the path with $l < j < l'$ exists. But $l' = next(l)$, so $l' \le j$, a contradiction. Q.E.D.

A more realistic and interesting example, a 3×3 finite-difference grid, is shown in Figure 2.

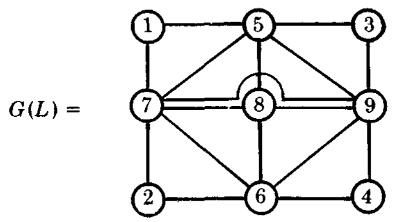
For the backsolving scheme, we need one last fact.

PROPOSITION 5. *For each* $1 \le j < n$, $col(j)$ *is a subset of the vertices on the path from vertex j to the root n of T.*

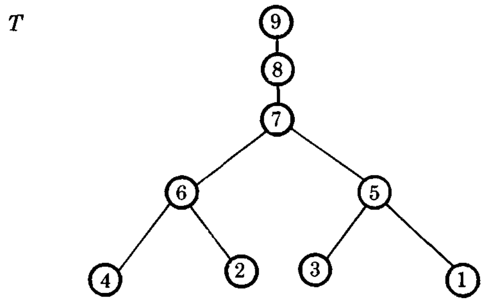
PROOF. Use induction on the depth of j in T. Certainly, $next(j) \in col(j)$ and $next(j)$ is the first vertex on the $j - n$ path. Moreover, by the corollary to Proposition 1, $col(j) \subset col(next(j)) \cup \{next(j)\}$. Q.E.D.

3. AN IMPLEMENTATION OF THE NEW SCHEME

In this section we present the details of an implementation of the new sparse LDL^T factorization method. The data structure is covered in Section 3.1. The algorithm is described in Sections 3.2 and 3.3.



k:	1	2	3	4	5	6	7	8
$next(k)$:	5	6	5	6	7	7	8	9



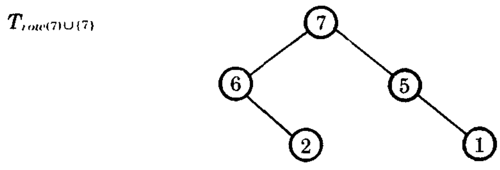


Fig. 2. A 3 × 3 finite-difference grid.

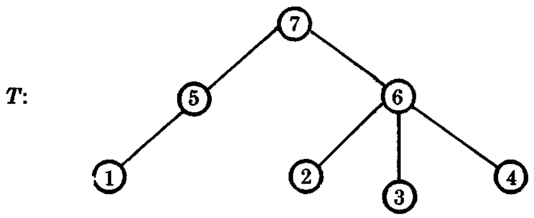
3.1 Storage Scheme

The nonzeros of L and D are stored in a one-dimensional array, *aa*. Initially, the array contains the corresponding elements of the lower triangle of A; the code overwrites them with L and D. The columns are stored together, sorted by row.

For each $1 \le j \le n$, $locdiag(j) + 1$ is the location in *aa* of a_{jj} and d_{jj}. Thus, the ith nonzero of column j is stored in $aa(locdiag(j) + i)$. Also, $locdiag(n + 1)$ is the location of the last element in *aa*. An integer array *wherenext* contains the relative row indices. Suppose the nonzero in position m of *aa* is a member of column k of L, and that $j = next(k)$. Moreover, suppose this nonzero is in the ith row of L. Somewhere in the storage for column j, say at the lth position, is a location for the element l_{ij} In other words, l_{ij} is stored at $aa(locdiag(j) + l)$. Then

$$wherenext(m) \equiv l.$$

Note that *wherenext* need not be defined for elements on the diagonal; they are never subtracted from elements in other columns.



	1	2	3	4	5	6	7
son	—	—	—	—	1	3	5
brother	—	—	4	2	6	—	—

Fig. 3. Storage of *T*.

	aa	*where-next*
1	11	—
	51	1
	71	2
	22	—
5	62	1
	72	2
	33	—
	53	1
	93	4
10	44	—
	64	1
	94	4
	55	—
	75	1
15	85	2
	95	3
	66	—
	76	1
	86	2
20	96	3
	77	—
	87	1
	97	2
	88	—
25	98	1
	99	—

	locdiag
1	0
2	3
3	6
4	9
5	12
6	16
7	20
8	23
9	25
10	26

son	*brother*
—	—
—	—
—	1
—	2
3	—
4	5
6	—
7	—
8	—

Fig. 4. Storage scheme applied to the 3 × 3 grid example of Figure 2.

T is encoded as a binary tree. An array *son*(j) contains a pointer to any of the sons of vertex j in *T*. Remaining sons are linked in a linear list, with pointers stored in *brother* (see Figure 3). Note that the *next* links are not stored. Figure 4 illustrates this storage scheme applied to the example of Figure 2. The numbers in *aa* are the indices (i, j) of the element l_{ij} stored at that position. It may be advantageous to store diagonal elements in a separate array; the n unused pointers could then be eliminated.

