ALGORITHMS FOR MATRIX PARTITIONING AND THE NUMERICAL SOLUTION OF FINITE ELEMENT SYSTEMS

Alan George
Joseph W.H. Liu

Research Report CS-76-30

Department of Computer Science

University of Waterloo
Waterloo, Ontario, Canada

May 1976

Work by the first author was supported in part by Canadian National Research Council grant A8111. The second author's work was supported in part by an Ontario Graduate Fellowship.

ABSTRACT

Let Ax = b be a sparse positive definite system of equations arising from the use of the finite element method to solve a two dimensional boundary value problem. A common method of solving these matrix problems is to use Cholesky's method together with an ordering which yields a small bandwidth or profile. This approach is reasonably efficient provided that the associated finite element mesh does not have appendages and/or holes. In this paper algorithms are described for finding orderings and partitionings of sparse finite element matrix problems. These allow the use of computational and storage techniques which lead to substantial improvements over standard solution methods when the associated mesh has appendages and/or holes. The issue of storage and execution time trade-offs naturally arises and is discussed.

§1 Introduction

In this paper we consider the problem of solving the N by N sparse symmetric systems of linear equations $Ax = b$ that arise in connection with the use of the finite element method. The method of solution is the standard Cholesky algorithm, where A is factored into LL^T, where L is lower triangular, followed by the solution of the triangular systems $Ly = b$ and $L^T x = y$. For positive definite A, no interchanges are required to maintain numerical stability [18]. Thus, we may instead solve the equivalent system

$$(1.1) \qquad (PAP^T)(Px) = Pb,$$

where P is any N by N permutation matrix.

When Cholesky's method is applied to A, the matrix usually suffers fill; that is, the triangular factor L will usually have nonzeros in some positions which are zero in A. For $P \neq I$, PAP^T will in general fill in differently, and it is well known that a judicious choice of P can sometimes lead to dramatic reduction in fill and/or the amount of arithmetic required to solve (1.1).

A common choice for P is one for which PAP^T has a small bandwidth or profile. For a given symmetric matrix M with nonzero diagonal components, its bandwidth $\beta(M)$ and envelope $Env(M)$ are defined as follows.

$$(1.2) \qquad \beta(M) = \max_{M_{ij} \neq 0} |i-j|.$$

$$(1.3) \qquad Env(M) = \{(i,j) | i \geq j \text{ and } j \geq f_i\},$$

where $f_i = \min\{j | M_{ij} \neq 0\}$, $1 \leq i \leq N$.

We define the profile of M as $|Env(M)|$, where $|S|$ denotes the cardinality of the finite set S. It is easy to establish that when no row and column interchanges are performed during the Cholesky factorization, all fill is confined to the envelope, and therefore also within the band, since $(i,j) \in Env(M) \Rightarrow |i-j| \leq \beta(M)$. Implicit in the use of small band or profile orderings is the assumption that zeros outside the band or envelope are to be exploited, but zeros within the band or envelope are normally not exploited. An efficient storage scheme by Jennings for utilizing such orderings has been described in [13].

If we are prepared to exploit all zeros, orderings which yield a small band $\beta(A)$ or $|Env(A)|$ may be far from optimal in the least-fill or least-arithmetic sense [4 ,15]. However, the use of such schemes is often a reasonable compromise between low arithmetic requirements and/or fill on one hand, and simple data structures and programming on the other. Optimal or near-optimal orderings (in the least-arithmetic or fill senses) characteristically lead to triangular factors which have their nonzero components scattered throughout the matrix, and relatively sophisticated data structures and programs are required to effectively exploit such sparsity [12]. Robust algorithms for generating optimal or near-optimal orderings for sparse matrix problems are not yet well developed, but there are several band and/or envelope reduction algorithms which experience has shown to be quite effective for a large class of problems, including the problems we consider in this paper and which we now characterize.

Let M be a planar mesh consisting of the union of triangles and/or quadrilaterals called elements, with adjacent elements sharing a common side or a common vertex. There is a node at each vertex of M, and there may also be nodes lying on element sides, and perhaps in the interior of each element. An example of such a mesh appears in Figure 1.1.

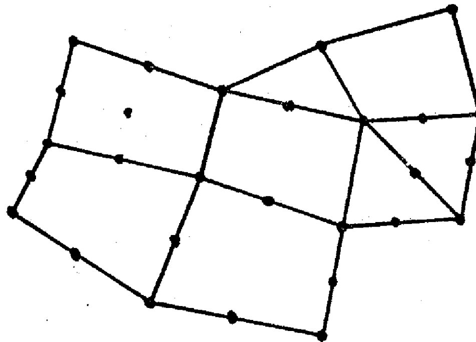


Figure 1.1 A 28 node finite element mesh having 8 elements.

For purposes of this paper, we associate one variable x_i with each of the N nodes of M. For some labelling of these nodes of M from 1 to N, we define a finite element system $Ax = b$ associated with M as one where A is symmetric and positive definite and for which $A_{ij} \neq 0 \Rightarrow$ variables x_i and x_j are associated with nodes of the same element. Thus, relabellings of M correspond to symmetric permutations of the system, as implied by (1.1). This definition is not quite general enough to cover many matrix problems which arise in finite element applications because more than one variable is often associated with each node. However, the extension of our ideas

to this situation is immediate, so to avoid needless complication we assume only one variable is associated with each node. Our test problems are of this type, but our algorithms and codes make no use whatsoever of the assumption.

The band and profile reduction algorithms commonly used on such finite element matrix problems work quite well when the associated finite element meshes are "featureless"; i.e., when they do not have "appendages" and/or "holes". However, when the meshes do have these features, it is possible to find orderings and solution methods which are considerably more efficient than standard band or profile methods, but which still retain the advantages of fairly simple data structures when compared to those needed for least-fill or least-arithmetic orderings. Our objective in this paper is to explore this "middle ground" between the standard band/profile orderings and the near optimal schemes.

We now outline our paper. In section 2 we review some basic sparse matrix techniques involving matrix partitioning [5], and then show how they can be recursively applied in a natural way. We also include some implementation details. Section 3 contains two examples which motivate our implementation scheme of section 2 and the ordering algorithms we describe in sections 5 and 6. Section 4 contains some basic graph theoretic notions as they relate to Cholesky's method. We also make the connection between a level structure in a graph, which is a central construct in many ordering algorithms (including the ones we propose in this paper), and the partitioning it induces in the matrix problem associated with the graph. We also review the significance of the class of graphs

called trees in connection with matrix orderings, and extend some of the important ideas to partitioned matrices. Section 4 also contains a brief review of two existing ordering algorithms, parts of which are used in a modified form in our refined quotient tree algorithm (RQT),which is described in section 5. Section 6 contains a description of our hole removal algorithm (RH), and section 7 describes how these two algorithms are combined with the computational scheme of section 2 to form a complete solution package called RH-RQT-BLKSLV. Section 7 also contains numerous numerical experiments comparing the performance of our solution schemes to some standard band/profile methods. Section 8 contains our concluding remarks.

§2 Sparse Matrix Techniques Associated with Partitioning, and their Recursive Application

§2.1 Review of Block Factorization Techniques

Suppose the problem $Ax = b$ is partitioned as shown in (2.1) below.

$$(2.1) \qquad \begin{pmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} .$$

The Cholesky factor L of A, correspondingly partitioned, is

$$(2.2) \qquad \begin{pmatrix} L_{11} & 0 \\ W^T & L_{22} \end{pmatrix},$$

where $L_{11}L_{11}^T = A_{11}$, $W = L_{11}^{-1}A_{12}$, and

$$(2.3) \qquad L_{22}L_{22}^T = \tilde{A}_{22} = A_{22} - A_{12}^T A_{11}^{-1} A_{12}.$$

Here and elsewhere in this paper it is understood that inverses are not computed explicitly; instead, the appropriate triangular systems are solved.

Now there are two important ideas which can be exploited in the above computations when A is sparse, and the value of these ideas hinges on the observation that A_{12} is frequently much sparser than W . When this is true, it may require fewer arithmetic operations to compute $\tilde{A}_{22}$ as $A_{12}^T(L_{11}^{-T}(L_{11}^{-1}A_{12}))$ rather than as $(A_{12}^T L_{11}^{-T})(L_{11}^{-1}A_{12}) = W^T W$, which is the way the computation is effectively done using any of the standard Cholesky factorization algorithms. The second important idea is that we do not need to retain W in order to carry out the solution of the triangular systems $Ly = b$ and $L^T x = y$. During these computations we

need to compute W^Ty_1 and Wx_2 (where x and y are partitioned corresponding to A), but these can be computed as $A_{12}^T(L_{11}^{-T}y_1)$ and $L_{11}^{-1}(A_{12}x_2)$ if A_{12} is available. If A_{12} is much sparser than W, we can save storage and perhaps arithmetic as well by using W in this implicit manner. An important point to note is that if we plan to discard W anyway, computing $\tilde{A}_{22}$ in the "asymmetric" way implied by $A_{12}^T(L_{11}^{-T}(L_{11}^{-1}A_{12}))$ requires very little temporary storage because $\tilde{A}_{22}$ can be computed <u>one column at a time</u>. On the other hand, if the product is calculated as W^TW, there seems to be no way to avoid storing all of W, even if we do not intend to retain it. The reader is referred to [5] for a complete analysis of the points we have just discussed. We will distinguish between the symmetric and asymmetric versions of the computation by F_1 and F_2 respectively.

§2.2 Recursive Application of Partitioning

We now generalize these techniques, as suggested by George [5 , page 586]. Let A be symmetrically partitioned into p^2 submatrices A_{rs}, $1 \le r,s \le p$, and let L_{rs} be the corresponding submatrices of L, where $A = LL^T$. Define the following matrices for $2 \le k \le p$.

$$(2.4) \qquad B_k = \begin{bmatrix} A_{1k} \\ A_{2k} \\ \vdots \\ A_{k-1,k} \end{bmatrix}, \qquad W_k = \begin{bmatrix} L_{k1}^T \\ L_{k2}^T \\ \vdots \\ L_{k,k-1}^T \end{bmatrix}.$$

For any N-vector $v = (v_1,v_2,\ldots,v_p)^T$, partitioned corresponding to A, let $v_{(k)} = (v_1,v_2,\ldots,v_k)^T$, $1 \le k \le p$. Similarly, for any N by N matrix M partitioned as A, let $A_{(k)}$ be the leading principal submatrix of A

obtained by deleting all blocks $A_{r,s}$ for which $r > k$ and/or $s > k$. It is easy to verify that $W_k = L^{-1}_{(k-1)}B_k$. These definitions are illustrated for $p = 5$ in (2.5) below.

$$(2.5)\qquad A = \begin{bmatrix} A_{11} & B_2 & B_3 & & \\ A^T_{12} & A_{22} & & B_4 & \\ A^T_{13} & A^T_{23} & A_{33} & & B_5 \\ A^T_{14} & A^T_{24} & A^T_{34} & A_{44} & \\ A^T_{15} & A^T_{25} & A^T_{35} & A^T_{45} & A_{55} \end{bmatrix}, \quad L = \begin{bmatrix} L_{11} & & & & \\ W^T_2 & L_{22} & & \bigcirc & \\ W^T_3 & & L_{33} & & \\ W^T_4 & & & L_{44} & \\ W^T_5 & & & & L_{55} \end{bmatrix}.$$

Now as before, our intention is to retain only the L_{kk}, $1 \le k \le p$, and the B_k, $2 \le k \le p$. The algorithms for solving $Ly = b$ and $L^Tx = y$ are given before the factorization algorithm because the latter one uses the triangular solvers.

<u>Solution of $Ly = b$</u> ("lowersolve (L,y,b,p)")

1) Solve $L_{11}y_1 = b_1$

2) If $p > 1$ then for $k = 2,3,\dots,p$ do the following:

   (2.1) Solve $L^T_{(k-1)}\tilde{y}_{(k-1)} = y_{(k-1)}$ (using "uppersolve" below with the appropriate arguments).

   (2.2) Compute $\tilde{b}_k = B_k{}^T\ \tilde{y}_{(k-1)}$

   (2.3) Solve $L_{kk}y_k = \tilde{b}_k$.

<u>Solution of $L^Tx = y$</u> ("uppersolve(L,y,x,p)")

1) Solve $L^T_{pp}x_p = y_p$

2) If $p > 1$ then for $k = p-1,p-2,\dots,1$ do the following:

   (2.1) Compute $z_{(k)} = B_{k+1}x_{k+1}$

(2.2) Solve $L_{(k)}\tilde{z}_{(k)} = z_{(k)}$ (using "lowersolve" above with the appropriate arguments)

(2.3) Set $y_{(k)} = y_{(k)} - \tilde{z}_{(k)}$

(2.4) Solve $L_{kk}^T x_k = y_k$

Note that the algorithms lowersolve and uppersolve are mutually recursive; i.e., in general, lowersolve invokes uppersolve, which in turn invokes lowersolve.... . We return to the implications of this after describing the factorization algorithm.

