A FORTRAN PROGRAM FOR PROFILE AND WAVEFRONT REDUCTION

S. W. SLOAN

Department of Civil Engineering and Surveying, The University of Newcastle, N.S.W., 2308, Australia

SUMMARY

A FORTRAN 77 program for reducing the profile and wavefront of a sparse matrix with a symmetric structure is described. The implementation is based on an algorithm published previously by the Author and appears in response to a large number of enquiries for the source code. Extensive testing of the scheme suggests that its performance is consistently superior to that of the widely used reverse Cuthill-McKee and Gibbs-King methods. In addition to presenting a complete listing of the program, we also describe how to interface it with a typical finite element code. The scheme is especially useful in finite element analysis where it can be employed to derive efficient orderings for both profile and frontal solution schemes.

INTRODUCTION

Many problems in engineering require the solution of a set of linear equations of the form \( Ax = b \), where \( A \) is a sparse positive definite \( N \times N \) matrix with a symmetric structure, \( b \) is a prescribed vector of length \( N \) and \( x \) is a vector of length \( N \) which is sought. Two methods for solving these types of equations, which have found wide use in finite element analysis, are the profile and frontal solution schemes. In order to compute the solution efficiently, both of these algorithms require the equations to be processed in a certain order. For the profile method to be effective it is necessary to order the equations, which in finite element analysis corresponds to labelling the nodes, so that the sum of the row bandwidths is small. More formally, we define the sum of the row bandwidths to be equal to the profile, \( P \), according to

\[
P = \sum_{i=1}^{N} b_i
\]

where the row bandwidth, \( b_i \), is the difference between \( i + 1 \) and the column index of the first non-zero in row \( i \). In profile and frontal solution schemes the efficiency of an ordering is related to the number of equations which are active during each step of the factorization process. Formally, we define row \( j \) to be active during the elimination of column \( i \) if \( j \geq i \) and there exists \( a_{ik} = 0 \) with \( k < i \). Thus, at the \( i \)th stage of the factorization, the number of active equations is simply the number of rows of the profile of \( A \), ignoring the rows already eliminated, which intersect column \( i \). Letting \( f_i \) denote the number of equations which are active during the elimination of the variable \( x_i \), it follows from the symmetric structure of \( A \) that

\[
P = \sum_{i=1}^{N} f_i = \sum_{i=1}^{N} b_i
\]

In the finite element literature, the quantity \( f_i \) is known as the current wavefront or frontwidth.
Assuming that $N$ and the average value of $f_i$ are reasonably large, it may be shown that a complete profile or frontal factorization requires $O(N F^2)$ operations, where $F$ is the root-mean-square wavefront which is defined by

$$F = \left( \frac{1}{N} \sum_{i=1}^{N} f_i^2 \right)^{1/2}$$

When using the profile method we thus wish to minimize the profile and root-mean-square wavefront in order to minimize, respectively, the storage requirement and solution time. In finite element analysis this is achieved by a judicious labelling of the nodes. For the usual form of the frontal algorithm, however, the order of the nodes does not determine the efficiency of the solution process owing to the manner in which the assembly and elimination phases are interleaved. Rather than assembling the matrix $A$ in total before commencing the elimination procedure, the frontal scheme eliminates any equations that are fully summed immediately after they are assembled for each element in turn. Owing to this method of assembly and elimination it is the order of the elements that dictates the efficiency of a frontal solution. In practice, a good element ordering can be derived from an efficient nodal ordering by processing the elements in ascending sequence of their lowest numbered nodes.

This paper describes a FORTRAN 77 implementation of an algorithm\(^1\) for reducing the profile and root-mean-square wavefront of a sparse matrix with a symmetric pattern of zeros. The program has been tested on a broad range of problems, including those of Everstine,\(^2\) and the results suggest that it is superior to the widely used reverse Cuthill-McKee\(^3,4\) and Gibbs-King\(^5,6\) methods. In addition to giving a complete listing of the program, we also describe how it may be incorporated in a typical finite element code.