3.2 The Numeric Factorization Algorithm

The following pseudo-ALGOL procedures perform the Cholesky factorization of the matrix $[a_{ij}]$, overwriting a_{ij} with l_{ij} for $i > j$ and a_{ii} with d_{ii}. It uses a column-oriented method. At the jth step it subtracts from column j a multiple of each column $k \in row(j)$ by first accumulating such column multiples.

```
1.  procedure factor $(a, n)$
2.  integer $n$; real array $a$;
3.    begin integer $i, j, k$;
4.    for $j := 1$ to $n$
5.      begin real array $t$;
6.      for $k \in row(j)$
7.        if $next(k) = j$ then begin
8.          searchtree $(j, k, t)$;
9.          for $i \in col(j)$
10.           $a_{ij} := a_{ij} - t_i$
11.         end;
12.     for $i \in col(j)$ $a_{ij} := a_{ij}/a_{jj}$
13.     end
14.   end (factor);
1.  procedure searchtree $(j, k, t)$
2.  integer $j, k$; real array $t$;
3.  comment add to $t$ multiples of all columns $l \in row(j)$ in the subtree of $T$ rooted at
    column $k$;
4.    begin integer $i, l$; real amult;
5.    real array $t^{j,k}$;
6.    comment the temporary $t^{j,k}$ accumulates the multiples of columns;
7.    if $k \in row(j)$ then begin
8.      amult $:= a_{kk} * a_{jk}$;
9.      for $i \in col(k)$ $t_i^{j,k} :=$ amult $* a_{ik}$;
10.     for $l < k$ if $next(l) = k$ then searchtree $(j, l, t^{j,k})$;
11.     for $i \in col(k)$ $t_i = t_i + t_i^{j,k}$
12.     end
13.   end (searchtree);
```

We now briefly discuss how membership of column k in $row(j)$ can be efficiently determined (line 7 of searchtree), how elements a_{ij} can be located in aa, and how temporary storage (the t vectors) can be managed.

Define, for every $k \in row(j)$

$$first(k, j) \equiv \text{the relative position of } l_{jk} \text{ in column } k\text{'s storage}$$

and let $first(j, j) = 1$. Note that, if $j = next(k)$, then $first(k, j) = 2$. The part of column k that is subtracted from column j begins at $locdiag(k) + first(k, j)$, and has length

$$num(k, j) \equiv locdiag(k + 1) + 1 - (locdiag(k) + first(k, j)).$$

Also, define

$$numcol(k) \equiv locdiag(k + 1) - locdiag(k),$$

which is the number of nonzeros, counting the diagonal, in column k of L.

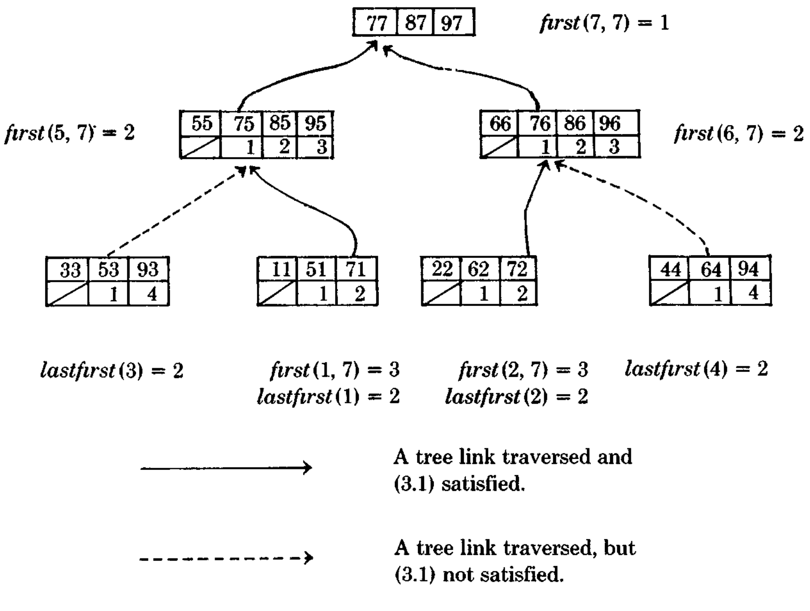


Fig 5. Data structures used during depth-first search

For each pivot column j, the program does a depth-first search of the subtree $T_{row(j)\cup\{j\}}$ starting at the root, vertex j. At every internal node k, a temporary vector *tempk* of size $num(k, j)$ is allocated and initially filled with the contribution of column k to the pivot column. The sons of vertex k in $T_{row(j)\cup\{j\}}$ are then all searched, the temporary *tempk* being passed to each of them. Then the elements of *tempk* are added to the temporary passed by vertex-k's father; correspondence of elements is determined using column k's *wherenext* pointers.

In some implementations of sparse elimination a data structure for representing $row(j)$ is maintained during numeric factorization; the columns belonging to $row(j)$ are explicitly available from that data structure [15]. In this implementation membership in $row(j)$ is determined as part of the tree search process.

When a son k' of k is searched, the pointer $first(k, j)$ is passed. Whether k' is an element of $row(j)$ at all can be determined by attempting to find the location $first(k', j)$ of the element $l_{j,k'}$. For, if $k' \in row(j)$, then $l_{j,k'} \neq 0$, so there exists p such that

$$wherenext(locdiag(k') + p) = first(k, j), \tag{3.1a}$$

$$1 \le p \le numcol(k'). \tag{3.1b}$$

If such a p exists, then we set $first(k', j) = p$ and continue the search process. If not, $k' \notin row(j)$, and the search immediately backtracks to column k.

It is not actually necessary to search the pointers of column k' to either find a p satisfying (3.1) or determine that none exists—only one value of p need be examined, one larger than the last to have satisfied (3.1). We store this value of p in $lastfirst(k')$; initially it is 1. Figure 5 illustrates the depth-first search of $T_{row(7)\cup\{7\}}$ for the 3×3 example of Figure 2.

Note that the overhead associated with tree searching is small, since the operations performed on every traversal of a tree edge are not dependent on the number of nonzeros in the corresponding columns.

The storage requirement of the method is certainly no greater than for absolute row-index methods. In fact, elements of *wherenext* are all less than the maximum of $numcol(j)$, $1 \leq j \leq n$, which will ordinarily be much less than n, so more can be packed in a word. The only other minor issue is that of temporaries. These can be allocated off a stack. If the depth of T is d, then at most d temporary vectors are needed. The symbolic factorization can determine the amount of stack space that will be required.

