NUMERICAL ANALYSIS PROJECT
MANUSCRIPT NA-80-06

SEPTEMBER 1980

A NEW IMPLEMENTATION OF SPARSE GAUSSIAN ELIMINATION

BY

ROBERT SCHREIBER

NUMERICAL ANALYSIS PROJECT
COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

A NEW IMPLEMENTATION OF SPARSE GAUSSIAN ELIMINATION

Robert Schreiber *

Abstract.

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.

* Department of Computer Science, Stanford University, Stanford, California 94305.

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

1. Introduction.

Let A be an nxn , irreducible, symmetric positive definite matrix. The system

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

can be solved by a Cholesky factorization

$$A = LDL^T \tag{1.2}$$

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

$$L z = b , \quad L^T x = D^{-1} z . \tag{1.3}$$

When only a small fraction of the elements of A are nonzero, A is said to be sparse. Such problems arise, for instance, in simulating electrical networks and in numerical methods for partial differential equations. We use the term sparse Gaussian elimination to mean all methods for computing a triangular factorization and back substitution in which maximum advantage is taken of zeros in the matrix and the factors. Pointers of some kind are used to keep track of the nonzeros. Elements which are zero initially and do not fill-in are neither stored nor used as operands. Band and profile methods differ in that they take advantage of some, but not all, of these zeros.

Sparse elimination methods incur the costs of storing and manipulating pointers. Given a sufficiently sparse factorization, the time and storage saved by not manipulating and storing zeros far exceeds these costs. Sherman [13] and George [4] give experimental evidence. When a sparse matrix is factored, "fill-in" occurs: the triangular factors contain nonzeros in positions where A has zeros. Ordinarily, the rows and columns of A are

first permuted so that the fill-in is made small. We shall not be concerned with the problem of finding such permutations.

Sparse Cholesky factorization is ordinarily implemented as a two-step process. First, the zero-structure of the factor is computed. The procedure, called symbolic factorization, predicts which elements of A will fill in. It need be done only once if several systems with the same zero structure are to be solved. 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 A , including zeros which later fill in, are stored in a one-dimensional array, which we call x . A separate array, row, records the row to which the corresponding element of array x belongs: if a_{64} is stored at $\underline{x}(12)$, then $\underline{row}(12) = 6$. Columns of A occupy successive contiguous blocks of x , and are sorted by row. Pointers to the first elements of columns are stored in an array, colbegin. Only the lower triangle of A need be stored. Figure 1 gives an example of this scheme.

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

$\underline{x}$:	a_{11}	31	41	22	32	33	43	44
row :	1	3	4	2	3	3	4	4
colbegin :	1	4	6	8				

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

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 L to a column $j > k$ such that $\ell_{jk} \neq 0$. Then, instead of the row to which a nonzero-element belongs (its absolute row-index), a pointer to the location in x of the nonzero in the same row (say row i , $i > k$) and the associated column is stored: the pointer for ℓ_{ik} points to the storage for ℓ_{ij}. . 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 [8].

The principle advantage of a relative row-index scheme is the efficiency with which a column (k, above) can be added to or subtracted from its associated column (j,above). In fact, the FORTRAN loop for such an operation is just

```
      Do 1  I = 1,NK
1           X(J+PTR(K+I))=X(J+PTR(K+I))+X(K+I) .
```

When an absolute row-index scheme is employed, the code for adding or subtracting two columns is more complicated. Gustavson [10] and Eisenstat, et.al. [2] avoid a complicated inner loop by unpacking one of the columns into a temporary array of size n , a scheme which suffers from a significant drawback: the code accesses this large temporary in a random way, which degrades performance on a machine with a cache memory.

No one has found a very good way to implement a code using absolute row-indices on a vector computer like the CDC CYBER-203 or CRAY-1 .

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 n x n grid problems n^2 pointers are required. Previous methods have required at least 12 n^2 row-indices ([8],[13]). Section 6 gives a detailed analysis of storage requirements for these problems. An analysis of implementation on vector computers is given in Section 7.

A scheme for finite element systems of Eisenstat, Schultz, and Sherman [2a], which is reminiscent of frontal elimination methods, has several of the characteristics and advantages of the proposed scheme.

2. The method.

This section first reviews the Cholesky (LDL^T) factorization algorithm, then shows how a relative row-index storage scheme can be used to carry it out.

2.1 Cholesky factorization.

Algorithm 1 is a column-oriented LDL^T factorization. The inner loop subtracts a multiple of every column $k < j$ from a particular "pivot" column, column j.

```
1    real array d[1:n], a,ℓ[1:n,1:n],
2    real multpyr;
3    integer n,i,j,k;

4    for     j: = 1 to n do begin
5        for k: = 1 to j-1 do begin
6            multpyr: = ℓ_jk * d_k;
7            for   i: = j to n
8                a_ij: = a_ij - multpyr * a_ik
9            end (for k);
```

```
10      d_j: = a_jj;
11      for i: = j+1 to n
12          ℓ_ij: = a_ij/d_j
13      end (for j).
```

Algorithm 1. Column oriented LDL^T factorization.

Note that only the elements in the lower triangle of A would be stored, and could be overwritten by L and D .

Algorithm 2 is a column oriented version of the "forward solve" Lz=b ; Algorithm 3 is a row-oriented version of the "backward solve" $L^T x = D^{-1} z$.

```
/*  forward solve   Lz=b  */
/*  L is unit lower triangular  */

1   real array  ℓ[1:n,1:n],z,b[1:n];
2   integer  i,j;
3   begin
4      for j:=1 to n do begin
/*  b(j)  now  contains  z(j)  */
5         z(j):= b(j);
6         for  i:= j+1 to n do
7            b(i):= b(i) - z(j)*ℓ(i,j)
8         end
9     end (forward solve).
```

Algorithm 2. Column oriented forward solve.

```
/*  backward solve L^T x = D^-1 z  */
1   real array  ℓ[1:n,1:n], z,x,d[1:n];
2   integer  i,j;
3   begin
4      for i:=n to 1 step -1 do begin
5         z(i):= z(i) / d(i);
6         for  j:= i+1 to n do
7            z(i):= z(i) - ℓ(j,i)* z(j)
8         end
9      end (backward solve).
```

Algorithm 3. Row oriented backward solve.

2.2 The new scheme.

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

Define

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

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

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

and $$N(L) \quad \{(j,\underline{\text{next}}(j)) \in E \mid 1 \leq j \leq n-1\}\ .$$

The edges $(j, \underline{\text{next}}(j))$ 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 $\underline{\text{next}}(j)$ 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 $k = v_0, v_1, \ldots, v_\ell = j$ of vertices with v_{i-1} adjacent to v_i, $1 \leq i \leq \ell$. It is monotone if $v_i > v_{i-1}$, $1 \leq i \leq \ell$. 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 closed paths: 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 [12]: If there is a j-k path in G(L) through vertices smaller than both j and k, then $(j,k) \in E$.

Corollary: If $j = \underline{\text{next}}(k)$ and $\ell_{ik} \neq 0$, then $\ell_{ij} \neq 0$.

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 $\underline{\text{next}}(k)$:

$$\underline{\text{col}}(k) \subset \underline{\text{col}}(\underline{\text{next}}(k)) .$$

Proposition 2: For each $1 \leq k < n$, $\underline{\text{col}}(k)$ is not empty.

Proof: $G(L)$ is strongly connected. Choose any vertex $\ell > k$ 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', $(k,\ell') \in E$, so $\ell' \in \underline{\text{col}}(k)$.

QED

The definition of next makes sense, then: the minima of nonempty sets are taken.