Factorization algorithms ("factor$_{F_1}$(A,p)" and "factor$_{F_2}$(A,p)")

(1) Factor A_{11}.

(2) If $p > 1$ then for $k = 2,3,\ldots,p$ do the following:

(2.1) F_1 version: solve $L_{(k-1)}W_k = B_k$, (using "lowersolve") and compute $\tilde{A}_{kk} = A_{kk} - W_k^T W_k$.

OR

(2.1)' F_2 version: for each column u of B_k, solve $A_{(k-1)}v = u$ (using "lowersolve" and "uppersolve"), compute $\tilde{v} = B_k^T v$, and subtract $\tilde{v}$ from the appropriate column of A_{kk}, yielding that column of $\tilde{A}_{kk}$.

(2.2) Factor $\tilde{A}_{kk}$.

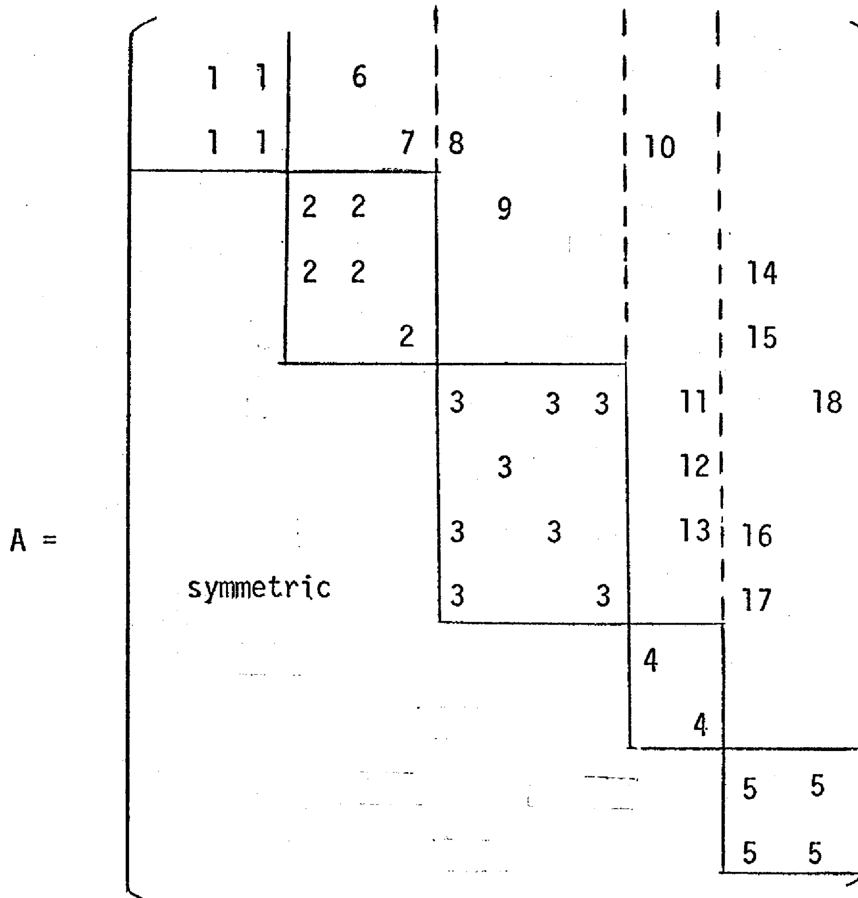
Note that in the F_1 version, temporary storage for the largest W_k must be available, while for the F_2 version the single vectors u and v (of length less than N) are the only temporary arrays needed. Of course, in both versions the symmetry of A_{22} and $\tilde{A}_{22}$ is exploited.

§2.3 The Implicit Storage Scheme

We now make some remarks about the implicit storage scheme we use for L. We store the diagonal blocks of L row by row, beginning with the first nonzero component in each row, in a single one dimensional array ENV. An additional vector XENV of length N is used to record the positions of the diagonal components of L in ENV. This scheme is due to Jennings [12]. In our application we require an addition vector XBLK of length p to record the row number of L corresponding to the first row of each block.

The nonzero components of B_k, $1 \le k \le p$ are all stored in a single one dimensional array NONZ, column by column, beginning with those of B_1. A parallel integer array SUBS is used to store the row subscripts of the numbers in NONZ, and a vector XNONZ of length N contains the positions in NONZ where each column resides. Figure 2.1 is an example of a matrix stored using this storage scheme. The array ENV contains the components of the A_{kk}, which are overwritten by those of L_{kk} during the factorization. For convenience, we set XNØNZ(N+1) = |NONZ|+1, where |NONZ| denotes the number of components in NONZ. Similarly, we set XBLK(p+1) = N+1 and XENV(N+1) = |ENV|+1. Note that XNONZ(i+1) = XNONZ(i) implies that the corresponding column of B_k is null.

The storage required for the vectors XENV, XBLK, XNØNZ, and SUBS must be regarded as overhead storage, since it is not used to store actual data. In addition, in our implementation of the F_2 version of the factorization we found it convenient to have an extra temporary vector of length N. In the F_1 version, we need temporary storage for the largest W_k which occurs, along with an integer vector equal to the maximum number of columns in any of the W_k. This temporary storage, as well as overhead storage, is sometimes reported separately in our numerical experiments



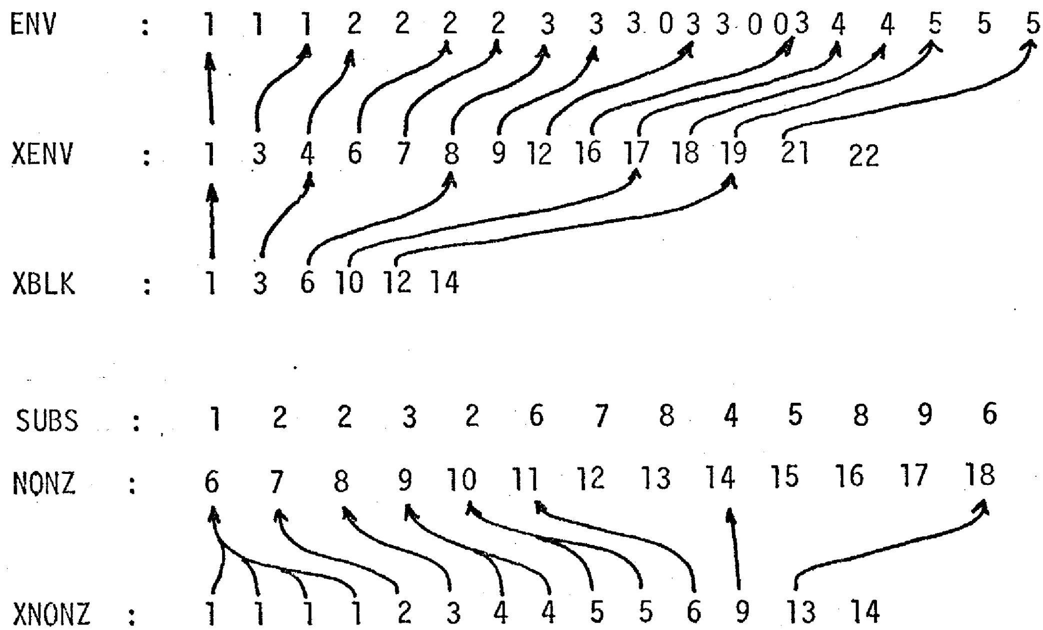


Figure 2.1 Example showing the arrays used in our storage scheme

in section 8, along with the primary storage used, which is that used for the actual matrix components. In section 8, total storage refers to the sum of these three components, along with storage for the right hand side and the solution.

Now it should be clear that our storage scheme will be most efficient if the partitioning is "fine"; obviously the limiting case is when $p = N$, and then we have a direct method for solving $Ax = b$ which (on the surface) appears to require storage proportional only to the number of nonzeros in A. However, the temporary storage needed to support the recursion (specifically, the vector z in "uppersolve") rises with the recursion depth, so there is in general a point where a finer partition no longer reduces total storage requirements. In addition, our experience with some promising partitioning strategies suggests that the computational cost rises sharply with the depth of recursion that is required by the partitioning. For example, it is not difficult to show that under several reasonable hypotheses on A, the cost of executing the algorithms we have just described is $O(2^N)$. The implications of using various partitionings on matrices of various classes, along with storage and time trade-off considerations, is explored in detail elsewhere. In this paper our concern is to find partitionings and orderings of our finite element problems for which our storage scheme is appropriate, and for which the recursive nature of the lowersolve and uppersolve algorithms is to a large extent unnecessary. These remarks will become clear after the discussion of the two examples in section 3.

§3 Two Motivating Examples

Suppose we have a finite element mesh consisting of $(n-1)^2$ square elements, and we number the nodes row by row from one to $N = n^2$. The matrix A will have the familiar block tri-diagonal form shown below, where each block corresponds to a grid line and is n by n, and n is assumed to be 5 for demonstration purposes.

$$(3.1) \qquad A = \begin{bmatrix} A_{11} & A_{12} & & & \\ A_{12}^T & A_{22} & A_{23} & & \bigcirc \\ & A_{23}^T & A_{33} & A_{34} & \\ \bigcirc & & A_{34}^T & A_{44} & A_{45} \\ & & & A_{45}^T & A_{55} \end{bmatrix}$$

Here the matrices B_k and W_k have the form

$$(3.2) \qquad B_k = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ A_{k-1,k} \end{bmatrix}, \quad W_k = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ L_{k-1,k-1}^{-1} A_{k-1,k} \end{bmatrix},$$

and the important point to note is that W_k depends explicitly only on $L_{k-1,k-1}$, and no W_j, $j < k$. Thus, the procedures lowersolve and uppersolve need not recursively call each other for this problem. It is easy to verify that for large n, our storage requirement will be about one half of that needed for the standard band or profile storage scheme. Moreover, the computation requirements are essentially the same.

Now consider the "annulus" mesh shown below.

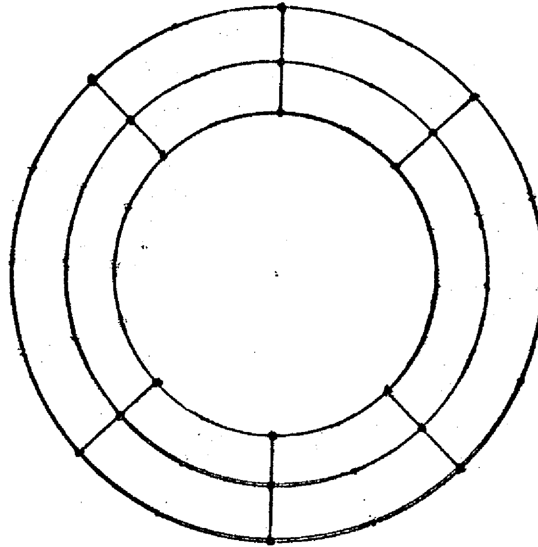


Figure 3.1 An annulus with thickness r=3 and circumference q= 6, yielding N=qr.

Suppose we number the "rays" of the mesh consecutively, proceeding clockwise (or counter-clockwise) around the annulus until we reach the ray at which we started. The resulting matrix structure will be as shown in (3.3) below.

$$
(3.3) \qquad A = \begin{bmatrix} A_{11} & A_{12} & & & & A_{16} \\ A_{12}^T & A_{22} & A_{23} & & \bigcirc & \\ & A_{23}^T & A_{33} & A_{34} & & \\ & \bigcirc & A_{34}^T & A_{44} & A_{45} & \\ & & & A_{45}^T & A_{55} & A_{56} \\ A_{16}^T & & & & A_{56}^T & A_{66} \end{bmatrix}
$$

In this case we have

$$B_k = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ A_{k-1,k} \end{bmatrix} \quad \text{and} \quad W_k = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ L_{k-1,k-1}^{-1} A_{k-1,k} \end{bmatrix}, \ 1 \le k \le 5,$$

as in the previous example. However, $W_6 = L_{(5)}^{-1} B_6$, so in order to execute the algorithms lowersolve and uppersolve of section 2, <u>one level of recursion</u> must be admitted.

Why choose this ordering? The main objective is to achieve a reduction in storage requirements. If we were to number the mesh to achieve a small bandwidth β, and then partition the problem into blocks of size about $\beta+1$ by $\beta+1$, the amount of storage required using our implicit storage scheme would be $qr^2+O(qr)$ for large q and r, since β turns out to be about 2r. On the other hand, using the ordering (3.3), our implicit storage scheme requires about $\frac{1}{2}qr^2+O(qr)$. The arithmetic operation count (multiplications and divisions) is increases slightly, however, from $\approx 2r^3q$ to $\approx \frac{5}{2}r^3q$. Thus, for this problem, we have exchanged an increase of about 25 percent in arithmetic for a decrease of about 50 percent in storage. In typical computing environments, this exchange would result in a substantial reduction in dollar cost.