**NOTATION**

As noted by Cuthill and McKee,\(^3\) the ordering of a sparse symmetric matrix for minimum profile and wavefront is equivalent to the labelling of an undirected graph. The terminology of graph theory provides a succinct means of discussing various ordering schemes, and they are usually described in this framework. Before outlining the algorithm and its implementation, it is thus appropriate to state some basic definitions.

A *graph* consists of a set of members called *nodes*, together with a set of unordered pairs of distinct nodes called *edges*. A graph satisfying this definition is said to be *undirected* since each nodal pair defining the set of edges is unordered. Note that *loops*, where edges connect nodes to themselves, and *multiple edges*, which are pairs of nodes connected by more than one edge, are excluded. The *degree* of a node is the number of edges that are incident to it and each node must be labelled with an integer in the range 1, 2, \ldots, $N$. Any pair of nodes that is connected by an edge is said to be *adjacent*, whilst a *path* is defined by a sequence of edges such that consecutive edges share a common node. Two nodes are said to be connected if there is a path joining them and the *graph* itself is *connected* if each pair of nodes is connected.

The *distance* between a pair of nodes is defined as the number of edges on the shortest path connecting them, whilst the *diameter* of a graph is the maximum distance between all pairs of nodes. In general, a diameter may be defined by several pairs of nodes and all of these are said to be *peripheral nodes*. Following the accepted notation for describing ordering schemes,\(^4-6\) a *pseudo-diameter* is any estimate of the diameter which is found by some approximate algorithm. Nodes which define a pseudo-diameter are known as *pseudo-peripheral nodes*.

A concept which is central to many successful ordering algorithms is the *rooted level structure*. This is a partitioning of the nodes such that each node is assigned to one of the levels $l_1, l_2, \ldots, l_{h(r)}$.
in accordance with its distance from a specified root node \( r \). All nodes which lie at the same distance from the root node belong to the same level. The depth of a rooted level structure, \( h(r) \), is simply its total number of levels, whilst its width, \( w(r) \), is the maximum number of nodes which belong to a single level. More formally, the partitioning may be written as \( L(r) = \{ l_1, l_2, \ldots, l_{h(r)} \} \) and is defined by

1. \( l_1 = r \).
2. For \( i > 1 \), \( l_i \) is the set of all nodes which lie at a distance of \( i - 1 \) from the root node \( r \).

Clearly, the maximum distance of any node from the root node is equal to \( h(r) - 1 \). Letting \( \omega_i(r) \) denote the number of nodes on level \( i \), the width of the level structure is defined by

\[
    w(r) = \max_{1 \leq i \leq h(r)} \{ \omega_i(r) \}
\]

Any sparse matrix with a symmetric structure may be viewed as an undirected graph, as shown in Figure 1. Each node in the graph corresponds to a row in the matrix and the degree of a node gives the number of off-diagonal non-zeros in its row. Since any pair of nodes \( i \) and \( j \) are connected by an edge only if \( a_{ij} \) and \( a_{ji} \) are both non-zero, it follows that the total number of non-zeros in the matrix is equal to \( 2E + N \), where \( E \) is the number of edges and \( N \) is the number of nodes. (Note that for the two-dimensional grid shown in Figure 1, where each node is associated with two equations, the quantity \( 2E + N \) actually corresponds to the total number of non-zero 2 x 2 submatrices.) In finite element applications, the graph corresponding to a mesh is constructed such that all nodes which share an element are connected by an edge.

![A Grid of 4-Noded Quadrilaterals](image)

![The Structure of the Corresponding Matrix (X = Nonzero Entry)](image)