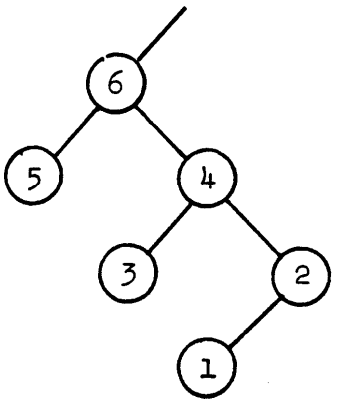
Using relative row-indices it is easy to add a multiple of a column to its next column. The inner loop of our scheme does this. The inner loop of Algorithm 1, on the other hand, subtracts from the pivot column (the j'th) a multiple of each column k such that $k \in \underline{\text{row}}(j)$. The key idea is to accumulate a sum of multiples of columns $k \in \underline{\text{row}}(j)$.

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 $T_{\underline{\text{row}}(j)}$ is an ordered tree with root j. Thus, for every $k \in \underline{\text{row}}(j)$ there is a unique k-j path in T that goes through other vertices in $\underline{\text{row}}(j)$. A sum of appropriate multiples of columns in $\underline{\text{row}}(j)$ is accumulated by a depth-first traversal of $T_{\underline{\text{row}}(j)}$.

Here is an example. Suppose

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

Then $T =$



Since $\underline{row}(6) = \{1,2,3,4,5\}$, the algorithm will subtract a multiple of each of those columns from column 6. It can subtract the multiple of column 5 directly: since $6=\underline{next}(5)$, it has the relative row-indices needed for that operation. 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 $T_{\underline{row}(6)}$ was an ordered tree with root 6.

Now the proofs.

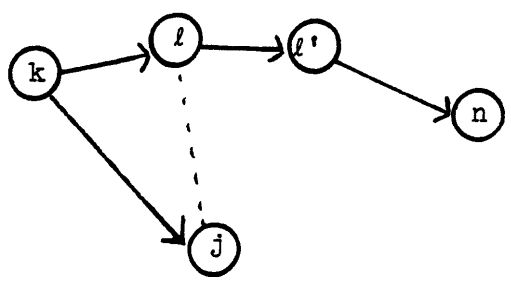
Proposition 3: T is an ordered tree with root n .

Proof: Construct T . Start with the single vertex n . Add the preceding vertices n-1, n-2,...,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 .

QED

Proposition 4: For every $1 \leq j \leq n$, $T_{row(j)}$ is an ordered tree with root j .

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



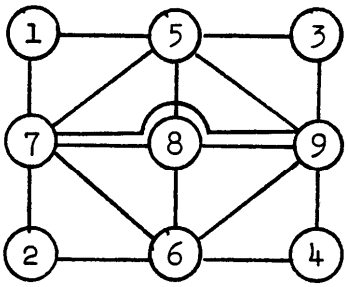
Vertices increase going left to right.] We claim that j is actually on the k-n path, showing that a unique monotone k-j path in $T_{row(j)}$ exists, which proves the proposition. Suppose not. Then an edge (ℓ,ℓ') on the path with $\ell < j < \ell'$ exists. But $\ell' = next(\ell)$, so $\ell' \leq j$,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

QED

Here is a more realistic and interesting example: a 3x3 finite difference grid.

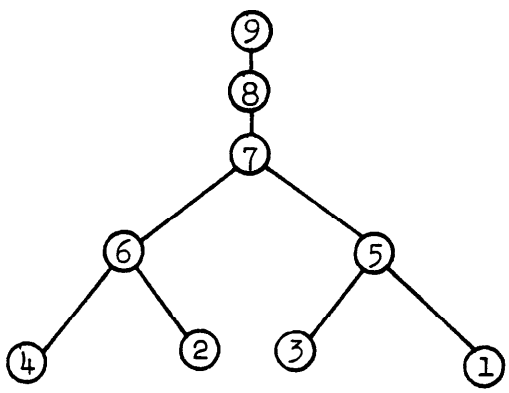
Example 1.

G(L) =

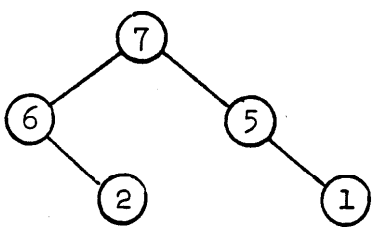


k	: 1	2	3	4	5	6	7	8
next(k)	: 5	6	5	6	7	7	8	9

T :



$T_{row(7)}$:



For the backsolving scheme, we need one last fact.

Proposition 5: For each $1 \leq j < n$, 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)).

QED.

3. An implementation of the new scheme.

In this section we present a detailed implementation of the new sparse LDL^T factorization method. The details of the data structure are covered in Section 3.1. The algorithm is described in Sections 3.2 and 3.3. A pseudo-algol implementation is given in an appendix.

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.

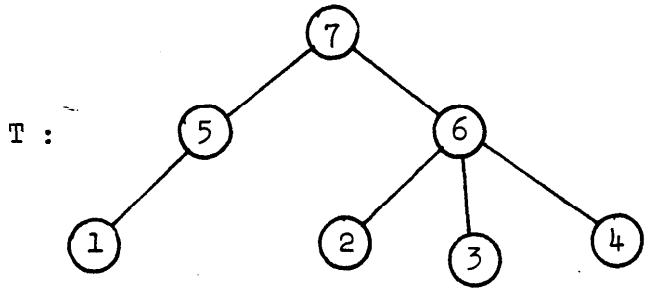
For each $1 \leq j \leq n$, locdiag(j)+1 is the location in aa of a_{jj} and d_{jj}. Thus, the i'th nonzero of column j is stored in aa(locdiag(j) + i). Also, locdiag(n+1) is the location of the last element + 1 of aa. An integer array wherenext contains the relative row-indices. Suppose the nonzero 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 i'th row of L. Somewhere in the storage for column j, say at the ℓ'th position, is a location for the element ℓ_{ij}.. (There is an i-j path (of length two) through k in G(L)). In other words, ℓ_{ij} is stored at aa(locdiag(j)+ℓ).

Then

$$\underline{\text{wherenext}}(m) \equiv \ell \ .$$

Note that wherenext needn't be defined for elements on the diagonal; they are never subtracted from elements in other columns.

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).



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

Figure 3. Storage of T .

Note that the next links are not stored. Figure 4 illustrates this storage scheme applied to example 1 .