These examples along with section 2 provide the motivation for our overall ordering algorithm and solution scheme. Roughly speaking, we first find a "tearing set" of nodes which, when removed from the mesh along with their incident edges, yields a mesh which has no holes. This

"remove holes" (RH) algorithm is described in section 6, and the tearing set corresponds to the <u>last</u> partition member of the matrix A, say A_{pp}. In general, W_p is a function of $L_{(p-1)}$. An ordering and partitioning of the remaining mesh is then found, using the RQT algorithm described in section 5, such that all the W_k in $L_{(p-1)}$ depend <u>only</u> on one or more of the L_{jj}, $j < k$. Thus, no recursion in the procedures lowersolve and uppersolve is necessary in solving problems of the form $L_{(p-1)}x = y$ or $L^T_{(p-1)}x = y$. In other words, our algorithms produce an ordering and partitioning so that the algorithms uppersolve and lowersolve of section 2 can be executed with the occurrence of only one level of recursion.

We want to keep the level of recursion low for two reasons. First, orderings and partitionings which require a lot of recursion appear to have such high operation counts that the savings in storage are not compensatory. Second, our implementations are in Fortran, which does not support recursive procedure calls, so we want to bound the level of recursion and thus avoid simulating more than one level of recursion.

§4 Preliminaries

In this section we review some basic graph theoretic notions and introduce a few definitions that are related to our implementations of Cholesky's method. We also establish some preliminary results needed in subsequent sections.

4.1 Some basic graph-theoretic definitions

A graph $G = (X,E)$ consists of a finite nonempty set X of nodes together with a prescribed edge set E of unordered pairs of distinct nodes. A graph $G' = (X',E')$ is a subgraph of $G = (X,E)$ if $X' \subset X$ and $E' \subset E$. For $Y \subset X$, $G(Y)$ refers to the subgraph $(Y,E(Y))$ of G, where $E(Y) = \{\{u,v\} \in E | u,v \in Y\}$.

Nodes x and y are said to be adjacent if $\{x,y\}$ is an edge in E. For a subset Y of nodes, the adjacent set of Y is defined as

$$\text{Adj}(Y) = \{x \in X \backslash Y | \{x,y\} \in E \text{ for some } y \in Y\}.$$

If $Y = \{y\}$, we shall write $\text{Adj}(y)$ instead of the formally correct $\text{Adj}(\{y\})$. The degree of a node x is the number of nodes adjacent to x, denoted by $|\text{Adj}(x)|$. Sometimes, we shall refer to $y \in \text{Adj}(x)$ as a neighbor of the node x.

A path of length ℓ is a sequence of ℓ edges $\{x_0,x_1\},\{x_1,x_2\},\ldots,\{x_{\ell-1},x_\ell\}$ where all the nodes $x_0,x_1,\ldots,x_\ell$ are distinct except possibly x_0 and x_ℓ. If $x_0 = x_\ell$, it is called a cycle. A graph G is connected if there is a path connecting each pair of distinct nodes. If G is disconnected, it consists of two or more maximal connected subgraphs called components.

For a subset Y of nodes, the span of Y is defined as

$$\mathrm{Span}(Y) = \{x \in X | \exists \text{ a path from } y \text{ to } x, \text{ for some } y \in Y\}.$$

If $Y = \{y\}$, Span(Y) is simply the connected component that contains the node y. When the graph is connected, the span of any nonempty subset is the node set X itself.

Unless otherwise specified, graphs in this paper are assumed to be connected. The distance $d(x,y)$ between two distinct nodes x and y is the length of a shortest path joining them. Following Berge [2], we define the eccentricity $\ell(x)$ of a node x to be

$$\ell(x) = \max\{d(x,y) | y \in X\}.$$

The diameter $\delta(G)$ of a graph is then

$$\delta(G) = \max\{\ell(x) | x \in X\}.$$

A node x is said to be peripheral if its eccentricity is the same as the diameter of the graph.

A tree is a connected graph with no cycles, or equivalently, it is a graph where every pair of distinct nodes is joined by a unique path. For a tree $T = (X,E)$, $|X| = |E|+1$.

A rooted tree is a tree $T = (X,E)$ with a distinguished node r, called the root of T. If $\{r,s\},\ldots,\{x,y\}$ is the (unique) path from the root r to the node y, x is said to be the father of y. It is well-known that this father relationship completely characterizes the rooted tree. Thus, a rooted tree with N nodes can be represented conveniently using N storage locations.

For a graph $G = (X,E)$ with $|X| = N$, an ordering or numbering of G is a bijective mapping $\alpha:\{1,2,\ldots,N\} \rightarrow X$. We use G_α and X_α to denote the ordered graph and ordered node set respectively.

It is often convenient to view permutations on a sparse symmetric matrix as orderings on a corresponding graph structure. Let M be a symmetric matrix. We associate an undirected graph $G^M = (X^M,E^M)$ with M, such that X^M is the set of nodes corresponding to and labelled as the rows of M, and $\{x_i,x_j\} \in E^M$ if and only if $M_{ij} \neq 0$ and $i \neq j$. Note that the graph G^M has an implicit ordering defined by the matrix M. Indeed, each ordering α on G^M identifies a permutation matrix P_α on the matrix M.

4.2 Quotient graphs and level structures

Motivated by the partitioning of a matrix into block submatrices, we introduce the concept of quotient graphs. Given a graph $G = (X,E)$, let $\mathcal{P}$ be a partition on the node set X:

$$\mathcal{P} = \{Y_1,Y_2,\ldots,Y_p\}.$$

That is, $\bigcup_{i=1}^{p} Y_i = X$ and $Y_i \cap Y_j = \phi$ for $i \neq j$. We define the quotient graph of G with respect to the partition $\mathcal{P}$ to be the graph

$$G/\mathcal{P} = (\mathcal{P},\mathcal{E})$$

where $\{Y_i,Y_j\} \in \mathcal{E}$ if and only if $\mathrm{Adj}(Y_i) \cap Y_j \neq \phi$.

When $G/\mathcal{P}$ is itself a tree, we call $\mathcal{P}$ a tree partitioning and $G/\mathcal{P}$ a quotient tree. A tree partitioning $\mathcal{P} = \{Y_1,Y_2,\ldots,Y_p\}$ is said to be maximal if there does not exist a tree partitioning $\mathcal{Q} = \{Z_1,Z_2,\ldots,Z_t\}$ such that $p < t$ and for each i, $Z_i \subset Y_k$ for some $1 \leq k \leq p$. In other words, it is maximal if any finer partitioning $\mathcal{Q}$ of $\mathcal{P}$ does not yield a quotient tree. In that case, we call $G/\mathcal{P}$ a maximal quotient tree.

Consider the tree partitioning $P = \{Y_1, Y_2\}$ in Figure 4.1, where $Y_1 = \{1,2\}$, and $Y_2 = \{3,4\}$. It does not have a finer tree partitioning so that it is maximal. However, note that the graph admits a tree partitioning with 3 members: $\{\{2\},\{1,3\},\{4\}\}$.

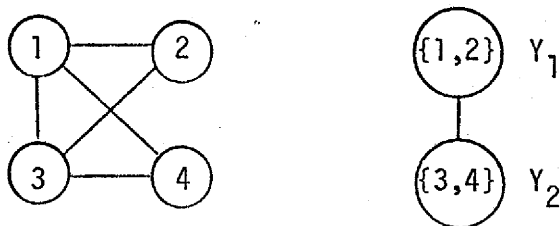


Figure 4.1 A maximal tree partitioning

We now establish a simple sufficient condition for a tree partitioning to be maximal.

Lemma 4.1 Let P be a tree partitioning. If for any $Y \in P$ and any distinct $x,y \in Y$, there exist two paths between x and y

$$x, x_1, \ldots, x_s, y$$

$$x, y_1, \ldots, y_t, y$$

such that

$$R = \cup \{Z \in P | x_i \in Z,\ 1 \leq i \leq s\}$$

and $$S = \cup \{Z \in P | y_i \in Z,\ 1 \leq i \leq t\}$$

are disjoint, then P is maximal.

Proof Assume for contradiction that P is not maximal. Then there exists a member $Y \in P$ such that Y can be decomposed into two disjoint nonempty sets U and V and yet $Q = P \cup \{U,V\} \setminus \{Y\}$ remains a tree partitioning. Pick $u \in U$ and $v \in V$. From the hypothesis in the lemma, we can find two paths from u to v:

$$u, x_1, \ldots, x_s, v$$
$$u, y_1, \ldots, y_t, v$$

where $\{x_i\}$ and $\{y_j\}$ do not belong to common partition members. Therefore in $G/\mathcal{Q}$, a circuit exists connecting U and V, so that P must be maximal. □

A simple class of quotient trees can be derived using the notion of level structures [1]. Formally, a level structure of a connected graph $G=(X,E)$ is a partition

$$\mathcal{L} = \{L_0, L_1, \ldots, L_\ell\}$$

of the node set X such that

$$Adj(L_i) \subset L_{i-1} \cup L_{i+1}, \quad i = 1, \ldots, \ell-1,$$

$Adj(L_0) \subset L_1$ and $Adj(L_\ell) \subset L_{\ell-1}$. The number ℓ is the length of the level structure. The quantity $\max\{|L_i| \mid i = 0,1,\ldots,\ell\}$ is called the width of $\mathcal{L}$. It should be clear that the corresponding quotient tree $G/\mathcal{L}$ is a simple "chain".

In practice, it is common to produce and work on rooted level structures. For a node $x \in X$, the rooted level structure at x is defined as the level structure:

$$\mathcal{L}(x) = \{L_0(x), L_1(x), \ldots, L_{\ell(x)}(x)\},$$

where
$$L_0(x) = \{x\}$$
$$L_i(x) = Adj\left(\bigcup_{j=0}^{i-1} L_j(x)\right), \quad i = 1, \ldots, \ell(x)$$

and $\ell(x)$ is the eccentricity of x.

It is an important construct in the algorithm developed in section 5.

We now make the connection between a level structure of a graph and the partitioning it induces on a matrix associated with the graph. Let G^A be the graph associated with a symmetric matrix A and let $\mathcal{L}$ be a level structure in G^A. The quotient graph $G^A/\mathcal{L}$ is a chain, so if we number the nodes in each level L_i consecutively from L_0 to L_ℓ, the levels in $\mathcal{L}$ induce a block tridiagonal partitioning on the permuted matrix. Figure 4.2 contains an example.

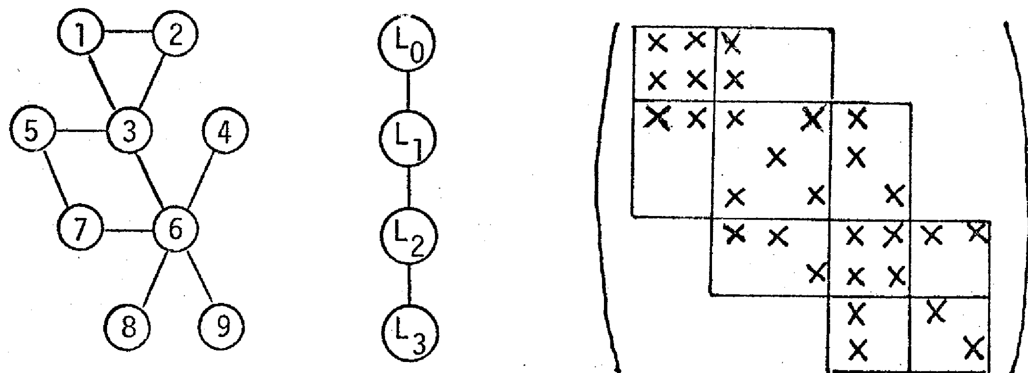


Figure 4.2 Block tridiagonal partitioning induced by a level structure

4.3 Elimination on tree structures

In [14], Parter studied the effect of Gaussian elimination on matrices associated with tree structures. In this context, he introduced the class of monotone orderings for rooted trees. In our notation, a <u>monotone ordering</u> α for a rooted tree is one that always numbers a node before its father. Clearly, the root is always numbered last by a monotone ordering.

We can extend this definition to general trees. Let α be an ordering on a tree $T = (X,E)$, where $|X| = N$. We call α a monotone ordering for T if it is one for the tree rooted at the node $\alpha(N)$. The following lemma is due to Parter [14].

Lemma 4.2 Let A be an N by N symmetric matrix associated with a monotonely-ordered tree. If $A = LL^T$ where L is the triangular factor of A, then $A_{ij} = 0 \Rightarrow L_{ij} = 0$, for $i \neq j$. □

In other words, matrices associated with monotonely-ordered trees do not have any fill in Gaussian elimination. We now extend this lemma in the following form.

Lemma 4.3 Let A and L be as in Lemma 4.2. Then $L_{ij} = L_{jj}^{-1} A_{ij}$ for $i \neq j$.