3.3 Backsolving

It is not evident that the relative row-index scheme is at all suitable for backsolving

$$Lz = b, \tag{3.2a}$$

or

$$L^{T}x = D^{-1}z. \tag{3.2b}$$

It appears at first that, to access b, z, and x (which share the same n storage locations) absolute row indices are required. These could be precomputed (by the symbolic factorization routine) and stored, or they could be generated from the relative row indices after the factorization is accomplished. The first alternative costs storage, the second, time.

It is, however, possible to solve both (3.2a) and (3.2b) using relative row indices. Moreover, the resulting algorithm accesses storage in a more nearly sequential manner than the obvious absolute row-index algorithm. Other advantages of relative row indices have already been mentioned. Thus, this scheme is as attractive for backsolving as for factorization.

The forward solve (3.2a) is done by a depth-first search of T. The basic process is, as in the factorization, the accumulation in temporary storage of multiples of columns of L. Here is a pseudo-ALGOL procedure. A call to fwd-solve(n) solves (3.2a). Let T_j be the subtree of T rooted at j.

```
procedure fwd-solve($j$); comment $t^j$ stores $\{t_i^j \mid i \in col(j) \cup \{j\}\}$;
begin real array $t^j$;
    comment determine $z_k$ for every $k$ in $T_j$ and, for all $i \in col(j) \cup \{j\}$
    set $t_i^j = \sum_{k' \in row(j) \cup \{j\}} z_{k'} l_{ik'}$;
begin
    for $i \in col(j) t_i^j := 0$;
    $t_j^j := 0$;
    for each son $l$ of $j$ begin
        fwd-solve ($l$);
        for $i \in col(l) t_i^j := t_i^j + t_i^l$
        end
    $z_j := b_j - t_j^j$;
    for $i \in col(j) t_i^j := t_i^j + z_j l_{ij}$;
    $t_j^j := t_j^j + z_j$;
    end
```

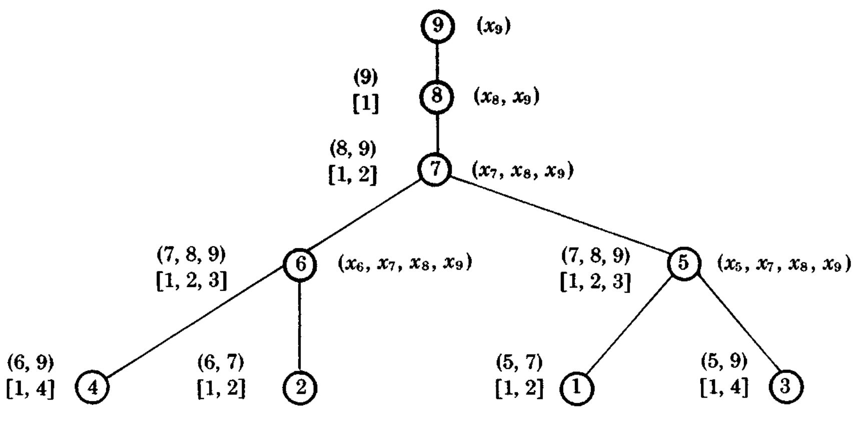


Fig. 6. Tree search for $L^Tx = D^{-1}z$.

Note that the accumulation of temporary column multiples (line 8) is done using relative row indices. A temporary vector for storing $\{t_i^j \mid i \in col(j) \cup \{j\}\}$ is allocated when the procedure is invoked and released after being used by the caller.

Backsolving (3.2b) is also done by a depth-first traversal of T. In contrast to the forward solve, no information need propagate up: as soon as a vertex k of T is searched, the value of x_k is determined.

To begin, $x_n = z_n/d_{nn}$. Now suppose vertex k is searched. Then

$$x_k = \frac{z_k}{d_{kk}} - \sum x_i l_{ik}, \tag{3.3}$$

where the sum is taken over indices i such that $l_{ik} \neq 0$, that is, for $i \in col(k)$. The elements of L involved, $\{l_{ik},\ i \in col(k)\}$, are a contiguous $(numcol(k) - 1)$-long vector. The elements $x^{(k)} \equiv \{x_i,\ i \in col(k) \cup \{k\}\}$ are also needed. Since (Proposition 5) $col(k)$ is a subset of the vertices on the path from k to the root n, $x^{(k)}$ is already known.

The solution vector x is stored in an n-vector b. It is not convenient to get at $x^{(k)}$ by accessing this array: absolute indices are required, and the elements needed are scattered randomly. Let $j = next(k)$. The algorithm passes a temporary vector containing $x^{(j)}$ when searching vertex k. The elements of $x^{(k)}$ reside in positions pointed to by column k's relative row indices. The inner loop extracts these elements, performs the dot product in (3.3), and creates the temporary vector holding $x^{(k)}$, which will be passed to the sons of vertex k in T.

Figure 6 illustrates the backsolve for Example 1. T is shown. To the right of vertex j is $\mathbf{x}^{(j)}$; to the left, $col(j)$ is shown in parentheses; the relative row indices are in square brackets. When vertex 3 is searched, x_5 and x_9 are extracted from $x^{(5)}$ using the relative row indices, 1, 4, of column 3. Then $x_3 = z_3/d_{33} - (l_{53}x_5 + l_{93}x_9)$.