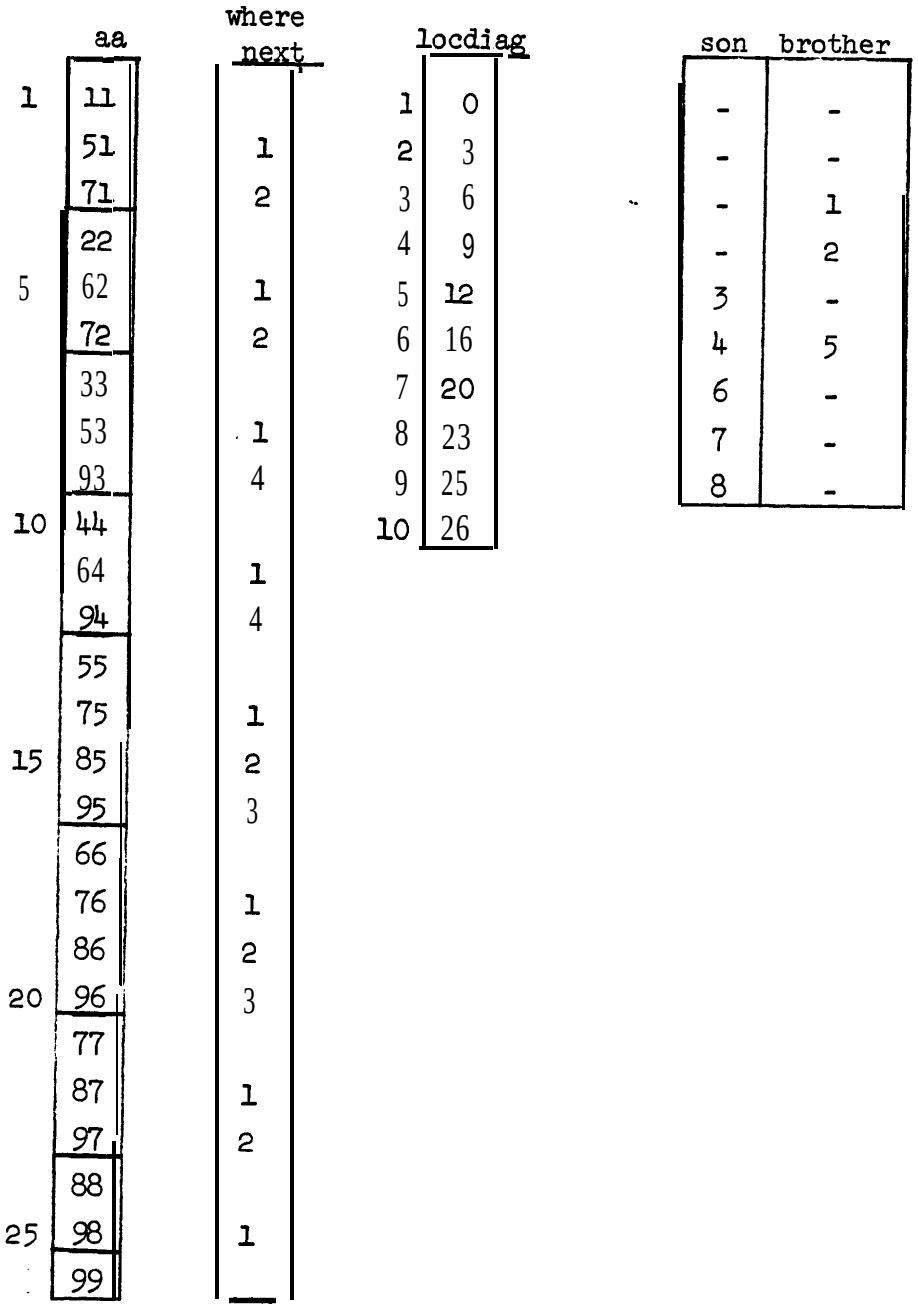


Figure 4. Storage scheme applied to Example 1.

The numbers in aa are the indices (i,j) of the element ℓ_{ij} stored at that position.

3.2 The algorithm.

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

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

and let $\underline{first}(j,j) = 1$.

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

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

Also, define

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

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

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

In some implementations of sparse elimination (the Yale package, for example), a data structure for representing $\underline{row}(j)$ is maintained during numeric factorization; the columns belonging to $\underline{row}(j)$ are explicitly available from that data structure [13] . In this implementation, membership in $\underline{row}(j)$ is determined as part of the tree search process.

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

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

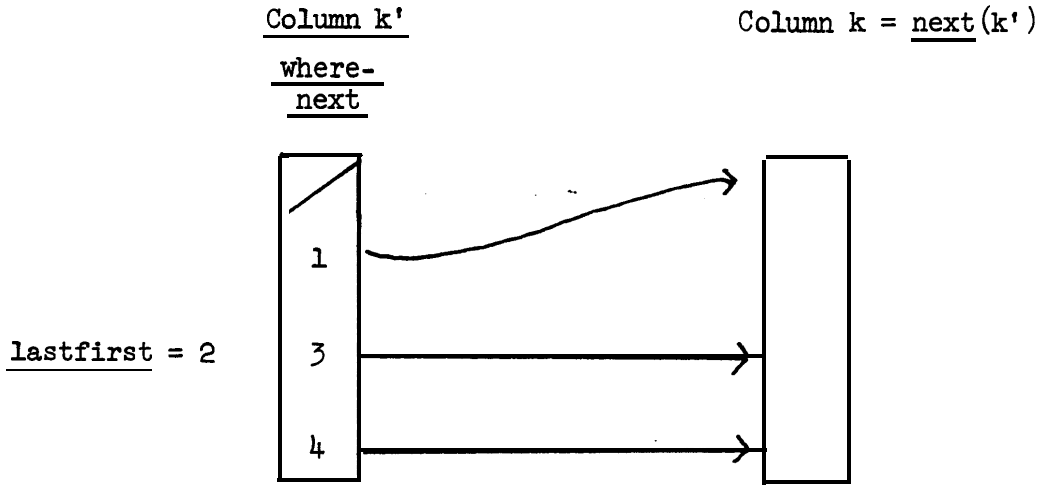
$$1 \leq p \leq \underline{numcol}(k') . \tag{3.1b}$$

If such a p exists, then we set $\underline{first}(k',j) = p$ and continue the search process. If not, $k' \notin \underline{row}(j)$, 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.

Let $\underline{lastfirst}(k')$ be the last value of p to have satisfied (3.1); initially $\underline{lastfirst}$ is given the value 1. Suppose column k' is searched by a call from its father in T, column k. Since the sequence of pivot columns j increases, and the elements of columns are sorted by row, $\underline{first}(k,j)$ will be larger on this call then on the last. Two possibilities occur. Maybe

$$\underline{wherenext}(\underline{locdiag}(k') + \underline{lastfirst}(k') + 1) > \underline{first}(k,j) .$$

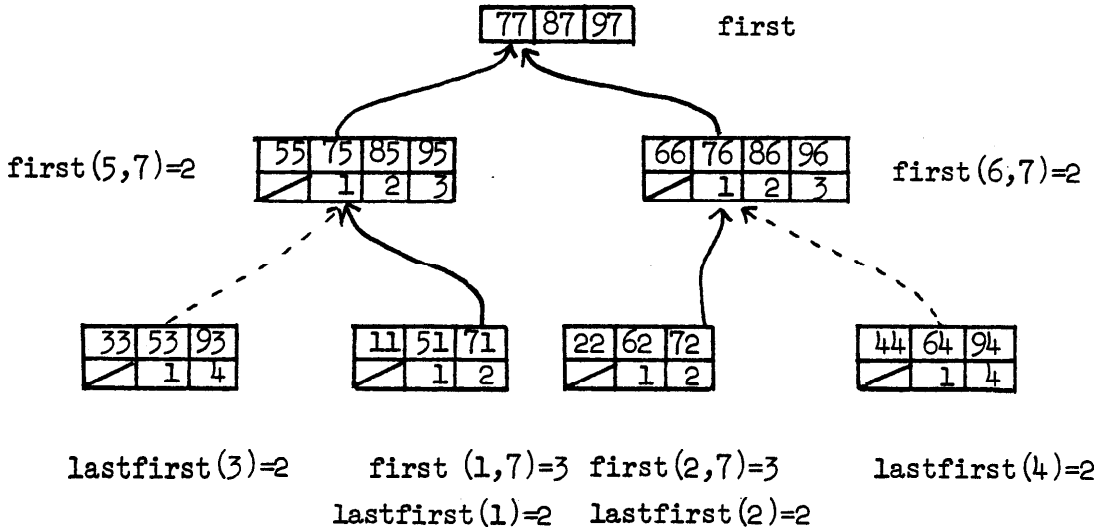


$$\underline{\text{first}}(k,j) = 2 \Rightarrow k' \notin \underline{\text{row}}(j$$

$$\underline{\text{first}}(k,j) = 3 \Rightarrow k' \in \underline{\text{row}}(j) .$$

Figure 5. Determining membership of k' in $\underline{\text{row}}(j)$.