Figure 1. Graph representation of a sparse matrix with a symmetric structure
In order to illustrate some of the foregoing definitions, we note that the graph shown in Figure 1 is defined by the set of nodes \{1, 2, 3, 4, 5, 6\} together with the edges \{1, 2\}, \{1, 5\}, \{1, 6\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{2, 6\}, \{3, 4\}, \{3, 5\}, \{4, 5\}, \{5, 6\}. The degrees of nodes 1, 3, 4 and 6 are equal to three, whilst the degrees of nodes 2 and 5 are equal to five. By definition, we note that the sum of the nodal degrees is equal to twice the number of edges. Since all nodes in the graph are connected, the graph itself is connected. In order to illustrate the concept of a level structure, it is necessary to understand the notion of distance. The distance between nodes 1 and 4, for example, is two, and these nodes are peripheral since they define the diameter of the graph. Nodes 3 and 6 are also peripheral for the same reason. The level structure rooted at node 1 is shown schematically in Figure 2, where the boxes denote level numbers. More formally, this level structure may be represented by the set \(L(1) = \{l_1, l_2, l_3\}\), where \(l_1 = \{1\}\), \(l_2 = \{2, 5, 6\}\) and \(l_3 = \{3, 4\}\). The width and depth of this level structure are both equal to three.

**THE ALGORITHM**

The labelling algorithm is comprised of two distinct steps which we will briefly outline. A more detailed description of the scheme may be found in Sloan.¹

*Selection of pseudo-peripheral nodes*

Before commencing the node labelling phase we first find a pair of pseudo-peripheral nodes which lie at opposite ends of a pseudo-diameter. These serve as starting and end nodes for the node labelling and are determined as follows:

1. (First guess for starting node.) Scan all nodes in the graph and select a node \(s\) with the smallest degree.
2. (Generate rooted level structure.) Form the level structure rooted at node \(s\), i.e. \(L(s) = \{l_1, l_2, \ldots, l_{h(s)}\}\).
3. (Sort the last level.) Sort the nodes in \(l_{h(s)}\) in ascending sequence of their degrees.
4. (Shrink the last level.) Scan the sorted level \(l_{h(s)}\) and form a list of nodes, \(q\), containing only one node of each degree.
5. (Initialize.) Set \(w(e) = \infty\).
6. (Test for termination.) For each node \(i \in q\), in order of ascending degree, generate \(L(i) = \{l_1, l_2, \ldots, l_{h(i)}\}\). If \(h(i) > h(s)\) and \(w(i) < w(e)\) set \(s = i\) and go to step 3. Else, if \(w(i) < w(e)\) set \(e = i\) and \(w(e) = w(i)\).
7. (Exit.) Exit with starting node \(s\) and end node \(e\).
This algorithm is similar to that described by Sloan,\textsuperscript{1} but the shrinking strategy in step 4 has been modified slightly following a suggestion of Scott.\textsuperscript{7} Computational experiments indicate that many of the nodes in $l_{h(s)}$ at step 3 have identical degrees, particularly if the size of this level is large. Removing nodes with duplicate degrees in this list, so that only one node of each degree is present, often minimizes the number of level structures that need to be generated in step 6 without affecting the quality of the resulting pseudo-peripheral nodes. This shrinking strategy performs better than the one used originally in Sloan,\textsuperscript{4} especially for 'square' grids which may have large numbers of nodes in $l_{h(s)}$ in step 3. A slight penalty for this improved performance is that the overall algorithm requires an additional $N/2$ words of memory to relabel an arbitrary graph. As in the original algorithm, the short circuiting strategy in step 6 is retained. Imposing the condition that $w(i) < w(e)$ allows some of the level structures $L(i)$ to be aborted during their assembly, and is based on the premise that deep level structures are usually narrow.

**Node labelling algorithm**

After a pair of pseudo-peripheral nodes has been found using the preceding algorithm, we then generate the node labels in a single pass. To describe the algorithm succinctly, we define any node which has been assigned a new label as postactive. Nodes which are adjacent to a postactive node, but do not have a postactive status, are said to be active. Any node which is adjacent to an active node, but is not postactive or active, is said to be preactive, while each node which is not postactive, active or preactive is said to be inactive.

As the labelling progresses, the growth in the current wavefront is measured by a quantity called the **current degree**. All postactive nodes have current degrees of zero, whilst the current degree of an active node is equal to its number of preactive neighbours. The current degree of a preactive node is equal to its number of preactive and inactive neighbours plus one. Each inactive node has a current degree which is equal to its degree plus one. Prior to labelling, each node in the graph is inactive and hence has a current degree equal to its degree plus one. After the labelling is completed, all the nodes are postactive and have current degrees of zero.