Proof In Gaussian elimination, the components of L are given by:

$$L_{ij} = L_{jj}^{-1}(A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}) \quad \text{for } i \neq j.$$

It is sufficient to show that $\sum_{k=1}^{j-1} L_{ik}L_{jk}$ is zero. Assume for contradiction that $L_{ik}L_{jk} \neq 0$ for some $k = 1,\ldots,j-1$. By Lemma 4.2, we have $A_{ik}A_{jk} \neq 0$, so that the node x_k is connected to both x_i and x_j. This contradicts the assumption that the associated tree is monotonely-ordered. □

Lemma 4.3 extends immediately to block matrices. Specifically, if each A_{ij}, $i \neq j$ represents a sparse submatrix rather than a nonzero component of A, the corresponding block L_{ij} in the triangular factor is given by $L_{ij}^T = L_{jj}^{-1} A_{ji}$. Thus, if the quotient graph of the partitioned matrix A is a monotonely ordered tree, all the W_k defined in section 2 have the simple definition

$$W_k = \begin{bmatrix} L_{11}^{-1}A_{1k} \\ L_{22}^{-1}A_{2k} \\ \vdots \\ L_{k-1,k-1}^{-1}A_{k-1,k} \end{bmatrix},$$

where of course some or all of the A_{jk} may be zero. The algorithms "lowersolve" and "uppersolve" need not involve <u>any</u> recursion for quotient tree orderings, because W_k does not explicitly depend on any W_j, $j < k$.

§5 The Refined Quotient Tree Ordering Algorithm [8]

5.1 A maximal tree partitioning

To use the implicit storage scheme effectively, it is apparent that we want to find a partition $P = \{Y_1, Y_2, \ldots, Y_p\}$ with as many members as possible and consistent with the property that G/P remains a quotient tree. This motivates the study in this section. We shall give a maximal tree partitioning, based on a rooted level structure.

Let $G = (X,E)$ be a connected graph and r be a node in X. As pointed out in section 4, the rooted level structure $\mathcal{L}(r) = \{L_0(r), L_1(r), \ldots, L_{\ell(r)}(r)\}$ forms a tree partitioning. In general, it may not be maximal. In what follows, we refine the structure $\mathcal{L}(r)$ to yield a maximal tree partitioning.

For each $j = 0,1,\ldots,\ell(r)$, let $X_j = \bigcup_{i=j}^{\ell(r)} L_i(r)$ and denote $G(X_j) = (X_j, E(X_j))$. In other words, $G(X_j)$ is the graph G without the first j levels and their incident edges. Then, each level $L_j(r)$ is refined into:

$$\{Y \mid Y = L_j(r) \cap C, \text{ for some component } C \text{ in subgraph } G(X_j)\}. \tag{5.1}$$

For example, in Figure 5.2, $L_4 = \{5,9,11,14,17\}$ is refined as $L_4 = \{5,11\} \cup \{9,14,17\}$.

Let $P = \{Y_1, Y_2, \ldots, Y_p\}$ be the refined partitioning obtained by (5.1). We now establish the maximality of this new partitioning.

Lemma 5.1 G/P is a quotient tree.

Proof: Note that G/P is a connected graph with p nodes and p-1 edges. □

Lemma 5.2 Let $Y \in P$, and $Y \subset L_j(r)$. For any two nodes x and y in Y, there exists a path between them in the subgraph $G(\bigcup_{i=0}^{j-1} L_i(r))$ and one in $G(\bigcup_{i=j}^{\ell(r)} L_i(r))$.

Proof Since $y,z \in L_j(r)$, there must be a path through them in the first j levels. On the other hand, by the definition of Y in (5.1), x and y belong to the same connected component in $G(X_j) = G(\bigcup_{i=j}^{\ell(r)} L_i(r))$. So, a path exists in $G(X_j)$ joining x to y. □

Theorem 5.1 G/P is a maximal quotient tree.

Proof That G/P is maximal follows from lemma 5.2 and lemma 4.1. □

For a given node r, to find this refined partitioning of $\mathcal{L}(r)$, it is not necessary to determine the connected components of the subgraphs $G(X_j)$ explicitly. An algorithm is described below, and it uses a stack to store partially formed partition members.

Step 0 (Initialization): Empty the stack. Generate the rooted level structure at r: $\mathcal{L}(r) = \{L_0, L_1, \ldots, L_{\ell(r)}\}$. Pick a node y in the last level $L_{\ell(r)}$, let $k \leftarrow \ell(r)$ and $S \leftarrow \{y\}$.

Step 1 (Pop stack): If the node set $\bar{S}$ on the top of the stack belongs to L_k, pop $\bar{S}$ from the stack and let $S \leftarrow S \cup \bar{S}$.

Step 2 (Form possible partition member): Determine the set $Y \leftarrow \text{Span}(S)$ in the subgraph $G(L_k)$. If some node in $\text{Adj}(Y) \cap L_{k+1}$ has not been selected in any partition member yet, go to step 5.

<u>Step 3</u> (New partition member): Put Y into P.

<u>Step 4</u> (Next level): Determine the set $S \leftarrow Adj(Y) \cap L_{k-1}$. Set $k \leftarrow k-1$. If $k \geq 0$, go to step 1, otherwise stop.

<u>Step 5</u> (Partially formed partition member): Push the set S onto the stack. Pick a node $y_{k+1} \in Adj(Y) \cap L_{k+1}$. Trace a path $y_{k+1}, y_{k+2}, \ldots, y_{k+t}$ where $y_{k+i} \in L_{k+i}$ and $Adj(y_{k+t}) \cap L_{k+t+1} = \phi$. Let $S \leftarrow \{y_{k+t}\}$ and $k \leftarrow k+t$; go to step 2.

The example in Figure 5.1 illustrates how the algorithm operates. The rooted level structure at node 1 is shown in Figure 5.2. On applying the algorithm to $\mathcal{L}(1)$, we obtain a maximal quotient tree with ten nodes. In this example, $Y_1 = \{20\}$, $Y_2 = \{18,19\}$, $Y_3 = \{16\}$, $Y_4 = \{10,15\}$, $Y_5 = \{9,14,17\}$ and $Y_6 = \{5,11\}$. So, we have $L_4 = Y_5 \cup Y_6$, $L_5 = Y_2 \cup Y_4$ and $L_6 = Y_1 \cup Y_3$.

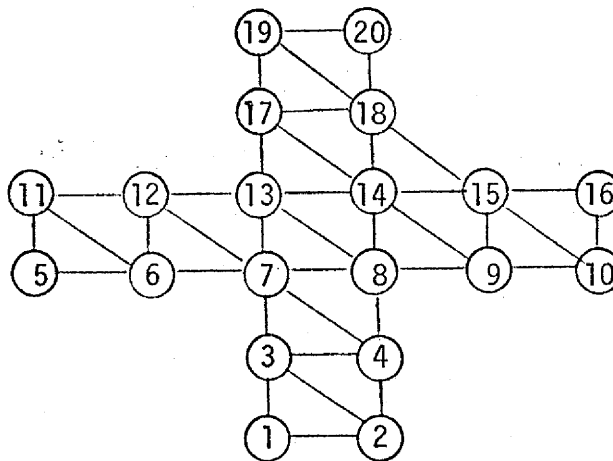


Figure 5.1 A '+' shaped graph

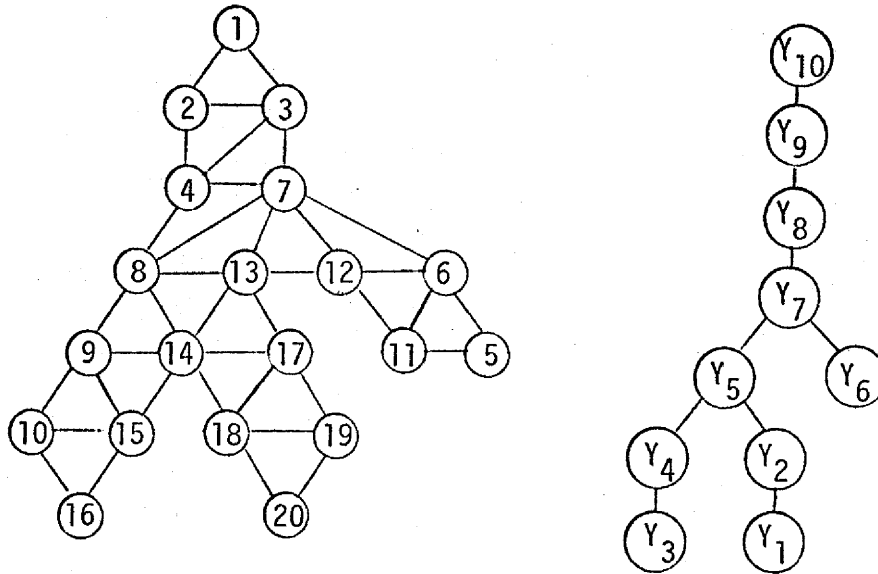


Figure 5.2 Rooted level structure $\mathcal{L}(1)$ and its refinement

5.2 Description of the ordering algorithm

The algorithm in section 5.1 gives a maximal tree partitioning on a connected graph. It is interesting to note that the order in which the partition members Y are formed defines a monotone ordering (see section 4.3) on the resulting quotient tree. A node ordering can then be obtained by numbering the nodes in each partition member consecutively, where the members are arranged in the same order as they are formed. The ordering algorithm will be complete if we specify how to choose the root r for the level structure $\mathcal{L}(r)$ and how the nodes within each partition member are to be numbered. In this section, we complete these specifications and the overall ordering algorithm will be hereafter referred to as the refined quotient tree (RQT) algorithm.

The comment at the beginning of section 5.1 suggests that the rooted level structure $\mathcal{L}(r)$ to be refined should have as many levels as possible. In this case, r is a peripheral node (see section 4.1) with the length of its rooted structure the same as the diameter of the graph. But in general, it is time consuming to find peripheral nodes.

In [11], Gibbs, Poole and Stockmeyer introduce the notion of pseudo-peripheral nodes. A node x is pseudo-peripheral if its eccentricity $\ell(x)$ is close to the diameter of the graph. They have also provided an efficient way of finding such nodes. Their approach is based on the following observation.

Let $\mathcal{L}(x) = \{L_0(x), L_2(x), \ldots, L_{\ell(x)}(x)\}$ be the rooted level structure at x. Then for any $y \in L_{\ell(x)}(x)$, $\ell(x) \le \ell(y)$.

In view of the efficiency of their algorithm, it becomes appropriate for our purposes to consider level structures rooted at pseudo-peripheral nodes. A modified version of their algorithm is used and it is described below. For discussion on merits of the modification, refer to [9].

Step 1: Choose an arbitrary node r.

Step 2: Generate the rooted level structure at r:

$$\mathcal{L}(r) = \{L_0(r), L_1(r), \ldots, L_{\ell(r)}(r)\}.$$

Step 3: Find all the connected components in $L_{\ell(r)}(r)$.

Step 4: For each component C in $L_{\ell(r)}(r)$, find a node x of minimum degree and generate its rooted level structure $\mathcal{L}(x)$. If $\ell(x) > \ell(r)$, put $r \leftarrow x$ and go to step 3'.

Step 5: r is a pseudo-peripheral node.

Finally, we address the problem of numbering nodes within each partition member. In order to find an appropriate numbering scheme, we consider the implicit storage scheme in section 2. In the storage scheme, the diagonal blocks are stored in the profile (envelope) form. Since each diagonal block corresponds to a partition member, it becomes clear that the nodes in each member should be ordered by some profile reduction scheme.

The reverse Cuthill-McKee (RCM) algorithm [3] is known to be a simple and yet effective ordering scheme for profile reduction. It is most suitable for our purpose. For completeness, we include a description of this scheme for a connected graph G.

Step 1: Determine a pseudo-peripheral node r and assign it to x_1.

Step 2: For $i = 1,2,\ldots,N$, find all the unnumbered neighbors of the node x_i and number them in increasing order of degree.

Step 3: The RCM ordering is given by: $x_N, x_{N-1}, \ldots, x_1$.

Extension of the algorithm for disconnected graphs is straightforward. However, we do not simply apply the RCM algorithm on each partition member; some extra care is taken. Consider a partition member $Y \in P$, where $Y \subset L_k$ and P is the refined partitioning of $\mathcal{L}(r) = \{L_0, L_1, \ldots, L_{\ell(r)}\}$. Let S be the set $\{y \in Y | Adj(y) \cap L_{\ell+1} \neq \phi\}$. Note that this is the same set S at the execution of step 2 in the algorithm of section 5.1.

Our internal numbering strategy follows. Nodes in the subgraph $G(Y\backslash S)$ are first numbered using the RCM ordering scheme. Note that the subgraph $G(Y\backslash S)$ may be disconnected. The nodes in S are then numbered arbitrarily. This part can be incorporated into the algorithm of section 5.1 simply by replacing step 3:

Step 3': (Internal numbering of partition member): Number $G(Y\backslash S)$ by the RCM scheme and then number nodes in S in an arbitrary order.