In that case, $k' \notin \underline{\text{row}}(j)$. If $\underline{\text{wherenext}}(\underline{\text{locdiag}}(k') + \underline{\text{lastfirst}} + 1 = \underline{\text{first}}(k,j)$, then $k' \in \underline{\text{row}}(j)$. (See figure 5). The third situation (c) never occurs. Figure 6 illustrates the depth first search of $T_{\underline{\text{row}}(7)}$ for Example 1 . .



——————→ A tree link traversed and (4.1) satisfied.

- - - - - - - 3 A tree link traversed, but (4.1) not satisfied.

Figure 6. Data structures used during depth first search.

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, maxstack, that will be required.

3.3 Backsolving.

It is not evident that the relative row-index scheme is even marginally suitable for backsolving

$$Lz = b \,, \tag{3.3a}$$

or

$$L^T x = D^{-1} z \,. \tag{3.3b}$$

It appears at first that, to access b, z, and x (which share the same n storage locations in our code) absolute row-indices are required. These could be precomputed (by the symbolic factorization routine) and stored, or 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 (3.3a) and (3.3b) 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 (3.3a) 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. In algorithm 2, lines 6-7, the j'th column of L multiplied by $z(j)$ is subtracted from the corre-

sponding elements of b ; in the new method, it is added to a temporary using wherenext to determine the positions involved.

When a column, k , is first searched, a temporary of length num(k) is allocated and initialized to zero. The sons of k , all $k' \in$ row(k) , are then searched. At this point, the temporary contains the num(k)-vector

$$t_i = \sum_{k' \in \text{row}(k)} z(k')\, \ell_{ik'} , \qquad i \in \text{col}(k) .$$

Now, $z(k) = b(k) - t_k$. Next, $z(k)\, \ell_{ik}$ is added to the temporary. Finally, the temporary is added to the appropriate places in the temporary of next(k) , using the relative row-indices of column k .

A pseudo-algol implementation appears in the appendix.

Backsolving (3.3b) 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

$$x_k = z_k/d_{kk} - \sum x_i\, \ell_{ik} , \tag{3.4}$$

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

The solution vector $\underline{x}$ 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 required are scattered randomly. Let $j=\underline{next}(k)$. The algorithm passes a temporary vector containing $\underline{x}^{(j)}$ when searching vertex k. The elements of $\underline{x}^{(k)}$ reside in positions pointed to by column k's relative row-indices. The inner loop extracts these elements, performs the dot product in (3.4), and creates the temporary vector holding $X^{(k)}$ which will be passed to the sons of vertex k in T.

Figure 7 illustrates the backsolve for Example 1. T is shown. To the right of vertex j is $x^{(j)}$; to the left, $\underline{col}(j)$ is shown in parens, the relative row-indices in square brackets. When vertex 3 is searched, x_5 and x_9 are extracted from $\underline{x}^{(5)}$ using the relative row-indices, [1,4], of columns 3. Then $x_3 = z_3/d_{33} - (\ell_{53}\, x_5 + \ell_{93}\, x_9)$.

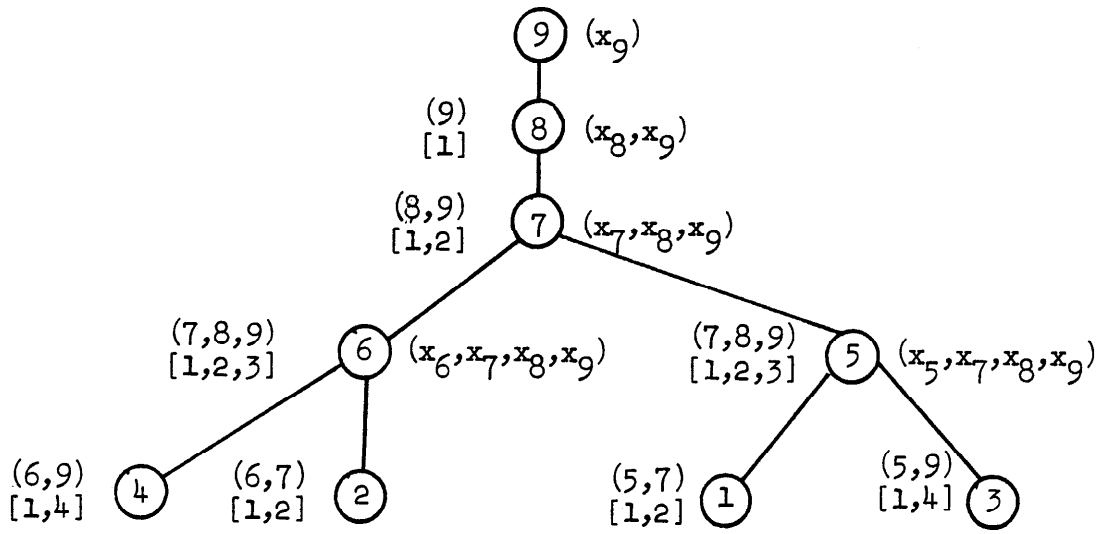


Figure 7. Tree search for $L^T x = D^{-1} z$.

4. Nonsymmetric matrices.

Relative row-index schemes are not useful for solving $Ax = b$ with arbitrary nonsymmetric A. But if A has a symmetric zero structure and can be factored as LU (without partial pivoting) so that the structure of U is the transpose of that of L, then a method based on the techniques of the earlier sections is possible. Such a linear system might arise, for example, in the finite-element or finite-difference solution of an elliptic boundary-value problem with relatively small nonsymmetric terms. Low Reynolds number flow problems are an example.

A suitable method is based on a version of the LU factorization in Algorithm 4. This algorithm works with columns of L and rows of U in the inner loop. Like the LDL^T factorization, Algorithm 1, multiples of columns are subtracted from a "pivot" column of L, multiples of rows are subtracted from a "pivot" row of U.

Details of the data-structure and algorithm are so like those used in the LDL^T method that we omit them. Of course, only one set of pointers suffices to describe the structure of both L and U.

```
for  j: =  1 to n do
    for i: = 1 to j-1 do
        subtract ℓ_ji* row i of U
        (from column j to n) from
        row j of U;
    if j < n then begin
          for k: = 1 to j-1 do
              subtract U_kj* column k of L
              (from row j+1 to n) from
              column j of L;
        multiply column j of L by 1/U_jj
        end (if j < n);
```

Algorithm 4. Doolittle LU factorization of a dense, nonsymmetric matrix.

5. An improvement to the method.

In this section the methods of Section 3 are re-examined and improved in three ways. The improvements stem from the observation that the relative row pointers for a column, say column k , are often just the sequence 1,2,...,num(k)-1 . There is, then, no reason to store these pointers. There is no reason to create a new temporary when searching a vertex with these relative row pointers. Finally, there is no reason to use these relative row-indices to access storage in the inner loops of the factorization and back substitution procedures. We say that the row-indices of a column are trivial in the situation here described.

Finite element problems, especially when ordered by nested dissection techniques (see Section 6) lead to matrices in which many columns have trivial row-indices.