4. REFINEMENTS TO THE METHOD

Finite-element problems, especially when ordered by nested dissection techniques, lead to matrices with many columns having relative row pointers of 1, 2, 3, ..., ν. Such a column has below its diagonal the same nonzeros as its *next* column. This situation can be exploited in three ways. Storage for the pointers can be saved—in fact, they are not needed at all. The inner loop used to add such a column to its *next* column can be simplified. The code

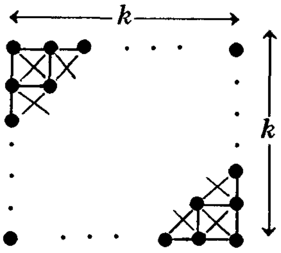
```
for $i$ := 1 to $nk$ do
    $aa(j + i) := aa(j + i) + aa(k + i)$;
```

is used—no pointers are needed. Finally, the temporary vector allocated for the *next* column can be passed to the sons of this column. The *sons'* relative row indices can be used to access it.

Nontrivial relative row-index sets can be redundant: two different columns may have exactly the same set of relative row indices, and so only one set would have to be stored. The difficulty in exploiting this redundancy is in recognizing the columns with identical relative row indices. One possibility is to exploit symmetries of the graph $G(L)$. If there is a k-fold symmetry in the graph, and the nodes are suitably numbered, then in general a vertex will have a row-index set identical to that of its $k - 1$ images. The model problem of Section 5 is another such situation.

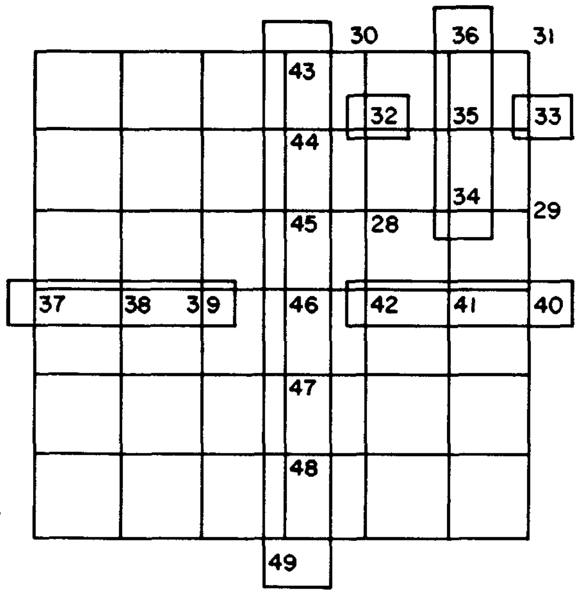
5. A MODEL PROBLEM

For a symmetric $n \times n$ matrix A we define the undirected graph $G(A) = (V, E(A))$, where $V = \{1, 2, \ldots, n\}$, and $E(A) = \{(i, j) \mid a_{ij} \neq 0\}$. The model problem is a symmetric positive definite problem with $n = k^2$ and a $k \times k$ "grid-graph."

$G(A) =$ 

Thus, vertices are adjacent to every other vertex with which they share a square cell, or element.

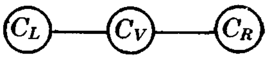
With a row-by-row ordering of the vertices, A is banded with bandwidth $k + 1$. To best utilize sparse matrix techniques, the vertices of $G(A)$ are ordered by nested dissection [1, 5]. The vertices of a separating cross are numbered last. The remaining vertices constitute four independent grid-graphs of size $(k - 1)/2$ by $(k - 1)/2$. These are numbered by (recursively applying) nested dissection. For example, with $k = 7$, $G(A)$ is



(The numbering of the other 3 × 3 subproblems is obvious.) The first level's separating cross consists of a vertical separator C_V (nodes 43–49), a left-horizontal separator C_L (nodes 37–39), and a right-horizontal separator C_R (nodes 40–42). The whole cross is denoted by C, where

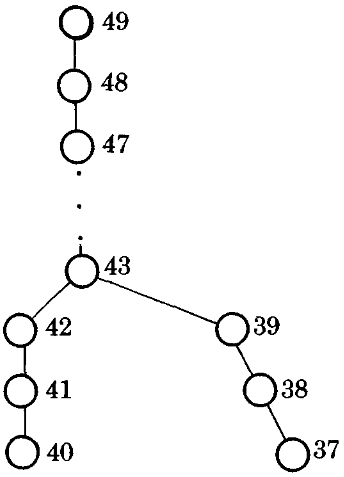
$$C \equiv C_V \cup C_L \cup C_R.$$

The structure of the Cholesky factor L of A, and hence of its graph G, can be surmised from Proposition 1. G_C, the subgraph of G describing the last $2k - 1$ rows and columns of L, has the structure



The "vertices" of this graph represent cliques, and the heavy lines indicate that all possible edges are present. We suppose that the three separators of a cross are always numbered in this sequence: vertices of C_L, vertices of C_R, vertices of C_V.

In effect, nested dissection defines a binary tree of grids; its structure is mirrored in the structure of the tree T. For the example with the numbering shown, T_C is