To illustrate the effect of this internal ordering, we consider the finite element mesh M in Figure 5.3. Here, nodes in each triangle are assumed to be pairwise adjacent in the corresponding graph. The rooted level structure at the node 1,

$$\mathcal{L}(1) = \{L_0, L_1, L_2, L_3, L_4\}$$

has 5 levels. No refinement can be done on the levels so that $P = \{Y_1, Y_2, Y_3, Y_4, Y_5\}$, where $Y_i = L_{5-i}$. The new ordering in Figure 5.4 is obtained using the internal numbering strategy on the Y's; the correspondingly permuted matrix is also given.

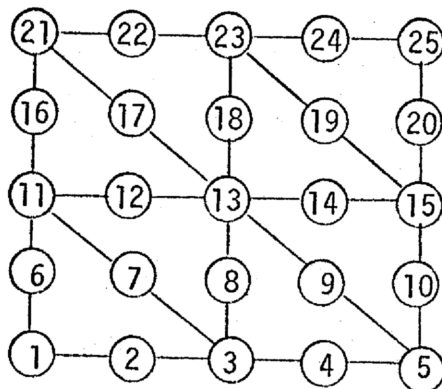


Figure 5.3 A finite element mesh M

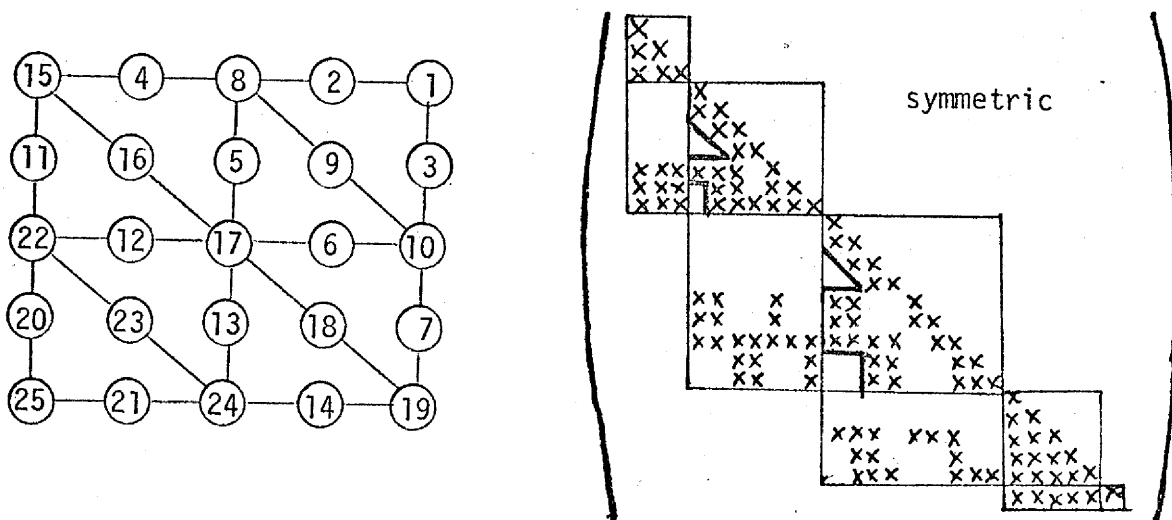


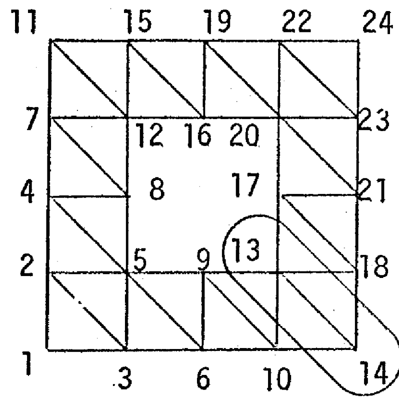
Figure 5.4 Reordered mesh and its corresponding matrix structure

§6 A Hole Removal Algorithm

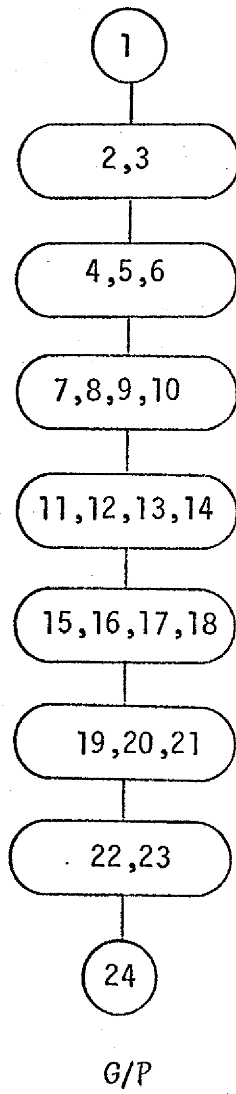
The refined quotient tree algorithm in section 5 produces an ordering which is highly appropriate for our implicit storage scheme. In addition, we shall see later that the amount of computation required using this ordering solution scheme is fully competitive with standard band or envelope methods. As we mentioned, no recursion is needed in the execution of "lowersolve" and "uppersolve".

However, if we are prepared to accept one level of recursion in the execution of our algorithms, it is often possible to obtain a substantial reduction in storage requirements for our implicit storage scheme when the mesh has one or more holes. The second example in section 3 was an illustration of this, and we now consider a similar example in graph theoretic terms. Suppose the quotient tree G/P in Figure 6.1 was produced by our RQT algorithm, where $P = \{\{1\},\{2,3\},\ldots, \{22,23\},\{24\}\} = \{Y_1,Y_2,\ldots,Y_9\}$. Now because M has a hole, some of the subgraphs $G(Y_i)$ are disconnected. Partitioning these Y_i corresponding to the connected components of $G(Y_i)$ yields a partitioning P' and a new quotient graph G/P', which is no longer a tree because it has a cycle. Thus, we have achieved a finer partitioning, which is what we want because it tends to reduce storage requirements, but we no longer have a tree, which means we must admit some recursion in our solution scheme.

Let $G = (X,E)$ be a given connected graph. Our purpose is to determine a partition $\{Y_1,Y_2,\ldots,Y_p,T\}$ where the quotient graph <u>without</u> T and its incident edges forms a tree. The set T is sometimes termed the <u>tearing set</u>. Obviously, we want to get a partition with as many members as



Mesh M



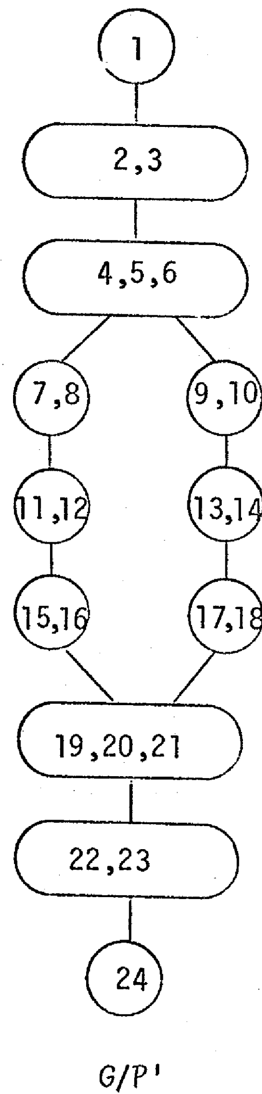


Figure 6.1 An example of a refinement of a quotient tree

possible and at the same time has a small tearing set. In terms of a finite element mesh, this corresponds to finding a set of nodes whose removal removes all the holes in the mesh. In this section, we give an algorithm for this purpose.

The algorithm builds up a certain quotient graph and in the process it also searches for cycles in this quotient graph. In this way, a subset T of nodes can be determined to break up all the cycles. Care is taken in the scheme to find a small subset T. The algorithm may be regarded as a depth first search on the resulting quotient graph, and it is similar to the cycle determination algorithm by Weinblatt [17].

We now give a formal description of the scheme. It maintains two stacks of node subsets $\mathcal{B}$ and $\mathcal{K}$. The stack $\mathcal{B}$ is used to store the currently formed quotient members, while the contents of $\mathcal{K}$ help to detect cycles. For convenience, we define the stack subsets

$$B = \{Y \subset X | Y \in \mathcal{B}\}$$

$$K = \{S \subset X | S \in \mathcal{K}\},$$

and we let T be the set of tearing nodes generated by the algorithm.

Step 0 (Initialization): $\mathcal{B} \leftarrow \phi$, $\mathcal{K} \leftarrow \phi$ and $T \leftarrow \phi$. Find a pseudo-peripheral node r and put $S \leftarrow \{r\}$.

Step 1 (Next block): Find a connected component Y in the subgraph S. If $Y \neq S$, push $S \backslash Y$ into the stack $\mathcal{K}$.

Step 2 (Advance block): Push Y into the stack $\mathcal{B}$ and determine $S = Adj(Y) \backslash (B \cup T)$. If $S = \phi$, go to step 4.

Step 3 (Cycle detection): If $S \cap K = \phi$, go to step 1. Otherwise a cycle is found.

In this cycle, find the smallest and most recently formed subset Y^* from the stack $\mathcal{B}$. Put $T \leftarrow T \cup Y^*$ and pop all the subsets formed after Y^* from $\mathcal{B}$ and K.

Step 4 (Pop stack K): If K is empty, stop. Otherwise, pop the top subset from K as the new S, and branch back to step 1.

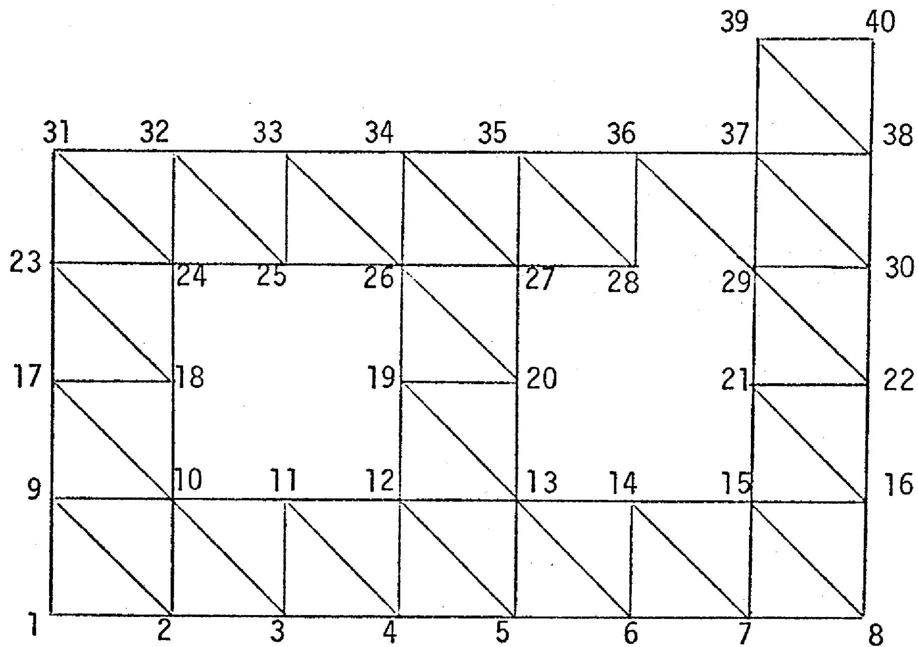


Figure 6.2 A 40-node graph with two holes

Consider the graph in Figure 6.2. Suppose node 1 is the starting node. Figures 6.3 and 6.4 show some important steps in the algorithm. Subsets within solid and dotted lines are the current blocks in the stack $\mathcal{B}$ and K respectively, while nodes in double lines belong to the set T.

In Figure 6.3(i), $Y = \{40\}$ and $S = \phi$ so that step 4 is executed and $\{15,16\}$ is popped from the stack K as the new S. In Figure 6.3(ii), $S = \{6,13\}$ has a non-empty intersection with the set of nodes in K. As a result, $Y^* = \{36\}$ is included into T. Again, another cycle is detected in Figure 6.4(i) so that $\{12,13\}$ is removed. Figure 6.4(ii) shows the structure of the final quotient graph, with $T = \{12,13,36\}$.