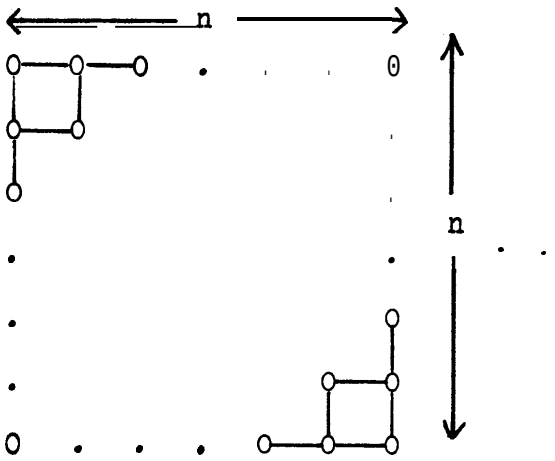
George and Liu have proposed an alternate storage scheme, based on using the zero structure of one column to "represent" that of a set of columns [8] . This scheme has some similarity to ours. George has used other storage schemes that take advantage of columns with like structure. Sherman's codes, which use absolute row-indices, also recognize and exploit redundancy in the structure of adjacent columns [13].

Non-trivial relative row-index sets can be redundant: two different columns may have exactly the same set of relative row-indices, 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 6 is one such situation.

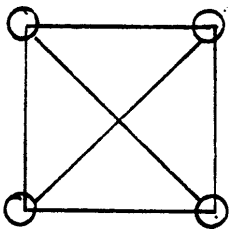
6. A model problem.

For a symmetric NxN matrix A we define the undirected graph $G(A) = V,E(A))$, where $V = \{1,2,\ldots,N\}$, and $E(A) = \{(i,j) | a_{ij} \neq 0\}$. The 5-point model problem is a symmetric positive definite problem with $N = n^2$ and an nxn "grid-graph".

G(A) =



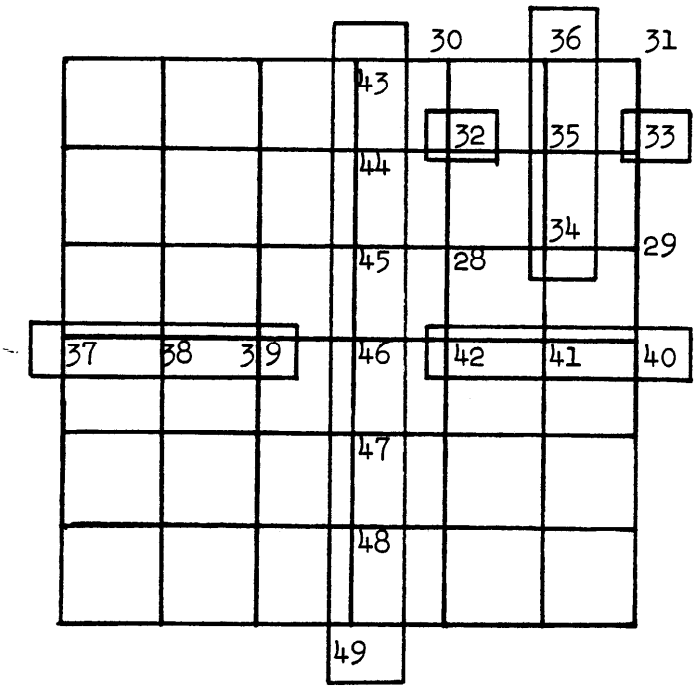
Finite-element methods lead to a 9-point model problem which differs from the 5-point problem in that the basic cell of G(A) is



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 n (5-point) or n+1 (9-point) . To best utilize sparse matrix techniques, the vertices of G(A) are ordered by nested dissection [1,3]. The vertices of a separating cross are numbered last. The remaining ver-

tices constitute 4 independent grid-graphs of size n/2 by n/2 . These are numbered by (recursively applying) nested dissection. For example, with n=7 , G(A) is

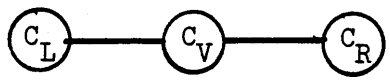


(The numbering of the other 3x3 subproblems is obvious.) The first level's separating cross consists of a vertical separator C_V (nodes 43-49) , a left-horizontal separator C_L (nodes 37-39), and a right-horizontal separator C_R (nodes 40-42). The whole cross,

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

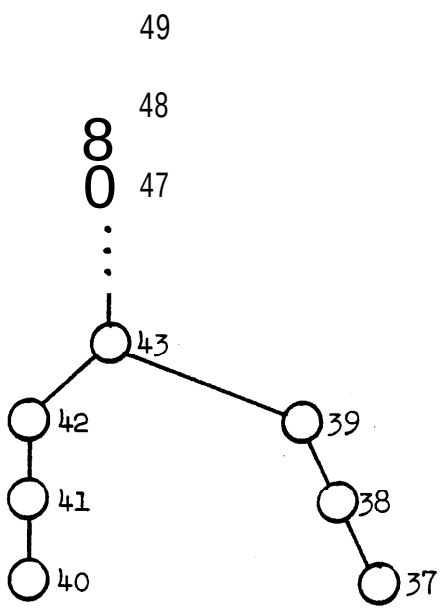
The structure of the Cholesky factor L of A ,and hence of its graph G , can be surmised from Tarjan's result [12]: vertices i and j are adjacent in G provided an i-j path consisting of smaller vertices

exists in $G(A)$. Suppose we have a 9-point problem. Then G_C , the subgraph of G describing the last 2n-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 the 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 7 vertices (those in C_V) and two sub-chains, 3 long (one for C_L and one for C_R). Denote by $\circ\overset{r}{=\!=}\circ$ a chain of length r,

let $n = 2^m - 1$, and $n_j \equiv 2^{m-j} - 1$. The tree for nested dissection of an nxn 9-point problem is shown in Figure 8.