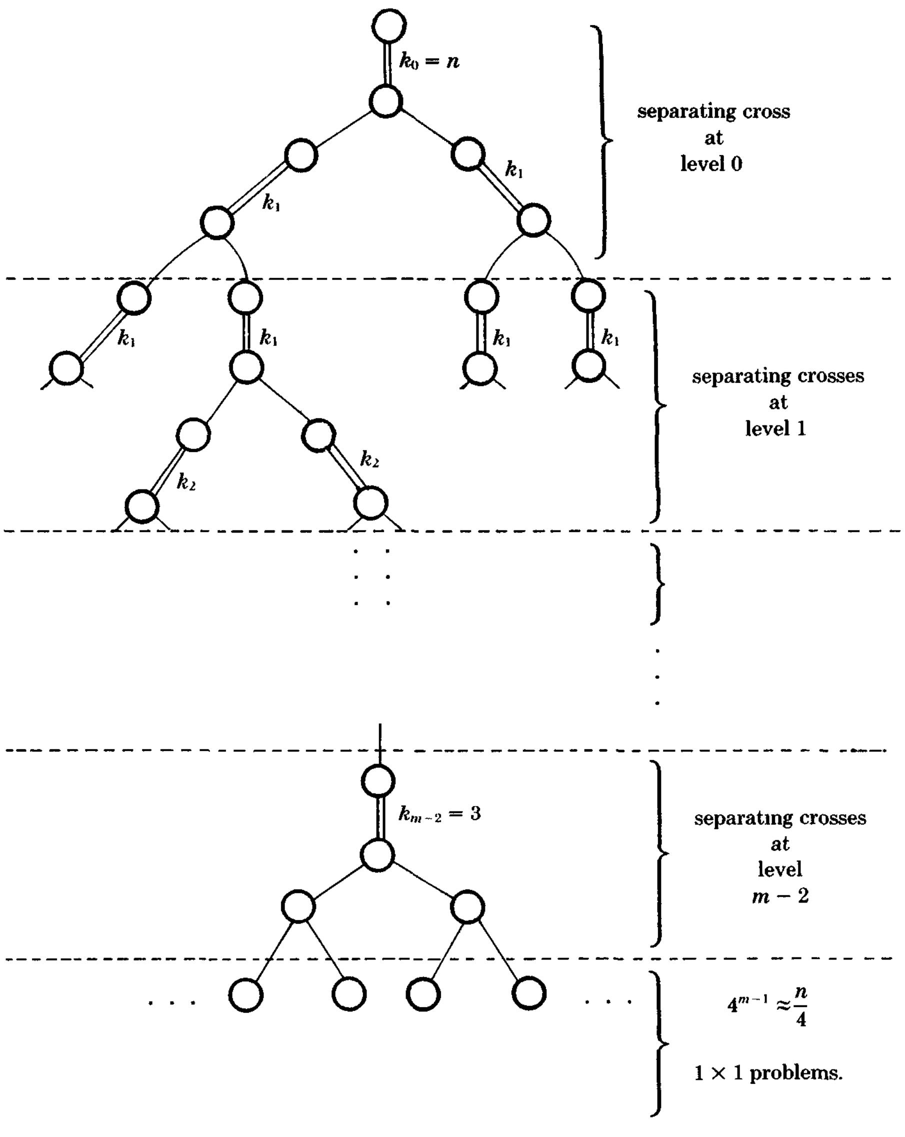
There is a chain of seven vertices (those in C_V) and two subchains, of three vertices each (one for C_L and one for C_R). Denote by $\circ\overset{r}{=}\circ$ a chain of length r, let $k = 2^m - 1$, and $k_j \equiv 2^{m-j} - 1$. The tree for nested dissection of a $k \times k$ model problem is shown in Figure 7.

A number of questions arise. For example, how well do the optimizations of Section 4 work for these problems? What is the amount of storage used for relative row indices? How large must the stack be? What will be the cost, in running time, for each multiplication, compared with that of a standard implementation?

We first show that only $12k^2$ row indices remain if trivial sets are not stored. At level l in the dissection, square subgraphs of size $k_l = 2^{m-l} - 1$ remain; Figure 8 illustrates the case $l = m - 2$. The numbers shown indicate the *order* in which these vertices are eliminated, rather than their number in the overall ordering of the grid.

When vertex 7 is eliminated, it is adjacent to all the vertices 8–25, that is, to all vertices on its separator and on the four surrounding separators. (For subgrids at the edge of the graph there will be only two or three surrounding separators.) Obviously, $next(7) = (8)$. Vertex 8 is adjacent to the same vertices as 7, so the relative row indices of vertex 7 are just 1, 2, ..., 18. Similarly, vertex 8 has a trivial relative row-index set. In fact, it is clear that for all but the highest numbered vertex on any separator, the row-index set is trivial. So, only one row-index set is needed for each separator in the grid.

To bound the total number of pointers required, assume l levels of nested dissection have left 2^{2l} independent square grids of size $(2^{m-l} - 1)$. The first separator of each of these will be adjacent to at most four surrounding separator pieces, each of size 2^{m-l}, and is itself $2^{m-l} - 1$ long, so less than $5 \cdot 2^{m-l}$ pointers are needed for its vertices. The two second separators are each $2^{m-l-1} - 1$ long, and are adjacent to two separator pieces of size 2^{m-l} and two of size 2^{m-l-1}, so two sets of less than $7 \cdot 2^{m-l-1}$ pointers are needed for these two separators. The total count



$k_0 = n$
k_1
k_1
separating cross at level 0
k_1
k_1
k_1
k_1
k_2
k_2
separating crosses at level 1
$k_{m-2} = 3$
separating crosses at level $m - 2$
$4^{m-1} \approx \frac{n}{4}$
1×1 problems.

Fig. 7. Separator tree for $k \times k$ model problem.

of pointers, therefore, is bounded by

$$\sum_{l=0}^{m-2} 12 \cdot 2^{m-l} \cdot 2^{2l} \leq 12k^2.$$

This agrees with the results of Sherman [16] and George and Liu [10], who use a different storage scheme, but take advantage of the structure of the model problem in essentially the same manner.

A second question is how much stack space for temporaries is needed. Edges of T within one of the separator chains meet the requirement for not generating a new temporary. So only one temporary vector, of length $k_j \approx k2^{-j}$ is needed for

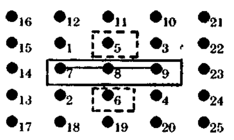


Fig. 8. A 3 × 3 subgrid.

each chain at level j. The total requirement is then

$$\sum_{l=0}^{m-1} k_l + k_{l+1} \leq 3k$$

words. Thus the stack is of trivial size compared with the storage for L, or even compared with that needed for the pointer arrays *locdiag*, *son*, and *brother*.