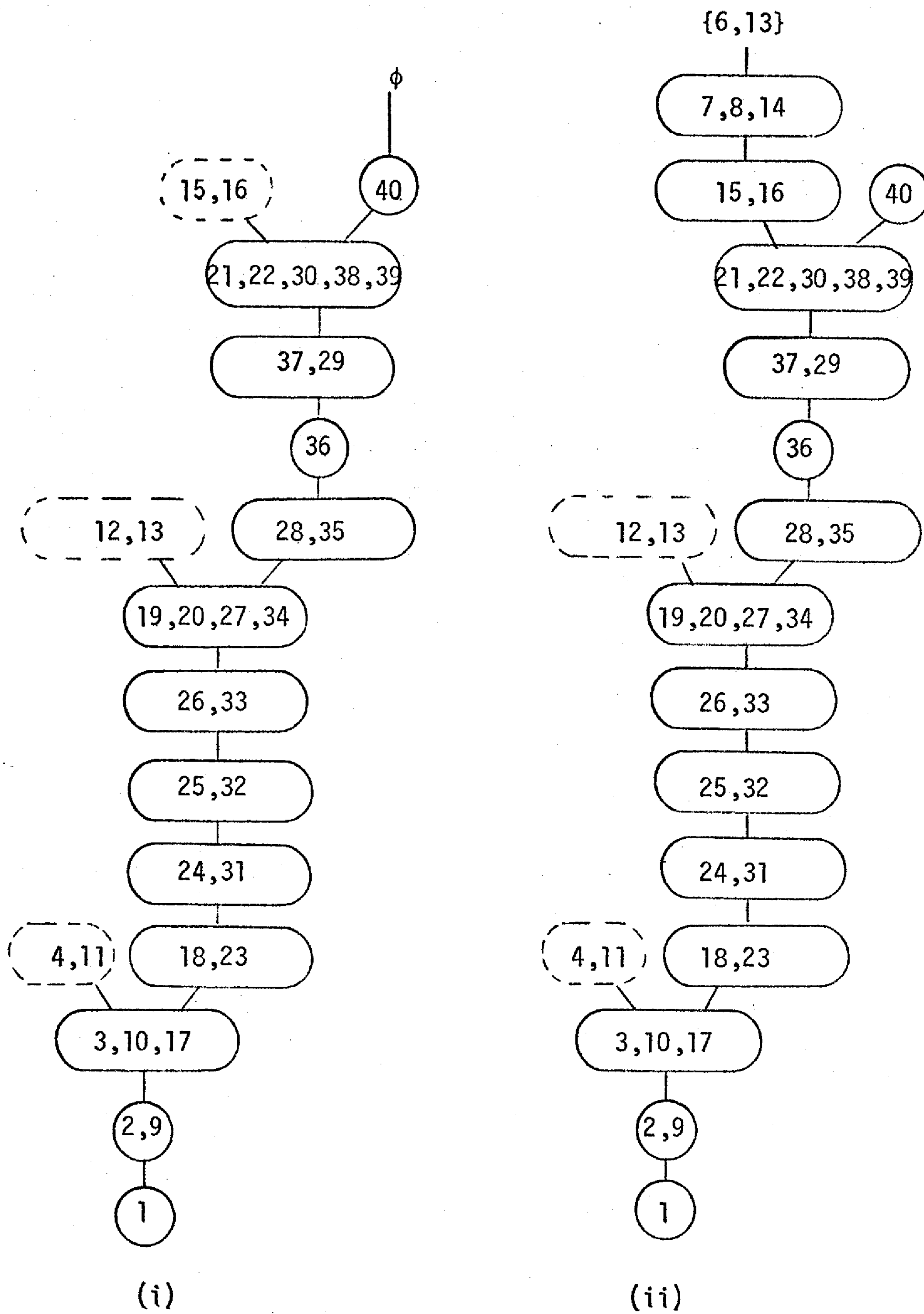


Figure 6.3 Stages in the RH algorithm

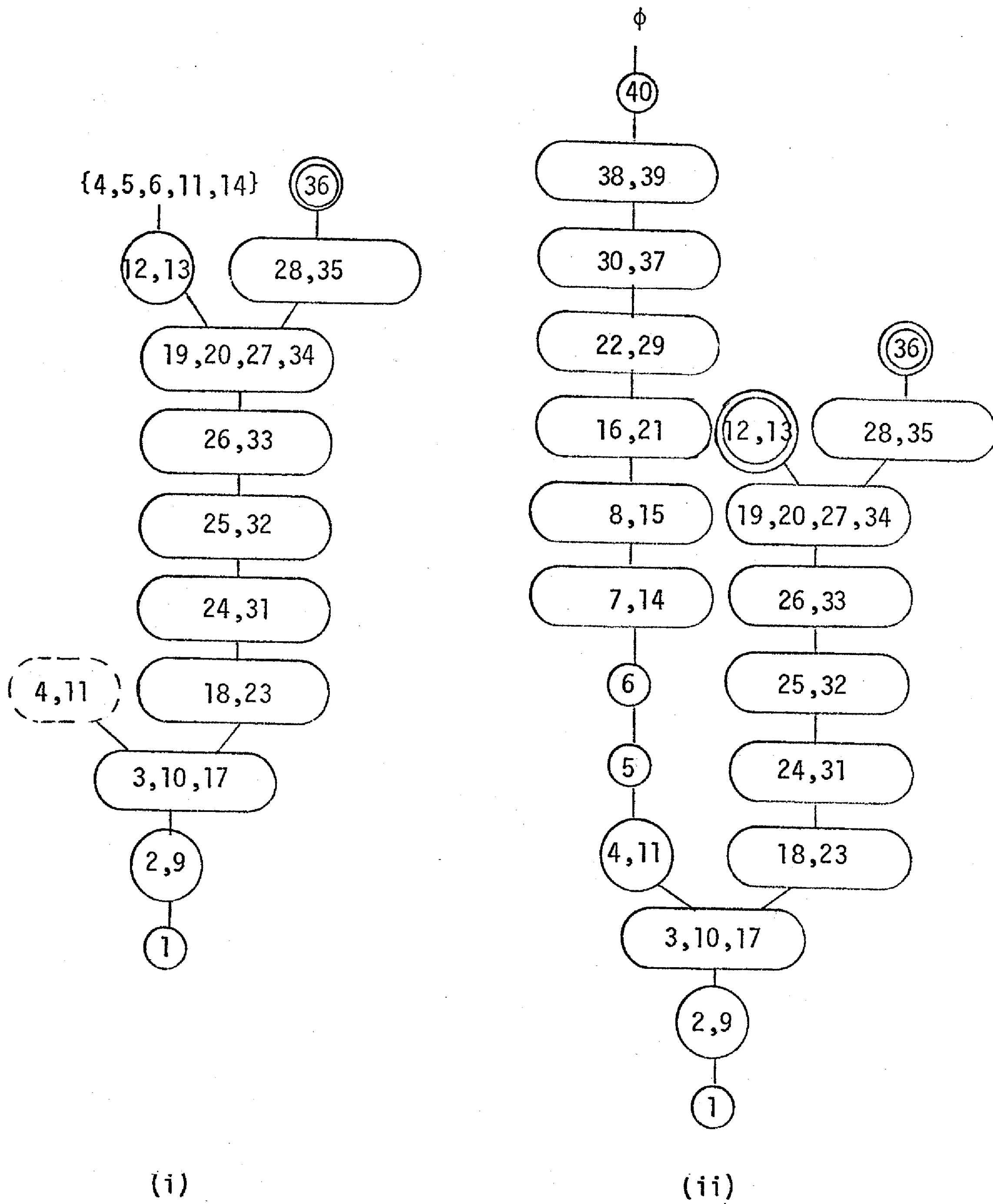
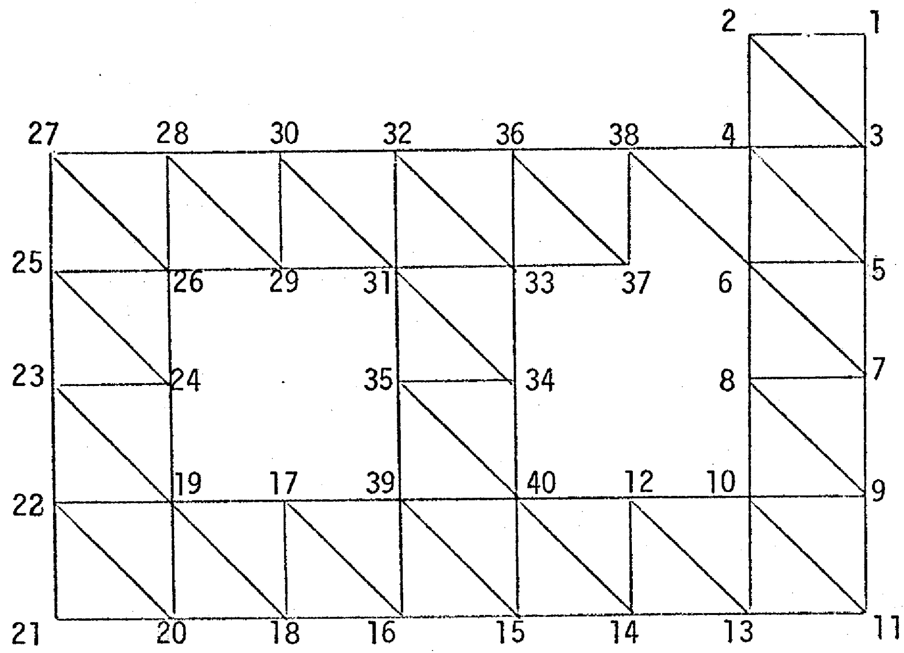
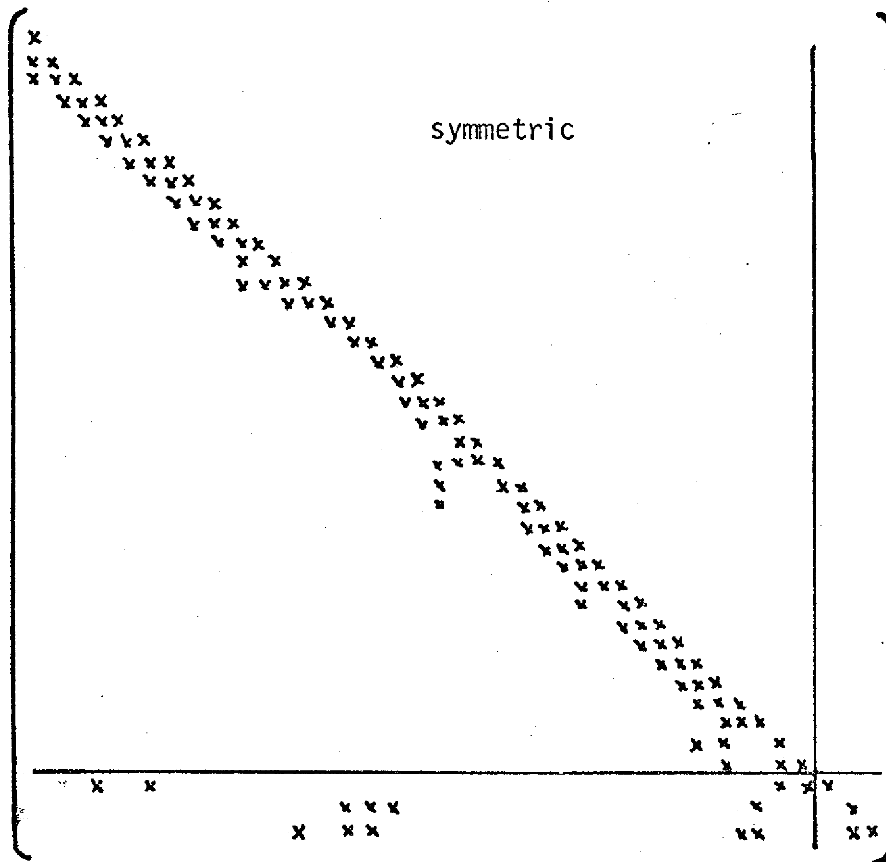


Figure 6.4 Stages in the RH algorithm


2
1
27
28
30
32
36
38
4
3
25
26
29
31
33
37
6
5
23
24
35
34
8
7
19
17
39
40
12
10
22
9
21
20
18
16
15
14
13
11


symmetric

§7 Summary of the Algorithm and Some Experiments

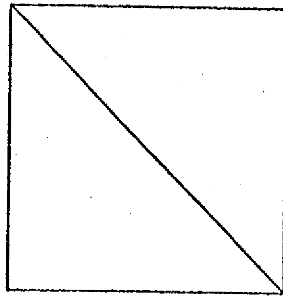
Sections 5 and 6 contain descriptions of two algorithms which we combine to generate an ordering and partitioning as follows, where $G = (X,E)$ is the graph of our given finite element matrix.

1. Using the hole removal algorithm described in section 6, find a subset $T \subset X$ such that the section graph $G(X\backslash T) = G(X_T) = (X_T, E(X_T))$ has no holes.

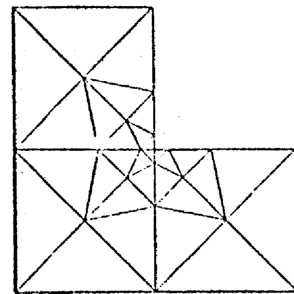
2. Apply the RQT algorithm to $G(X_T)$, yielding a partitioning $\{Z_1, Z_2, \ldots, Z_s\}$ of X_T, and an ordering of X_T. This provides a labelling of the first $|X_T|$ nodes of X; now label the nodes of T from $|X_T|+1$ to N in any order, completing the labelling of X.