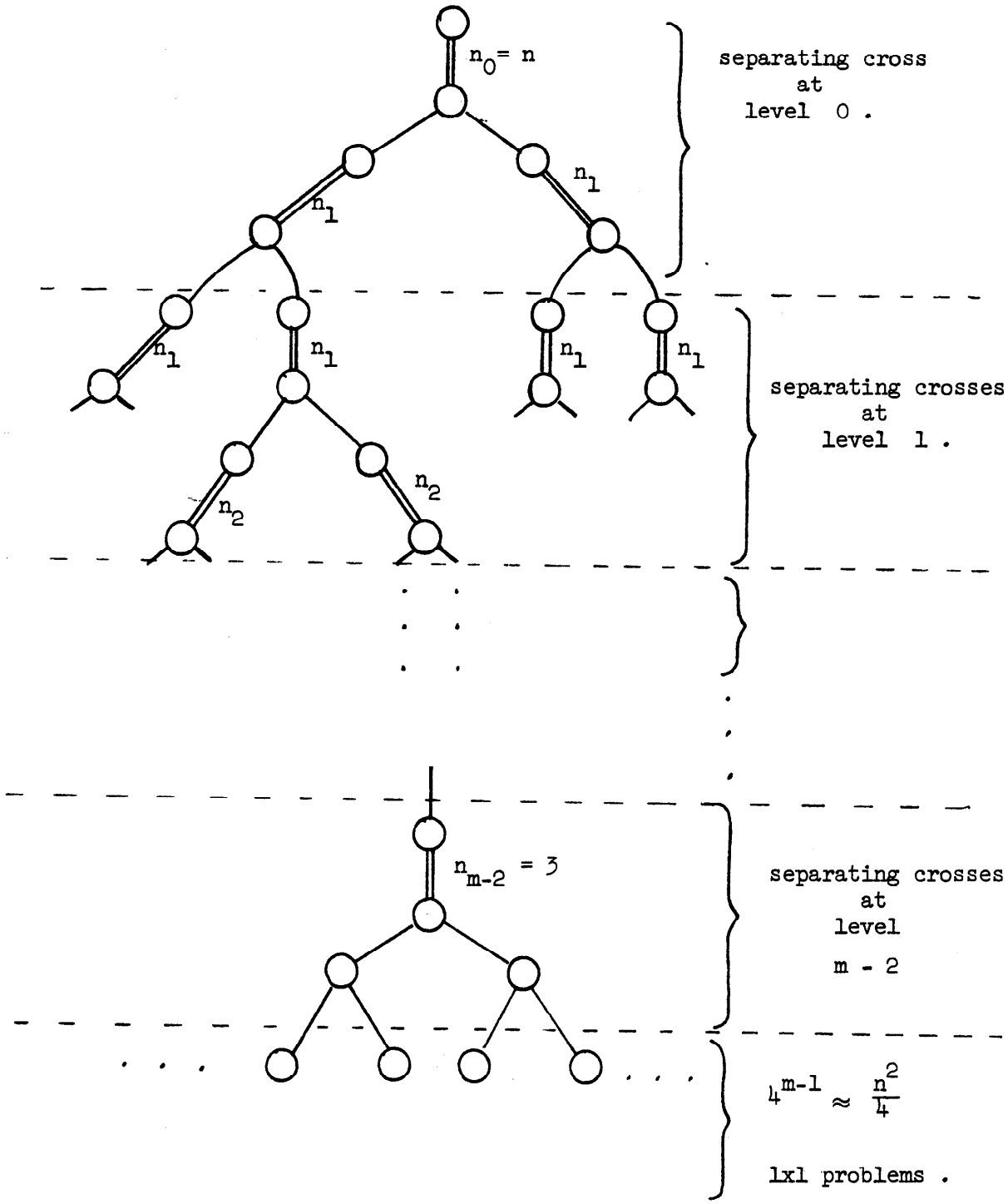


Figure 8. 9-point model problem.

How well do the optimizations of Section 5 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 ?

At level ℓ in the dissection, square subgraphs of size $n_\ell = 2^{m-\ell} - 1$ remain; Figure 9 illustrates the case $\ell = m - 2$.

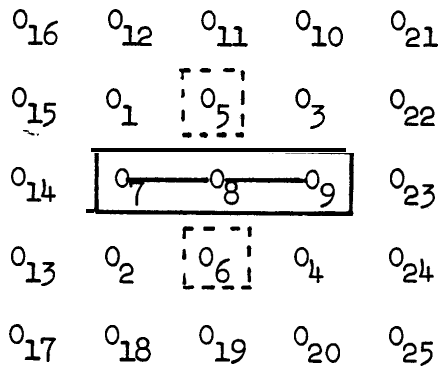


Figure 9. A 3x3 subgrid.

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 is needed for each separator in the grid.

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

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

This agrees with the results of Sherman [13] and George and Liu [8], who use a different storage scheme, but take advantage of the structure of the model problem in essentially the same manner. Like the method of Eisenstat, Schultz, and Sherman [2a] the inner loop doesn't refer to pointers in this situation.

With the 5-point model problem, the analysis is not valid. For example, only the leftmost vertex of the three on the horizontal separator, vertex 9 in Figure 9, is adjacent to vertex 23 on the right boundary separator. No paths from 7 or 8 to 23 exist without going through 9, 22 or 24; hence, $(7,23) \notin E(L)$ and $(8,23) \notin E(L)$. Thus, if nested dissection is used on a 5-point problem, then fewer columns have trivial pointers. But it is already known that nested dissection is not optimal, from the viewpoint of work and fill-in, for such problems. Rather, a diagonal nested dissection should be used [1]. In this ordering, the $n^2/2$ vertices in positions

corresponding to red squares on a checkerboard are eliminated first. The remaining graph has a g-point structure, but is rotated by 45°. It is dissected using the usual nested dissection ordering (the separators are now at 45° angles to the vertical). Thus, for the second half of the vertices, the observations above apply.

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 $n_j \approx n\, 2^{-j}$ is needed for each chain at level j . Thetotalrequirement is then

$$\sum_{k=0}^{m-1} n_k + n_{k+1} \leq 3n$$

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, and brother.

It is evident that these results can be generalized to dissection orderings of arbitrary graphs, as proposed by Lipton, Rose, and Tarjan [11], and George and Liu [7] .

Next, consider the possibility that non-trivial 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-j , the sets have $O(2^j)$ elements, and no more than C of them are required, where C is independent of n and j . Thus, $O(n)$ 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

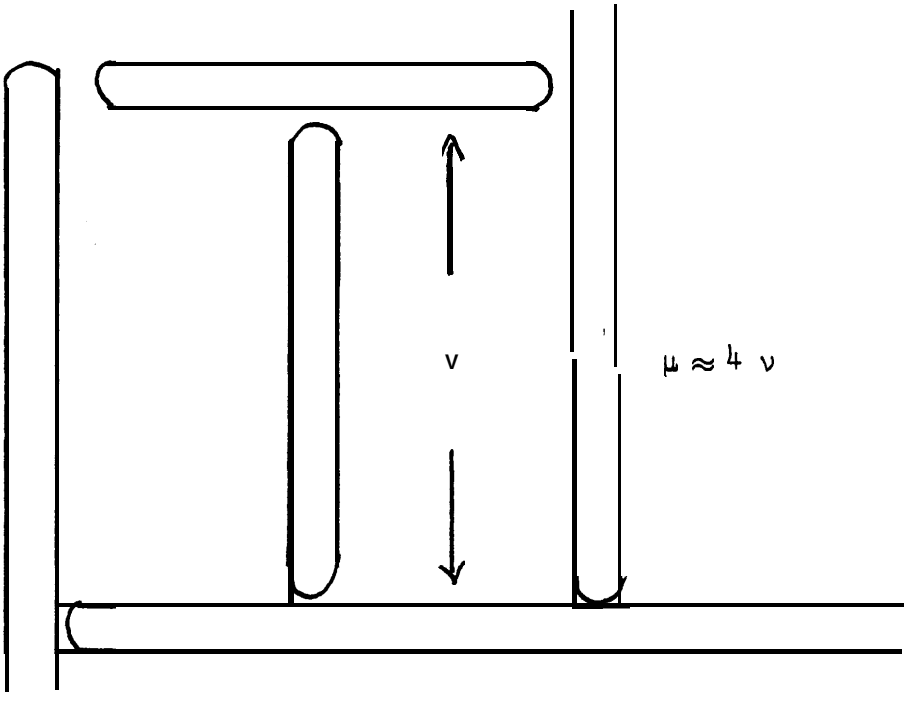
$$n^2 + O(n)$$

pointers for the model problem, a 12-fold savings compared with earlier results. (It should be noted that' $12n^2$ pointers use far less space than the approximately $7\frac{3}{4}\, n^2 \log_2 n$ 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.

7. Timing for a vector implementation.

I. 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,

Eg:



There are $\mu + \nu - j + 1$ elements in the column for the j'th 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,...,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 L by a scalar is $S_M + P_M L$, and the cost of adding two vectors of length L is $S_A + P_A L$. The total of all costs, then, for the j'th vertex of the separator is

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

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

The cost for all vertices $1 \leq j \leq \nu$ 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.