These results can be generalized to dissection orderings of arbitrary graphs, as proposed by Lipton et al. [13], and George and Liu [9].

Next, consider the possibility that nontrivial row-index sets are repeated. This occurs frequently in the model problem. In fact, only a constant number of different row-index sets occurs for vertices on separators at a given level. At level $m - l$, the sets have $O(2^l)$ elements, and no more than C of them are required, where C is independent of n and l. Thus $O(k)$ relative row indices are needed!

Of course, when sharing the relative row indices, a pointer is needed for every vertex showing where its relative row indices are stored. Thus, we require only

$$k^2 + O(k)$$

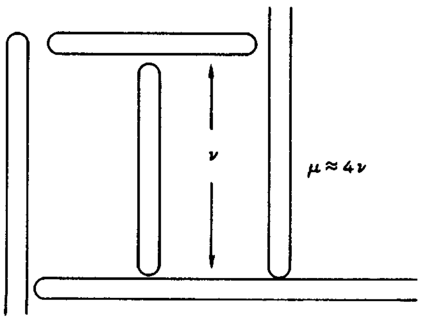
pointers for the model problem, a 12-fold savings compared with earlier results. (It should be noted that $12k^2$ pointers use far less space than the approximately $7\frac{3}{4}k^2\log_2 k$ nonzeros in L, so the overall storage savings are relatively minor.) This sort of situation occurs whenever a simple repeated pattern of elements is used to discretize a differential equation.

The method requires storage for three auxiliary arrays of size n. This is the same number required by the numeric factorization routine of the Yale sparse matrix package [3].

An experiment reported in Section 7 shows that for other sparse matrix problems the pointer storage requirement of this method is nearly the same as for absolute row-index methods.

6. TIMING FOR A VECTOR IMPLEMENTATION

Consider the ν vertices on a separator. Assume that each is adjacent to μ other higher numbered vertices and to the higher numbered vertices of the separator. The situation for the vertical separator of an interior subgrid is this:



There are $\mu + \nu - j + 1$ elements in the column for the jth vertex of the separator. For each of these there is a column from which the given column will be subtracted.

Except for the highest numbered vertex of the separator, all the vertices j are sons in T of a vertex with the same *col* set. Therefore, the *wherenext* pointers for these vertices are just $(1, 2, 3, \ldots, \mathit{numcol}(j))$. Therefore, whenever column j is used by the factorization algorithm, it is just multiplied by a scalar and added to another vector.

We assume a machine in which the cost of multiplying a vector of length V by a scalar is $S_M + P_M V$, and the cost of adding two vectors of length V is $S_A + P_A V$. S_M, P_M, S_A, and P_A are machine-dependent constants. S_M and S_A are called "start-up" costs. On current machines, $S_x \gg P_x$ for either operation $x = M, A$. The total of all costs for the jth vertex of the separator is, then,

$$\sum_{i=j+1}^{\nu+\mu} (S_A + S_M) + (P_A + P_M)(\nu + \mu + 1 - i)$$
$$= (\nu + \mu - j)(S_A + S_M) + \frac{(\nu + \mu - j)(\nu + \mu + 1 - j)}{2}(P_A + P_M).$$

The cost for all vertices $1 \le j \le \nu$ of the separator is approximately

$$(S_A + S_M)\sum_{j=1}^{\nu} (\nu + \mu - j) + (P_A + P_M)\sum_{j=1}^{\nu} \frac{(\mu + \nu - j)^2}{2}$$
$$\simeq \frac{(S_A + S_M)}{2}[(\mu + \nu)^2 - \mu^2] + \frac{(P_A + P_M)}{6}[(\mu + \nu)^3 - \mu^3].$$

We have ignored the complications due to the inapplicability of this analysis to the last vertex of a separator.

The separators at level 0 are a vertical separator of length $\nu = k = 2^m - 1$, and two of length $\nu = 2^{m-1} - 1$, with $\mu = k = 2^m - 1$.

The cost for these are (from the analysis above)

$$\frac{(S_A + S_M)}{2}\left[k^2 + \frac{5}{2}k^2\right] + \frac{(P_A + P_M)}{6}\left[k^3 + \frac{38}{8}k^3\right].$$

At deeper levels, there are three different types of regions to be separated: corners, sides, and interiors. At levels l, the regions being separated are of size $(2^{m-l} - 1) \equiv k_l$ square. There are 4^l such regions. Of these, 4 are corners, $4(2^l - 2)$ are sides, and $(2^l - 2)^2$ are interior. For corner regions there is a vertical separator with $\nu = k_l$ and $\mu = 2k_l + 1$, and two different horizontal separators with $\nu = k_{l+1}$ and $\mu = 5k_{l+1} + 4$ and $\mu = 3k_{l+1} + 2$, respectively. For sides, there is a vertical separator with $\nu = k_l$ and $\mu = 3k_l + 2$, and two horizontal separators with $\nu = k_{l+1}$ and $\mu = 5k_{l+1} + 4$. Finally, in interior regions, the vertical separator has $\nu = k_l$ and $\mu = 4(k_l + 1)$, while the horizontal separators have $\nu = k_{l+1}$ and $\mu = 6(k_{l+1} + 1)$ (see Figure 9). Summing all the contributions at level l yields, approximately, for $l \ge 1$,

$$C_l = \frac{(S_A + S_M)}{2}k^2\left[\frac{38}{2^{2l}} + 50\frac{(2^2 - 2)}{2^{2l}} + 62\frac{(2^l - 2)^2}{2^{2l+2}}\right]$$
$$+ \left(\frac{P_A + P_M}{6}\right)k^3\left[\frac{140}{2^{3l}} + 239\frac{(2^l - 2)}{2^{3l}} + \frac{371(2^l - 2)^2}{2^{3l+2}}\right].$$