An example which illustrates the above terminology is shown in Figure 3. The current degrees of nodes $x$, $y$ and $z$ are, respectively 1, 2 and 3.

To begin the node labelling procedure we require a starting node and an end node which define a pseudo-diameter. We then relabel the starting node as node one and form a list of nodes which

![Figure 3. Terminology for node-labelling algorithm](image-url)
are eligible to be labelled next. This list is comprised of nodes which are either active or preactive and constitutes a priority queue. Each node in the queue has a priority which is determined so that a low current degree and a large distance from the end node ensures a high priority. The node with the highest priority is chosen as the next node to be labelled and then deleted from the queue. The queue is then updated using the connectivity information for the graph and the whole process is repeated until all the nodes have been labelled. The overall algorithm may be summarized as follows:

1. (Entry.) Enter with the endpoints of a pseudo-diameter, nodes $s$ and $e$.
2. (Compute distances.) Form the level structure rooted at the end node, $L(e) = \{l_1, l_2, \ldots, l_{b(e)}\}$, and compute the distance, $\delta_i$, of each node $i$ from the end node. Note that if node $i$ belongs to $l_j$, then $\delta_i = j - 1$.
3. (Assign initial status and priority.) Assign each node in the graph an inactive status and compute its initial priority, $p_i$, according to
   \[ p_i = W_1 \cdot \delta_i - W_2 \cdot (d_i + 1) \]
   where $W_1$ and $W_2$ are integer weights and $d_i$ is the degree of node $i$.
4. (Initialize node count and priority queue.) Set $I = 0$, where $I$ is the count of nodes that have been labelled, and assign node $s$ a preactive status. Let $q$ denote a priority queue of length $n$. Insert $s$ in the priority queue by setting $n = 1$ and $q_1 = s$.
5. (Test for termination.) While the priority queue is not empty, which is signified by $n > 0$, do steps 6–9.
6. (Select node to be labelled.) Search the priority queue and locate the node $i$ which has the maximum priority according to
   \[ p_i = \max_{j \in q} \{p_j\} \]
   Let $m$ be the index of node $i$ such that $q_m = i$.
7. (Update queue and priorities.) Delete node $i$ from the priority queue by setting $q_m = q_n$ and decrementing $n$ according to $n \leftarrow n - 1$. If node $i$ is not preactive go to step 8. Else, examine each node $j$ which is adjacent to node $i$ and increment its priority according to $p_j \leftarrow p_j + W_2$ (this corresponds to decreasing the current degree of node $j$ by unity). If node $j$ is inactive, then insert it in the priority queue with a preactive status by setting $n \leftarrow n + 1$ and $q_n = j$.
8. (Label the next node.) Label node $i$ with its new number by incrementing the node count according to $n \leftarrow n + 1$ and setting $\eta_i = I$, where $\eta$ is the list of new node numbers. Assign node $i$ a postactive status.
9. (Update priorities and queue.) Examine each node $j$ which is adjacent to node $i$. If node $j$ is not preactive, take no action. Else, assign node $j$ an active status, set $p_j = p_j + W_2$, and examine each node $k$ which is adjacent to node $j$. If node $k$ is not postactive, increment its priority according to $p_k \leftarrow p_k + W_2$. If node $k$ is inactive, insert it in the priority queue with a preactive status by setting $n \leftarrow n + 1$ and $q_n = k$.
10. (Exit.) Exit with the new node labels, $\eta$, such that $\eta_i$ is the node number for node $i$.

The above algorithm differs from that described by Sloan\(^1\) in that we no longer insist that all priorities are non-negative. This does not affect the labellings produced. The values assigned to the weights $W_1$ and $W_2$ in step 3 reflect the importance of distance from the end node relative to that of current degree. Empirical tests on large collections of problems suggest that these criteria should be weighted with $W_1 = 1$ and $W_2 = 2$ to obtain good orderings.\(^1\) If we set $W_1 = 0$ and $W_2 = 1$, the node labelling algorithm is similar to one of the schemes proposed by King\(^2\) (except that
King gave no systematic procedure for determining the starting node and employed a criterion for breaking ties. Indeed, increasing the size of $W_2$ relative to $W_1$ places more emphasis on current degree and forces the algorithm to produce labellings which are similar to those produced by King's method. Because it uses current degree as the principal measure of priority, which is a purely local criterion, the performance of King's strategy is erratic and it may yield very poor labellings. If, on the other hand, we increase the size of $W_1$ relative to $W_2$, this places more emphasis on the global criterion of distance from the end node and forces the nodes to be labelled level by level. In order to avoid excessive growth in the wavefront, which may occur if insufficient weight is given to current degree, it is generally prudent to insist that $W_2 \geq W_1$.