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

The cost for these are approximately

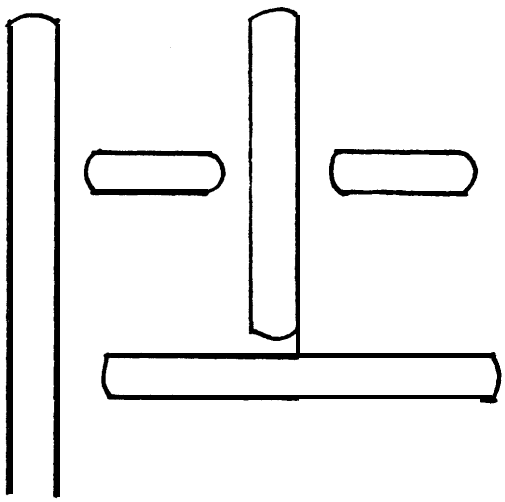
$$\frac{(S_A + S_M)}{2} [n^2 + \frac{5}{2} n^2] + \frac{(P_A + P_M)}{6} [n^3 + \frac{38}{8} n^3]$$

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 $(2^{m-\ell} - 1) \equiv n_a$ square. There are 4^ℓ such regions. Of these, 4 are corners, $4(2^\ell - 2)$ are sides, and $(2^\ell - 2)^2$ are interior. For corner regions there is a vertical separator with $\nu = n_a$ and $\mu = 2\, n_\ell + 1$, and two different horizontal separators with $\nu = n_{\ell+1}$ and $\mu = 5\, n_{\ell+1} + 4$ and $\mu = 3\, n_{\ell+1} + 2$ respectively. For sides, there is a vertical separator with $\nu = n_a$ and $\mu = 3\, n_\ell + 2$, and two horizontal separators with $\nu = n_{\ell+1}$ and $\mu = 5\, n_{\ell+1} + 4$. Finally, in interior regions, the vertical separator has $\nu = n_\ell$ and $\mu = 4(n_\ell + 1)$ while the horizontal separators have $\nu = n_{\ell+1}$ and $\mu = 6(n_{\ell+1} - 1)$. (See Figure 10). Summing all the contributions at level ℓ yields, approximately, for $\ell \geq 1$,

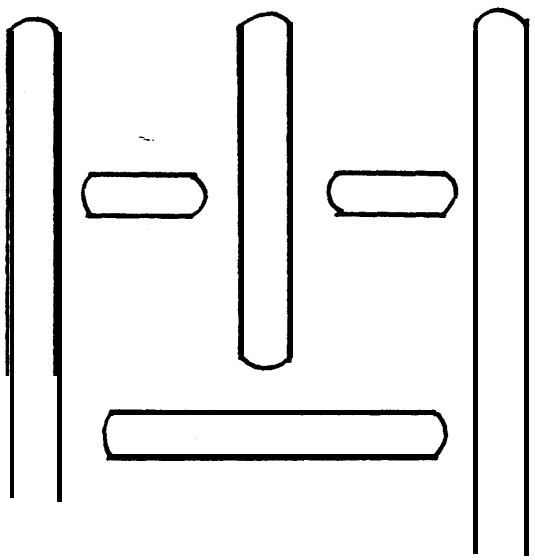
$$C_\ell = \frac{(S_A + S_M)}{2} n^2 \left[\frac{38}{2^{2\ell}} + 50 \frac{(2^2 - 2)}{2^{2\ell}} + 62 \frac{(2^\ell - 2)^2}{2^{2\ell+2}} \right]$$

$$+ \left(\frac{P_A + P_M}{6}\right) n^3 \left[\frac{140}{2^{3\ell}} + 239 \frac{(2^\ell - 2)}{2^{3\ell}} + \frac{371(2^\ell - 2)^2}{2^{3\ell+2}} \right] .$$

corner:



side:



interior:

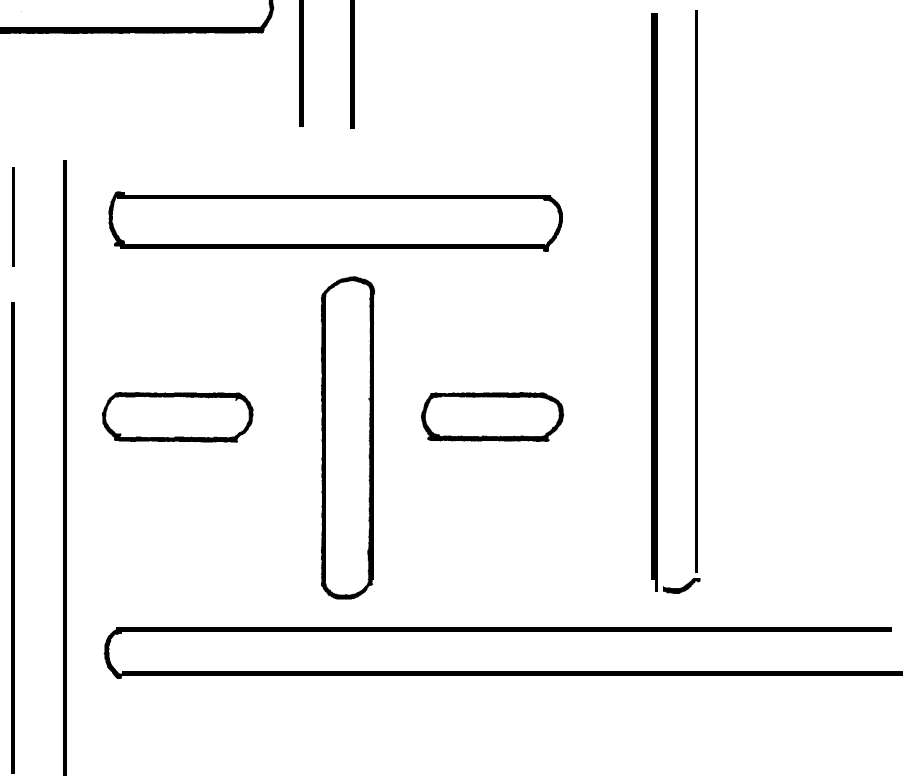


Figure 10: Corner, side, and interior regions.

Summing the various geometric series yields a total cost of

$$\sum_{\ell=0}^{m-1} C_\ell = n^3 \ (P_A + P_M) \cdot \frac{829}{84}$$

$$+n^2 \log 2\ n\ (S_A + S_M) \cdot \frac{31}{4} + o(n^2) \ .$$

George, Poole, and Voigt [5] proposed an implementation of sparse Gaussian elimination which uses a block factorization and is most suitable for matrices arising from dissection of grids. The corresponding timing for this scheme is

$$n^3[\ (P_A + P_M)\ \frac{31}{12} + P_I\ \frac{51}{7}\ ]$$

$$+\ n^2 \log_2\ n((S_A + S_M)\ \frac{31}{4} + S_I\ .17\ ) \quad ,$$

the time for an inner product being $S_I + P_I L$. 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 $17\ n^2 \log_2 n$ startups of inner products. We have neglected an additional $o(1)$ startups associated with the last vertex of every separator. As there are approximately $\frac{n^2}{4}$ separators in total, this does not change the leading terms.

George, Poole, and Voigt [6] 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 startups.

Appendix I.

Programs.

The algol programs below implement the methods and data structures described in Sections 3.1, 3.2, and 3.3. The program assumes that the data structure, the arrays aa, bb, locdiag, wherenext, son, and brother have been previously computed.