We use our implicit storage scheme to store L, using the partitioning $\{Z_1, Z_2, \ldots, Z_s, T\}$. In what follows, ENVSLV refers to a collection of factorization and triangular solution subroutines which implement Cholesky's method for solving a positive definite system, using Jenning's storage scheme. This package is an integral part of our code TRESLV, which implements the algorithms of section 2 <u>assuming that the underlying quotient graph is a tree</u>, such as provided by the algorithm RQT. Finally, BLKSLV implements the algorithms of section 2 assuming that the matrix with the last block-row and block-column deleted corresponds to a quotient tree; thus, BLKSLV is appropriate for the partitioning/ordering provided by the RH-RQT combination of algorithms. For obvious reasons, a major part of BLKSLV consists of the code TRESLV.

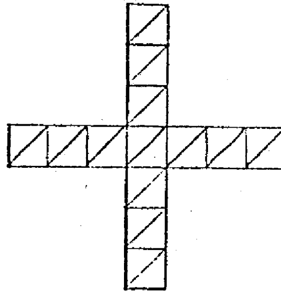
We now turn to our numerical experiments. Our test problems consist of 9 two-parameter mesh problems typical of those arising in structural analyses. The basic meshes, shown in Figure 7.1, are subdivided



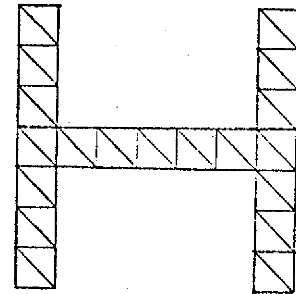
a) Square



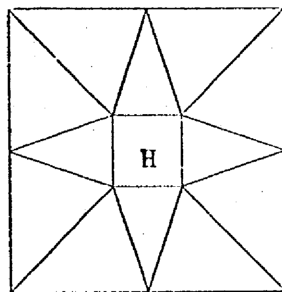
b) Graded L



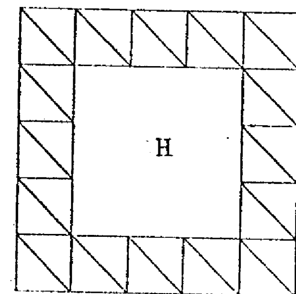
c) + shaped domain



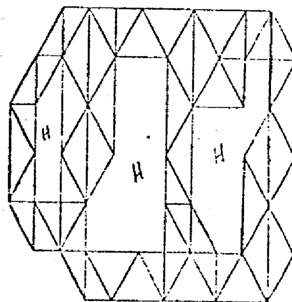
d) H-shaped domain



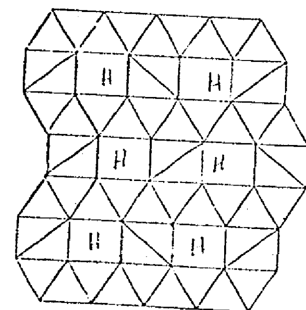
e) Hollow square (small hole)



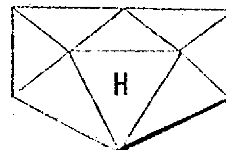
f) Hollow square (large hole)



g) 3 hole problem



h) 6 hole problem



i) Pinched hole problem

Figure 7.1 Mesh problems with $\alpha=1$

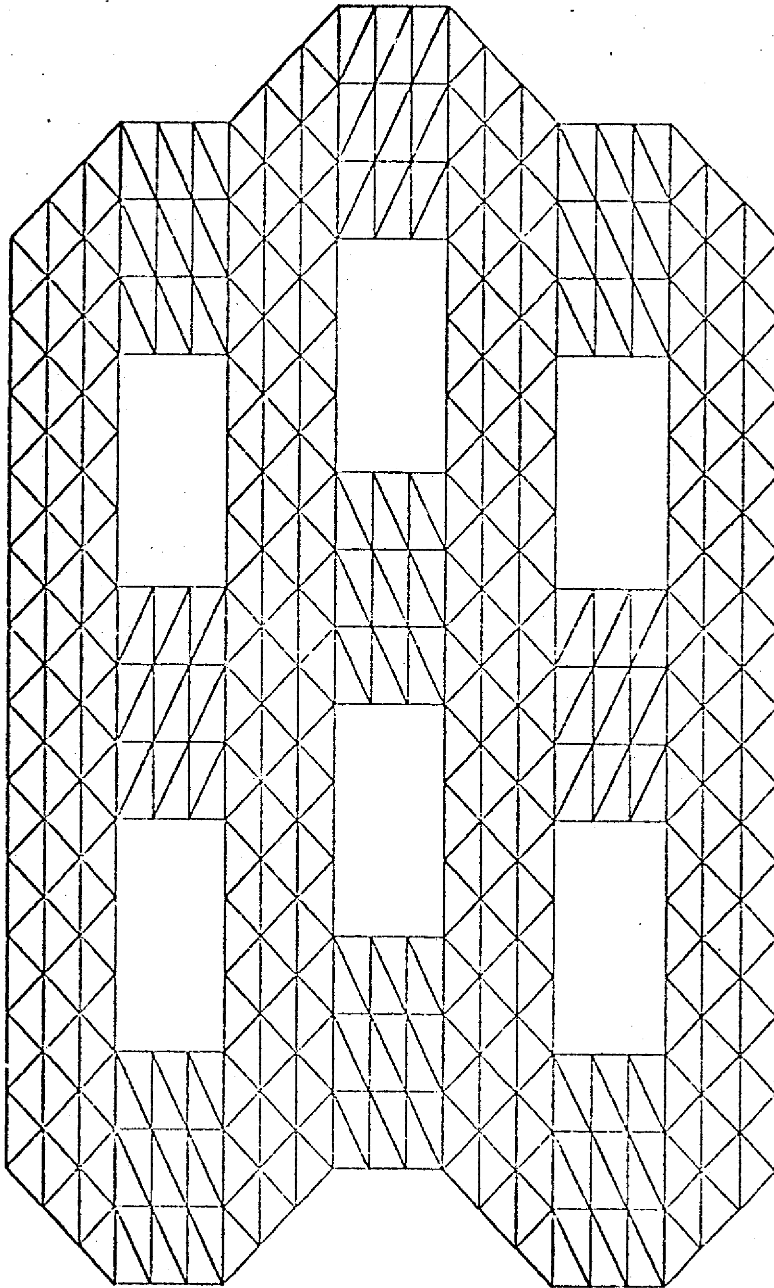


Figure 7.2 The six-hole problem with $\alpha=3$

Test Example	Domain	No. of Divisions α	No. of Equations N
a	Square	32	1089
b	Graded L	8	1009
c	+ shaped	9	1180
d	H shaped	8	1377
e	Square (small hole)	12	936
f	Square (large hole)	9	1440
g	3 hole problem	6	1138
h	6 hole problem	6	1141
i	Pinched hole	19	1349

Table 7.1 Test Examples with degree $\mu = 1$.

by a factor α in the obvious way, yielding a mesh having α^2 times as many triangles as the original mesh, as shown in Figure 7.2 for the six-hole mesh with $\alpha = 3$. The second parameter μ governs the distribution of nodes on the mesh, and corresponds essentially to the degree of certain piecewise polynomials used in finite element applications [6]. For $\mu \geq 1$, there is 1 node at each triangle vertex, μ-1 nodes along each edge, and $(\mu-1)(\mu-2)/2$ nodes in the interior of each triangle.

The reverse Cuthill-McKee algorithm (RCM) [6] is designed to produce an ordering yielding a small profile. This algorithm has been found to be very effective for finite element problems [6,11], and is widely used, in combination with Jenning's storage scheme, in many industrial applications. Considerable additional testing on our test problems and others by the author's led us to choose the RCM-ENVSLV combination as a benchmark for our experiments [8].

We are interested in answers to, or evidence bearing on, the following issues.

1) Which of the two ways of performing the factorization (F_1 or F_2) appears to be more efficient for our set of problems?

2) How effective is the RQT-TRESLV combination compared to the benchmark RCM-ENVSLV pair?

3) How does RH-RQT-BLKSLV compare to RQT-TRESLV? (That is, is the removal of holes worthwhile?) Obviously, this question only makes sense for problems with holes.

4) How does RQT-TRESLV compare to more sophisticated optimal or near-optimal schemes which attempt to minimize fill?

In the tables that follow operations mean multiplications and divisions. Primary storage refers the actual number of matrix components of L (or A) stored, while overhead refers to the associated pointers etc. required to store those components. In all our codes, the same number of bits are used to store data as pointers; on machines with a large word size it would make sense to pack two or more pointers per word, and this would make the methods having large overhead relatively more attractive. Temporary storage refers to storage needed in the course of the computation, but not involved in the actual data structures for L. (See section 2.3) The times reported are in seconds on an IBM 360/75. The programming language used was H-level Fortran with the optimizer turned on.

We begin by investigating the relative merit of the F_1 and F_2 versions of TRESLV. We solved the plain square for $1 \le \mu \le 3$ and values of α adjusted so that in all cases the number of equations was equal to 961. Recall from section 2.3 that primary and overhead storage will be the same whether we use the F_1 or F_2 versions (as will the ordering and solution times), but temporary storage and factorization times in general will be different. The F_2 version requires a vector of length about N, while the F_1 version requires temporary space for the largest off-diagonal block W_k. Since the sizes of these blocks increase with increasing μ we expect the F_1 version to require more temporary storage than the F_2 version. However, as μ increases for these problems, a fraction (equal to about $(\mu\text{-}1/\mu)$) of the columns of each W_k are null, as shown in the example in Figure 5.4 where $\mu = 2$. Since the ordering algorithm automatically numbers the nodes so that the null columns of each W_k appear first, they are easy to exploit and our F_1 version does so. Table 7.2 summarizes the relevant information on these experiments.

μ	FACTORIZATION				STORAGE					
	OPERATIONS		TIME		TOTAL			TEMPORARY		
	F_1	F_2	F_1	F_2	F_1 †	F_1^*	F_2	F_1 †	F_1^*	F_2
1	2.717	2.916	3.57	4.71	18106	18106	18106	961	961	960
2	4.948	5.726	5.05	6.72	21225	23025	23025	1860	3660	960
3	5.516	6.258	4.95	6.30	24798	29994	23030	2728	7920	960
	$x(10^5)$	$(x10^5)$								

† null columns of W_k not stored

* null columns of W_k stored

Table 7.2 Factorization and storage statistics for the F_1 and F_2 versions of TRESLV for the plain square problem with increasing μ, and α adjusted so that N = 961 for all problems.

Several aspects of Table 7.2 are noteworthy. First, the exploitation of the null columns of the off-diagonal blocks yields a substantial reduction in temporary storage requirements. However, this null column phenomenon is entirely due to the existence of edge and interior nodes in our finite element meshes. In other finite element problems where higher degree bases are used, all variables may be associated with vertex nodes, with several variables per node. In such cases, the off-diagonal blocks will be relatively large but there will usually be no null columns; then a comparison of the F_1^* column with the F_2 column in Table 7.2 is more indicative of the relative storage requirements of the two versions of TRESLV as the degree of the underlying basis increases. Note also that we appear to pay a slight cost in execution time by using the F_2 version.

The execution times for $\mu = 3$ are actually less than the corresponding execution times for $\mu = 2$, even though the operation counts are higher. This appears to be due to the fact that the partition is coarser for $\mu = 3$, and substantially fewer subroutine calls and other execution overhead is incurred because the blocks are larger.

Our next set of experiments were designed to investigate the relative effectiveness of the RQT-TRESLV combination compared to the benchmark RCM-ENVSLV pair. We ran both codes on the nine test problems a) through i) described in Figure 7.1 and Table 7.1. Since the problems were constructed with $\mu = 1$, in view of the results in Table 7.2, we used the F_1 version of TRESLV. The results are summarized in Tables 7.3 and 7.4.

TEST EXAMPLE	ORDER TIME		STORAGE			
	RCM	RQT	ENVSLV		TRESLV	
			TOTAL	OVERHEAD	TOTAL	OVERHEAD
a	.31	.28	27744	1092	21243	4358
b	.38	.34	28285	1012	20895	4030
c	.39	.51	28276	1183	16737	4740
d	.41	.37	24449	1380	18389	5494
e	.26	.24	24638	939	18612	3724
f	.37	.38	39061	1443	30142	5674
g	.77	.39	26908	1141	20701	4464
h	.71	.32	36038	1144	26005	4452
i	.35	.45	42475	1352	30387	5362

Table 7.3 Ordering times and storage requirements for RCM-ENVSLV and RQT-TRESLV (F_1) for the nine test problems of Table 7.1.