**Element labelling algorithm**

The schemes described in the previous sections may be used to generate efficient node or element orderings for finite element analysis. For a profile solution strategy, where the solution efficiency is dictated by the order of the nodes, the elements may be numbered in an arbitrary fashion. With a frontal algorithm, however, it is necessary to determine an efficient element ordering since the equations are assembled and eliminated for each element in turn. A good element labelling may be derived from an optimized nodal ordering by sorting the elements in ascending sequence of their lowest numbered nodes. This ensures that the equations are eliminated in a similar order to that dictated by the optimized nodal ordering and has proved most effective in practice. An efficient method for determining the new element labels is as follows:

1. (Entry.) Enter with the list of nodes defining each element and the new node ordering, $\eta$, such that $\eta_i$ is the new node number for node $i$. Let the number of nodes for each element be $n$ and the list of element nodes be given by the set $\{v_1, v_2, \ldots, v_n\}$.
2. (Initialize.) Compute and store the lowest numbered node, $m_i$, for each element $i$ according to

   \[ m_i = \min_{1 \leq j \leq n} \{ \eta(v_j) \} \]

   Also initialize the list of element numbers, $e$, by setting $e_i = i$.
3. (Sort elements.) Sort the list $e$ in ascending sequence of its keys held in $m$.
4. (Exit.) Exit with the list of element numbers $e$ such that $e_i$ holds the old number for element $i$.

Note that the inverse permutation vector, which gives the new element numbers, can easily be generated from $e$. Once the element labels have been determined, the new node labelling held in $\eta$ can be discarded. For large grids with many elements, it is desirable to use an efficient sorting algorithm in step 3. A fast quicksort routine, which is written in FORTRAN 77 and requires only slight modification for the present application, may be found in Houlsby and Sloan. Alternatively, if an additional $N$ storage locations are available, it is possible to execute this step using a linear bin sort. As opposed to quicksort, which requires $O(e \log_e e)$ operations for a mesh with $e$ elements, this approach has the advantage that it takes only $O(e)$ operations. For grids comprised of high order elements, Sloan and Randolph have noted that it is necessary to consider the corner nodes only when relabelling the elements for a frontal solution scheme. This follows from the observation that an element ordering which is efficient for a grid of low order elements is also efficient for an equivalent grid of high order elements. Since a mesh of high order elements may have a small number of corner nodes, but a large number of nodes in total, this approach leads to considerable economies in the ordering phase.
IMPLEMENTATION

An illustrative subroutine, which generates a node graph from typical finite element data, is given in Appendix I. A complete implementation of the algorithm for reducing profile and wavefront is given in Appendix II. To the best of the Author's knowledge, these codes obey the syntax of standard FORTRAN 77 and ought to be portable. The algorithm for generating a node graph from finite element data is implemented in a single subroutine named GRAPH. The node labelling program is comprised of six subroutines (LABEL, DIAMTR, ROOTLS, ISORTI, NUMBER and PROFIL) and deals with both connected and disconnected graphs. All of the codes contain numerous comments in the hope that they are self-documenting. Each of the routines will be discussed in turn, however, to illustrate the detail of the implementation.