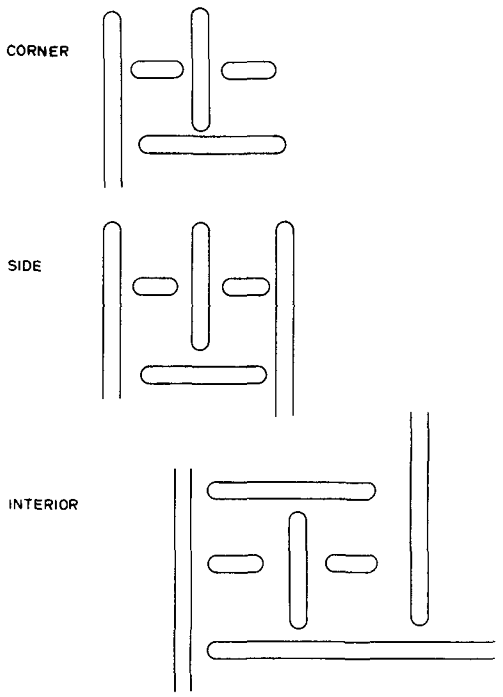


Fig. 9 Corner, side, and interior regions

Summing the various geometric series yields a total cost of

$$\sum_{l=0}^{m-1} C_l = k^3(P_A + P_M) \cdot \frac{829}{84} + k^2 \log_2 k(S_A + S_M) \cdot \frac{31}{4} + O(k^2).$$

George et al. [7] proposed an implementation of sparse Gaussian elimination that uses a block factorization and is most suitable for matrices arising from dissection of grids. The corresponding timing for this scheme is

$$k^3\left[(P_A + P_M)\frac{31}{12} + P_I\frac{51}{7}\right] + k^2 \log_2 k\left[(S_A + S_M)\frac{31}{4} + S_I \cdot 17\right] + O(k^2),$$

the time for an inner product being $S_I + P_I V$. Thus both schemes do the same number of operations, but the new scheme avoids the use of inner products, which on some machines are relatively slow, and also saves $17k^2 \log_2 k$ start-ups of inner products. We have neglected an additional $O(1)$ start-ups associated with the last vertex of every separator. As there are approximately $k^2/4$ separators in total, this does not change the leading terms.

George et al. [8] have shown that an incomplete nested dissection ordering, in which dissection stops one or two levels early and the remaining small grids are ordered row by row, yields an improved time estimate when using their implementation. The same observations are valid with the new scheme. When stopping with small grids, it may be better to treat the corresponding matrices as dense, thereby saving some storage for pointers and some vector start-ups.

7. SYMBOLIC FACTORIZATION

In order to use an absolute row-pointer storage scheme in a sparse matrix code, the nonzero structure of L has to be determined. This is done before numeric factorization, but after a suitable ordering of the rows and columns has been obtained. A "symbolic factorization" of A is carried out. Lists of the nonzeros in the columns of A are input; lists of the nonzeros in L are output. By merging the completed lists for columns $k \in row(j)$ into the list for column j (initialized with the nonzeros from A) the nonzeros in column j of L are obtained.

In part, sparse Gaussian elimination is a success because symbolic factorization can be done efficiently. The key to its speed is that if $j = next(k) = next(next(k'))$, then there is no need to merge the nonzero list of column k' into the list for column j: since $col(k') \subset col(k) \cup \{k\}$ already, all the necessary nonzeros are represented in the column k list. Thus, only lists of columns $k \in next^{-1}(j)$ (the sons of j in T) need be merged into the j list. The speed of this accelerated procedure is thus proportional to the size of L (since each column except the last is merged exactly once) and this is usually much smaller than the number of operations done during the numeric phase.

In the Yale sparse matrix package [3] and a method of George and Liu [10], symbolic factorization is even faster. The output is a compressed representation of the nonzero lists; redundancy in the lists is exploited, much as we have taken advantage of trivial relative row-index sets. These methods have running times proportional to the size of their compressed outputs, usually much smaller again then the size of L.

The obvious approach to symbolic factorization with relative row indices is to use an existing absolute row-index method, then process the output to get the relative row indices. But such a scheme would seem to have to spend a lot of time searching for elements in lists or arrays of indices, so it has not been pursued.

The essence of the problem is this. Lists of the nonzeros in columns (of L) $k \in next^{-1}(j)$ are merged with the nonzero-list of column j (of A) to form the nonzero-list of column j of L. In addition to this (ordered and linked) list, we also must find, for every element of every k-list, the relative position in the j-list of the nonzero in the same row.

We have implemented a code to do this with a small modification of the routine SSF of the Yale package. This code processes columns in the order $1, 2, \ldots, n$. In processing column j it creates a list of the nonzero elements of this column of L (the j-list). An n-long array q holds the j-list in a linked form:

$$q(i) = \begin{cases} 0, & \text{if } L_{ij} = 0 \\ i', & \text{if } L_{ij} \neq 0 \quad \text{and} \quad i' = \min\{\{n+1\} \cup \{r > i \mid L_{rj} \neq 0\}\}. \end{cases}$$

After the array q is created it is traversed, the nonzeros in column j are stored in an array holding the nonzero-lists of columns of L, and each entry is replaced by its relative position in the list. The code is this:

```
      $i := 0$;
      $mm := j$;
loop: $m := mm$;
      store $m$ into $j$-list;
      $mm := q(m)$;
      $i := i + 1$;
      $q(m) := i$;
      if $mm \leq n$ then go to loop;
```

Finally, every k-list (for $k \in next^{-1}(j)$) is traversed, and each (absolute) row index is replaced by the relative index in the element of q to which it points. This

Table I Comparison of Symbolic Factorization Codes

		Relative scheme		Absolute scheme	
Problem	*N*	Time (seconds)	*ptrs*	Time	*ptrs*
Graded-L	3937	0.242	26862	0.195	26733
Square	4225	0.256	27931	0 211	27834
H-domain	5185	0 295	30711	0 244	29920

process is slower than the unmodified procedure, since every k-list is scanned twice.