TEST PROBLEM	FACTORIZATION				SOLUTION			
	OPERATIONS		TIME		OPERATIONS		TIME	
	ENVSLV	TRESLV	ENVSLV	TRESLV	ENVSLV	TRESLV	ENVSLV	TRESLV
a	3.446	3.461	3.17	4.15	5.111	5.434	.41	.62
b	3.749	3.810	3.39	4.25	5.251	5.495	.41	.60
c	3.190	1.160	2.83	2.01	5.181	3.377	.42	.55
d	1.890	1.001	2.00	2.02	4.336	3.478	.37	.62
e	3.018	3.052	2.80	3.52	4.551	4.827	.37	.57
f	5.377	5.342	4.83	6.04	7.234	7.451	.60	.87
g	2.888	2.925	2.81	3.55	4.924	5.135	.41	.63
h	5.623	5.938	5.10	6.16	6.749	6.929	.57	.77
i	6.495	6.393	6.20	6.80	7.953	8.307	.67	.87
	$(\times 10^5)$	$(\times 10^5)$			$(\times 10^4)$	$(\times 10^4)$		

Table 7.4 Execution times and operation counts for ENVSLV and TRESLV for the nine test problems of Table 7.1.

The results of Table 7.3 illustrate the savings in storage achieved by the RQT-TRESLV combination compared to the standard scheme. As expected, the new scheme is most effective on problem c and d, which have appendages. We included the overhead component in the total storage requirements for the solvers for the following reason. On machines with a large word size it is sensible and often convenient to pack two or three pointers to a word. Since a substantially larger fraction of the storage is overhead in TRESLV, packing of pointers would increase the storage advantage of our quotient tree approach even more, compared to the standard scheme.

The results of Table 7.4 indicate that one pays a modest penalty in execution time by using RQT-TRESLV rather than RCM-ENVSLV, unless the problem has many appendages, such as problems c) and d). It is interesting to note that the increased execution time is due to data structure complexity rather than increased operation counts. (As is often the case in sparse matrix computations, access to the data plays a crucial role in the effectiveness of an algorithm. Operation counts alone, without careful implementation and experimentation, can be highly misleading and should be viewed with skepticism.) This phenomenon is also illustrated in Tables 7.6-7.8.

Our next set of experiments was designed to investigate the effectiveness of our hole removal strategy and the associated BLKSLV code which implements a one level version of the recursive solution scheme described in section 2. The RH-RQT-BLKSLV code was run on problems e through i, and the relevant results are compiled in Table 7.5 and 7.6. Generally speaking, they indicate that the hole removal can reduce storage requirements, but at substantially increased execution cost. Whether such an exchange pays depends on various circumstances which we discuss in section 8.

TEST EXAMPLE	ORDER TIME		Nodes Removed by	STORAGE			
				TRESLV		BLKSLV	
	RQT	RH-RQT	RH	TOTAL	OVERHEAD	TOTAL	OVERHEAD
e	.24	.70	13	18612	3724	16087	3727
f	.38	1.04	10	30142	5674	22362	5745
g	.39	1.17	21	20701	4464	16424	4549
h	.32	1.26	44	26005	4452	16750	4570
i	.45	1.14	20	30387	5362	29977	5346

Table 7.5 Ordering and storage results for RQT-TRESLV and RH-RQT-BLKSLV, for the problems with holes.

TEST PROBLEM	FACTORIZATION				SOLUTION			
	OPERATIONS		TIME		OPERATIONS		TIME	
	TRESLV	BLKSLV	TRESLV	BLKSLV	TRESLV	BLKSLV	TRESLV	BLKSLV
e	3.052	5.726	3.52	8.33	4.827	6.823	.57	1.01
f	5.342	5.710	6.04	9.24	7.451	8.537	.87	1.40
g	2.925	5.490	3.55	10.04	5.135	5.683	.63	1.01
h	5.938	9.775	6.16	17.95	6.929	5.472	.77	1.00
i	6.393	7.309	6.80	8.50	8.307	15.375	.87	1.78
	$(\times 10^5)$	$(\times 10^5)$			$(\times 10^4)$	$(\times 10^4)$		

Table 7.6 Execution times and operation counts for RQT-TRESLV and RH-RQT-BLKSLV, for the problems with holes.

Finally, we wish to show that these schemes do indeed occupy a "middle ground" between the standard band oriented schemes, as exemplified by our RCM-ENVSLV package, and the more sophisticated optimal or near-optimal ordering strategies. As a representative of this latter approach we include a few results obtained by using a slightly revised version of our automatic nested dissection ordering algorithm and solver (ND-NDSLV), described in [10]. Test example b, the graded-L, was used with $\mu = 1$ and subdivision factors $\alpha = 4,5,\ldots,12$, thus providing a range of similar problems of increasing size. We feel this choice of problem provides a fair comparison, since it has no appendages which would aid the RQT-TRESLV scheme. The results are summarized in Tables 7.7 and 7.8.

	ORDER TIME			STORAGE					
				TOTAL			OVERHEAD		
N	RCM	RQT	ND	ENVSLV	TRESLV	NDSLV	ENVSLV	TRESLV	NDSLV
265	.12	.10	.36	4279	3925	5433	269	1083	1210
406	.19	.15	.61	7764	6611	9135	410	1652	1881
577	.27	.22	.90	12748	10246	13845	581	2341	2720
778	.36	.29	1.27	19497	14963	19683	782	3150	3511
1009	.49	.41	1.72	28277	20895	26691	1013	4079	4572
1270	.60	.50	2.25	39354	28175	35252	1274	5128	5913
1561	.73	.58	2.89	52994	36936	44957	1565	6297	7366
1882	.88	.68	3.54	69463	47311	55924	1886	7586	8861
2233	1.04	.81	4.35	89027	59433	68244	2237	8995	10622

Table 7.7 Comparison of RCM, RQT, and ND ordering times and the storage requirements of the corresponding solvers.

N	FACTORIZATION						SOLUTION					
	OPERATIONS			TIME			OPERATIONS			TIME		
	ENVSLV	TRESLV	NDSLV	ENVSLV	TRESLV	NDSLV	ENVSLV	TRESLV	NDSLV	ENVSLV	TRESLV	NDSLV
265	.297	.306	.330	.34	.52	.69	.748	.815	.738	.06	.13	.11
406	.662	.677	.685	.70	1.04	1.28	1.389	1.489	1.288	.12	.22	.18
577	1.288	1.315	1.201	1.27	1.77	2.05	2.318	2.459	1.994	.19	.31	.28
778	2.278	2.316	1.988	2.14	3.06	3.09	3.587	3.776	2.923	.29	.49	.38
1009	3.749	3.809	3.003	3.38	4.96	4.35	5.251	5.495	4.020	.41	.64	.52
1270	5.837	5.917	4.404	5.10	6.83	6.05	7.362	7.667	5.360	.58	.88	.68
1561	8.695	8.807	6.113	7.38	9.83	8.07	9.973	10.348	6.893	.77	1.17	.87
1882	12.490	12.633	8.295	10.40	13.52	10.47	13.139	13.590	8.659	1.01	1.47	1.06
2233	-	17.600	10.837	-	18.25	13.33	-	17.445	10.631	-	1.81	1.28
	$(\times 10^5)$	$(\times 10^5)$	$(\times 10^5)$				$(\times 10^4)$	$(\times 10^4)$	$(\times 10^4)$			

Table 7.8 Comparison of operation counts and execution times for the envelope, quotient tree, and nested dissection solvers.

§8 Concluding Remarks

We now provide some observations which we feel shed some light on the questions posed in section 7. There are several important issues which make definite answers to any of these questions virtually impossible. These are as follows.

a) Is the given matrix problem to be solved only once? If so, a comparison between two ordering/solution packages should include the cost of producing the ordering and initializing the data structures. On the other hand, if many problems having the same zero-nonzero structure must be solved, it may be reasonable to ignore the ordering and initialization cost in the comparison.

b) Is there more than one right hand side involved in the matrix problem? In the solution of some mildly nonlinear and time dependent problems, many systems having the same coefficient matrix must be solved. In these situations, the cost of solving the problem, given the factorization, may be the primary factor determining the merit of the method.

c) How does one measure cost? Obviously the real cost C is some function of execution time T and storage used S. Since the use of RQT-TRESLV rather than RCM-ENVSLV may simply increase T and decrease S, the function C(S,T) will be fundamental in determining which method of solving the problem results in the least cost. In this connection, we contend that storage reduction should often be regarded as at least as important as the reduction of execution time. Since computer memory continues to be a relatively expensive hardware component, computing center charging algorithms are usually designed to discourage large main

storage demands. A typical charging function C(S,T)is of the form Txp(S), where p is a polynomial in S whose degree d is sometimes greater than 1.In this case,for large S, if method 1 uses T_1 seconds and S_1 storage, while method 2 uses T_2 seconds and S_2 storage, as a rough comparison of the methods we could use the ratio $S_1T_1^d/S_2T_2^d$. Another important point is that a reduced storage requirement may allow one to solve a problem in main storage rather than using auxiliary storage. The value of this is hard to assess, but can be very substantial if the use of auxiliary storage is not carefully implemented.

With these issues in mind, we now make some observations which are suggested by Tables 7.2-7.7.

1) With regard to the variations F_1 and F_2 of BLKSLV (see section 2), for our examples, the F_1 version was significantly more efficient in terms of execution time, and for the low order elements ($\mu = 1$), required about the same storage. However, for $\mu > 1$, the F_2 version in some cases may require substantially less storage.

2) The use of RH-RQT-BLKSLV rather than RQT-TRESLV typically increases computation substantially and decreases storage. Whether this amounts to a net improvement depends on the issues raised above.

3) The same remarks as 2) apply when we compare RCM-ENVSLV with either RQT-TRESLV or RH-RQT-BLKSLV. In addition, as we pointed out in section 2, on machines with a large word size where it would make sense to pack

pointers two or three to a word, the total storage requirements for RQT-TRESLV and RH-RQT-BLKSLV would drop substantially, while that for RCM-ENVSLV would remain about the same.

4) Finally, the results of Table 7.7 and 7.8 show that there are problem sizes and circumstances where the RQT-BLKSLV combination could be an attractive alternative to either the standard band-oriented schemes or the much more sophisticated near-optimal ordering schemes. When storage is limited, and/or when auxiliary storage is difficult to use or not available, the RQT-TRESLV and RH-RQT-BLKSLV combinations may be quite attractive, even though their execution times may be larger than their competitors.

§9 References

[1] I. Arany, W.F. Smyth and L. Szoda, "An Improved Method for Reducing the Bandwidth of Sparse Symmetric Matrices", in Informacion Processing 71: Proceedings of IFIP Congress, North-Holland, Amsterdam, 1972.

[2] C. Berge, The Theory of Graphs and its Applications, John Wiley & Sons Inc., New York, 1962.

[3] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices", Proc. 24th Nat. Conf., Assoc. Comput. Mach., ACM Publ. P-69, 1122 Ave. of the Americas, New York, N.Y., 1969.

[4] Alan George, "Nested Dissection of a Regular Finite Element Mesh", SIAM J. Numer. Anal., 10 (1973), pp.345-363.

[5] Alan George, "On Block Elimination for Sparse Linear Systems", SIAM J. Numer. Anal., 11 (1974), pp.585-603.

[6] Alan George, "Computer Implementation of the Finite Element Method", Stanford University Technical Report STAN-CS-208, 1971.

[7] Alan George, "Numerical Experiments Using Dissection Methods to Solve n by n Grid Problems", SIAM J. Numer. Anal., to appear.

[8] Alan George and Joseph W.H. Liu, "An Automatic Partitioning and Solution Scheme for Solving Large Sparse Positive Definite Systems of Linear Algebraic Equations", Research Rept. CS-75-17, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, June 1975.

[9] Alan George and Joseph W.H. Liu, "An Implementation of a Pseudo-peripheral Node Finder", Research Rept. CS-76-44, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1976.

[10] Alan George and Joseph W.H. Liu, "An Algorithm for Automatic Nested Dissection and its Application to General Finite Element Problems", Report CS-76-38, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada (1976).

[11] N.E. Gibbs, W.G. Poole, and P.K. Stockmeyer, "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix", SIAM J. Numer. Anal., to appear.

[12] F.G. Gustavson, "Some Basic Techniques for Solving Sparse Systems of Equations", Sparse Matrices and their Applications, D.J. Rose and R.A. Willoughby eds., Plenum Press, New York, 1972.

[13] A. Jennings, "A Compact Storage Scheme for the Solution of Simultaneous Equations", Comput. J., 9 (1966), pp.281-285.

[14] S.V. Parter, "The Use of Linear Graphs in Gauss Elimination", SIAM Rev., 3 (1961), pp.364-369.

[15] D.J. Rose, "A Graph-theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations", in Graph Theory and Computing, edited by R.C. Read, Academic Press, New York, 1972.

[16] K.L. Stewart and J. Baty, "Dissection of Structures", J. Struct. Div., ASCE, Proc. paper No.4665, (1966), pp.75-88.

[17] Weinblatt, H., "A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph", JACM 19 (1972), pp.43-56.

[18] James H. Wilkinson, The Algebraic Eigenvalue Problem, Clarendon Press, Oxford, 1965.

[19] O.C. Zienkiewicz, The Finite Element Method in Engineering Science, McGraw-Hill, London, 1970.