Two main programs are given. One computes the LDL^T factorization using a procedure dfs to conduct the depth first search. The second solves for x given the factorization.

```
/* Sparse numeric factorization */
begin   /* main program */
real array aa(1:numa), stack(1:maxstack);
integer array locdiag(1:n+1), son(1:n), brother(1:n),
    wherenext(1:numa), lastfirst(1:n);
integer    n, j, first, firstson, numcol, temp,
      stackptr, numa, maxstack, i, j;

for j: = 1 to n do lastfirst(j):=1;
stackptr: = 0;
for j: = 1 to n do begin
   firstson: = son(j);
   numcol: = locdiag(j+1) - locdiag(j);
   if (firstson < > 0) then begin
        first: = 1;
        temp: = stackptr;'
        stackptr: = stackptr + numcol;
        dfs (j, firstson, temp, first);
        for i: = 1 to numcol do
           aa(locdiag(j)+i): = aa (locdiag(j)+i)
                    stack(temp + i)
        end (if firstson < > 0);
   /* d(j,j): = a(j,j) * /
   /* l(i,j): = a(i,j) / d(j,j) * /
   for i: = 2 to numcol do
      aa(locdiag(j) + i): = aa(locdiag(j) + i) /
           aa(locdiag(j) + 1)
 e n d  (for j: = 1 to n);

procedure  dfs(k, kp, tempk, firstly')
integer k, kp, tempk, firstkj; value kp;
begin   real multpyr;
   integer loc0, locfirst, loclast, myfirst,
     i, numcol, tempkp, ii;

/*   loop over brothers */
     while  (kp < > 0) do 'begin
```

```
        loc0: = locdiag (kp );
        loclast : = locdiag(kp+1);
/*      test whether kp is in row(j) * /
        myfirst: = lastfirst(kp) + 1;'
        if  (myfirst <loclast - loc0 . and
             wherenext (loc0 + myfirst) = firstkj) do begin
             locfirst: = loc0 + myfirst;
             lastfirst(kp): = myfirst;
             numcol: = loclast - locfirst;
/*      allocate   temp storage for column kp * /
             tempkp: = stackptr;
             stackptr: = stackptr + numcol;
/*      initialize  temp with contribution of column kp * /
             multpyr: = aa(locfirst) * aa(loc0 + 1)
             for i: = 1 to numcol
                  stack(tempkp + i): = aa(locfirst - 1+i)*multpyr;
/*      get contributions  of  subtree  below  kp  * /
             if  (son(kp) < > 0) then
                  dfs(kp, son(kp), tempkp, myfirst);
/*      add contribution to that of column k * /
             for i: = 1  to numcol do
                  ii: = wherenext(locfirst + i - 1),
                        - firstkj + 1
                  stack(tempk + ii): = stack(tempk + ii)
                      + stack(tempkp + i);
/*      replace tempkp and go to next brother * /
             stackptr: = stackptr - numcol
             end (if myfirst < loclast . ..);
        kp: = brother(kp)
        end (while kp < > 0)
end (dfs)
end(main program)
```

```
procedure fwdsolve;
begin integer tempn;
     tempn: = stackptr: = 0;
     dfslower (son(n), tempn);
     bb(n): = (bb(n)-stack(tempn + 1)) / aa(locdiag(n) + 1)
     end (fwdsolve)

procedure dfs lower (kp, tempk);
     integer kp, tempk; value kp;
     begin integer  numcol, tempkp,i,
          while (kp < > 0) do begin
            if (son(kp) < > 0) then begin
               numcol: = locdiag(kp + 1) - locdiag(kp);
               tempkp: = stackptr;
               stackptr: = stackptr + numcol;
               for i: = 1 to numcol do
                     stack(tempkp + i): = 0;
               dfs lower (son(kp), tempkp);
               bb(kp):= bb(kp) - stack(tempkp + 1);
               for i: = 2 to numcol do
                  stack(tempk + wherenext(locdiag(kp) + i): =
                  stack(tempkp + i) + b(kp) * aa(locdiag(kp) + i);
                  stackptr: = stackptr + numcol;
             end (then)
           else
             for i: = 2 to numcol do
                 stack(tempk + wherenext(locdiag(kp) + i)): =
                 bb(kp) * aa(locdiag(kp) + i);
/*  b overwritten with D^-1 z * /
           bb(kp): = bb(kp) / aa(locdiag(kp) + 1);
/*  go to next brother of kp * /
           kp: = brother (kp)
           end (while kp < > 0)
         end (dfs lower)
```

```
procedure backsolve;
    begin  integer templ;
       templ: = 0;  stack(templ + 1): = bb(n);
       stackptr: = templ + 1;
       dfs upper(son(n), templ);
       end (backsolve);


procedure  dfs upper(kp, temp);
   integer kp, tempk; value kp;
   begin integer  numrow, tempkp, j;
   real sum, xkj;
   while (kp < > 0) do begin
      numrow: = locdiag(kp + 1) - locdiag(kp);
/*     dot product of sparse x vector with row of L^T * /
/*     and create new temporary of x(j), j∈col(kp). * /
     tempkp: = stackptr;
     stackptr: = stackptr + numrow;
     sum: = 0;
     for j: = 2 to numrow do begin
          xkj: = stack(tempk + wherenext(locdiag(kp) + j));
          sum: = sum + aa(locdiag(kp) + j) * xkj;
          stack(tempkp + j): = xkj
          end (for j);
/*     compute b(kp) and add to temporary row * /
     bb(kp): = bb(kp) - sum;
     stack(tempkp + 1): = bb(kp);
     dfs upper (son(kp), tempkp);
/*   release temp storage, loop * /
     stackptr: = stackptr - numrow;
     kp: = brother(kp)
     end(while kp < > 0)
end (dfs upper)
```

REFERENCES.

1. G. Birkhoff and J. A. George.
Elimination by nested dissection.
In J. F. Traub, editor, Complexity of Sequential and Parallel Numerical Algorithms, Academic Press, 1973.

2. S. C. Eisenstat, M. C. G rsky, M. H. Schultz, and A. H. Sherman.
Yale sparse matrix package I. The symmetric codes.
Yale Computer Science Department Research Report # 112.

2a. S. C. Eisenstat, M. H. Schultz, and A. H. Sherman.
Software for sparse Gaussian elimination with limited core storage.
In Iain S. Duff and G. W. Stewart, eds.,
Sparse Matrix Proceedings,
SIAM, 1978.

3. Alan George.
Nested dissection of a regular finite element mesh.
SIAM Journal on Numerical Analysis 10 : 345-363, 1973.

4. Alan George.
Numerical experiments using dissection methods to solve n by n grid problems.
SIAM Journal on Numerical Analysis 14 : 161-179, 1977.

5. A. George, W. G. Poole, and R. G. Voigt.
Analysis of dissection algorithms for vector computers.
ICASE Report.

6. A. George, W. G. Poole, and R. G. Voigt.
Incomplete nested dissection for solving n by n grid problems.
SIAM Journal on Numerical Analysis 15 : 662-673, 1978.

7. A. George and Joseph W. H. Liu.
An automatic nested dissection algorithm for irregular finite element problems.
SIAM Journal on Numerical Analysis 15 : 1053-1069, 1978.

8. A. George and Joseph W. H. Liu.
An optimal algorithm for symbolic factorization of symmetric matrices.
Research Report CS-78-11, Faculty of Mathematics, University of Waterloo, Waterloo, Canada.

9. A. George and Joseph W. H. Liu.
A quotient graph model for symmetric factorization.
Research Report CS-78-04, Faculty of Mathematics, University of Waterloo, Waterloo, Canada.

10. Fred G. Gustavson.
Some basic techniques for solving sparse systems of linear equations.
In Donald J. Rose and Ralph A. Willoughby, editors, Sparse Matrices and their Applications, Plenum, New York, 1972.

11. Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan.
Generalized nested dissection.
SIAM Journal on Numerical Analysis 1.6 : 346-358, 1979.

12. D. J. Rose, R. E. Tarjan, and G. S. Lueker.
Algorithmic aspects of vertex elimination on graphs.
SIAM Journal on Computing 5 : 266-283, 1975.

13. Andrew Harry Sherman.
On the Efficient Solution of Sparse Systems of Linear and Nonlinear Equations.
Ph.D. Thesis, Yale University, 1975.

Acknowledgement

Phillip Keukes came up with.the idea of a relative pointer scheme. My discussion with him were essential to the initiation of this work.