The arrays *son* and *brother* are also constructed during symbolic factorization. Initially all entries are zero. Whenever a j-list is completed, j is added to the list of sons of $next(j)$ by setting

$$brother(j) := son(next(j)) \quad \text{and} \quad son(next(j)) := j.$$

Of course $next(j)$ is known since it is the second element in the j-list. Whenever the set $next^{-1}(j)$ is needed, the arrays *son* and *brother* are used to generate it.

Trivial row-index sets can be recognized readily. They occur when either (case A) k is the only column in $next^{-1}(j)$ and every nonzero in column j of A is present in column k of L; or (case B) the number of nonzeros in column k of L below the diagonal is the same as the number of nonzeros in column $j = next(k)$ (including the diagonal). In case A the merging of lists and creation of the j-list can be avoided altogether by making the k-list the j-list.

A potential difficulty (observed by the referee) is that of putting the elements of A into the data structure. This can be done by replacing the absolute row indices of these elements with their relative row indices, exactly as is done for columns of L. This might involve the use of an additional array of pointers, one for every nonzero in A.

An experiment was done to compare the speed of this "relative" symbolic factorization to that of the "absolute" version in the Yale package. Three test problems due to George and Liu [11] were used; we omit a detailed description. Orderings were generated using the Yale minimum-degree subroutine. The results are shown in Table I. The times shown are each the average of two trails. The numbers of row indices generated are also shown. The experiments were done on the IBM 3081 at the Stanford Linear Accelerator Center (SLAC). The data show that the relative code takes between 20 and 25 percent more time than the absolute code. Because numeric factorization is so much more time-consuming, this difference is not important.

ACKNOWLEDGMENT

I am especially grateful to Mr. Philip Kuekes for his suggestions and comments.

REFERENCES

(Note. Reference [6] is not cited in the text)

1. BIRKHOFF, G , AND GEORGE, J.A Elimination by nested dissection In *Complexity of Sequential and Parallel Numerical Algorithms*, J. F. Traub (Ed), Academic, New York, 1973.
2. DUFF, I.S. Full matrix techniques in sparse Gaussian elimination. In Proc. Dundee Conf. on Numerical Analysis, Springer-Verlag, New York, 1981.

3. EISENSTAT, S.C., GURSKY, M.C., SCHULTZ, M H., AND SHERMAN, A.H. Yale sparse matrix package I. The symmetric codes. Res. Rep. 112, Yale Computer Science Dep. Yale Univ., New Haven, Conn.
4. EISENSTAT, S.C., SCHULTZ, M.H., AND SHERMAN, A.H. Software for sparse Gaussian elimination with limited core storage. In *Sparse Matrix Proceedings*, Iain S. Duff and G.W Stewart (Eds.), SIAM, 1978.
5. GEORGE, A. Nested dissection of a regular finite element mesh. *SIAM J. Numer Anal 10* (1973), 345-363
6. GEORGE, A Numerical experiments using dissection methods to solve *n* by *n* grid problems. *SIAM J Numer Anal. 14* (1977) 161-179.
7. GEORGE, A, POOLE, W.G., AND VOIGT, R G. Analysis of dissection algorithms for vector computers Math Dep. Tech. Rep. 13, College of William and Mary, Williamsburg, Va, 1976.
8. GEORGE, A., POOLE, W G, AND VOIGT, R G. Incomplete nested dissection for solving *n* by *n* grid problems. *SIAM J. Numer Anal. 15* (1978), 662-673.
9 GEORGE, A., AND LIU, J W H An automatic nested dissection algorithm for irregular finite element problems. *SIAM J Numer Anal 15* (1978), 1053-1069.
10 GEORGE, A., AND LIU, J.W H. An optimal algorithm for symbolic factorization of symmetric matrices. Res Rep. CS-78-11, Faculty of Mathematics, Univ. Waterloo, Waterloo, Ont, Canada
11. GEORGE, A, AND LIU, J.W. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, N J, 1981.
12 GUSTAVSON, F.G. Some basic techniques for solving sparse systems of linear equations. In *Sparse Matrices and Their Applications*, D.J. Rose and R.A. Willoughby (Eds), Plenum, New York, 1972.
13 LIPTON, R J, ROSE, D.J., AND TARJAN, R.E. Generalized nested dissection. *SIAM J Numer Anal. 16* (1979), 346-358.
14 PETERS, F.J. *Sparse Matrices and Substructures· A Novel Implementation of Finite Element Algorithms.* Mathematical Center Tracts MC 119, The Mathematical Center, 49, 2e Boerhaaverstratt, Amsterdam
15. ROSE, D J., TARJAN, R.E., AND LUEKER, G.S. Algorithm aspects of vertex elimination on graphs *SIAM J Comput 5* (1975), 266-283.
16 SHERMAN, A.H. *On the Efficient Solution of Sparse Systems of Linear and Nonlinear Equations* Ph D. Thesis, Yale Univ, New Haven, Conn., 1975.

Received February 1981, revised August 1981, accepted March 1982