Subroutine GRAPH

This routine generates the node graph which corresponds to a finite element mesh. It is based on the rule that all nodes which share an element are connected by an edge and requires the node lists for each element to be input. Following George and Liu, the graph is stored as an adjacency list which is accessed through a pointer vector. If ADJ and XADJ denote the adjacency list and pointer vector, respectively, the nodes adjacent to node i are found in ADJ(J) for J = XADJ(J), XADJ(J) + 1, ..., XADJ(J + 1) - 1, whilst the degree of node i is simply XADJ(i + 1) - XADJ(i). This data structure stores each edge twice for speed of access and requires \(2E + N + 1\) words of memory. To illustrate this scheme, the adjacency list for the graph shown in Figure 1 is stored as ADJ = \{2, 6, 5, 1, 3, 6, 4, 2, 5, 4, 5, 2, 3, 6, 1, 2, 3, 4\} and the corresponding pointer vector is XADJ = \{1, 4, 9, 12, 15, 20, 23\}. A similar data structure, which has proved convenient in practical finite element programs, is used to store the nodal lists for each element (the routine may easily be modified to deal with other storage conventions). The algorithm used to generate the node graph is an enhanced version of that given by Collins. Although fast, Collins's scheme requires excessive storage if the maximum nodal degree is significantly greater than the average nodal degree and is thus unsuitable for grids of high order elements. The proposed algorithm avoids this limitation by first scanning the element node lists to compute an upper bound estimate for the degree of each node. This information is then used to allocate space in the adjacency vector ADJ which is initially filled with zeros and has a specified length of IADJ. For a mesh comprised of NE elements, where each element is of the same type and has NEN nodes, sufficient space in ADJ is guaranteed by setting IADJ = NE * NEN * (NEN - 1). For a general mesh of elements where each element i has NEN(i) nodes, IADJ needs to be set equal to \(\sum_{i=1}^{NE} NEN(i) * (NEN(i) - 1)\). The adjacency list ADJ is formed by looping over each element and connecting all pairs of nodes in its node list. Any pair of nodes that has already been connected by a scan of a previous element is ignored. Vacant entries in ADJ are signified by zeros which allow new entries to be inserted in their correct positions. After the node lists for all the elements have been examined, any zeros remaining in ADJ are removed by compression and the pointer vector XADJ is updated. On average, IADJ needs to be about 25 per cent greater than the final length of ADJ (which is equal to \(2E\)). The amount of 'elbow room' required, however, is dependent on the type of element used in the mesh. A fine grid of 3-noded triangles, for example, requires that IADJ/2E = 2, whilst for a grid of 2-noded bar elements IADJ/2E = 1.

Subroutine LABEL

Once the node graph has been generated from the finite element data, this is the only routine that needs to be called by the user in order to form the list of node labels. It controls the overall
flow of the program and, excluding the $2E + N + 1$ words of memory needed to hold the structure of the graph, requires $4N + 1$ words of memory to label an arbitrary graph. The new node labels are held in a permutation vector called NNN, such that NNN($j$) is the new node number for node $i$. Note that if the profile for the original ordering is less than the profile for the new ordering, then the old node numbers are retained by setting NNN($1$) = $i$. Subroutine LABEL calls three subroutines: DIAMTR to select the starting and end nodes, NUMBER to label the nodes and PROFIL to compute the profiles for the old and new node labels. It deals with disconnected graphs by labelling each component in turn.

Subroutine DIAMTR

This routine computes a pair of nodes which define the endpoints of a pseudo-diameter. It uses the algorithm described in the section ‘Selection of pseudo-peripheral nodes’ and operates on each component of the graph one at a time. Following George and Liu, we flag components which have already been considered (labelled) by using a masking vector called MASK. The only nodes that are visible to DIAMTR are those which have MASK = 0. This convention allows the arrays MASK and NNN, which are both of length $N$, to share the same storage. Subroutine DIAMTR calls two other subroutines; ISORT1 to perform an insertion sort and ROOTLS to generate a rooted level structure. After the starting and end nodes have been selected, which typically requires only two or three iterations, we use MASK to hold the distance of each node from the end node. The other output parameters are SNODE, the node at which the labelling is to be started, and NC, the total number of nodes in the same component as SNODE.

Subroutine ISORT1

This code is a straightforward implementation of insertion sort and orders an integer list in ascending sequence of its integer keys.

Subroutine ROOTLS

This subroutine generates a level structure rooted at the node ROOT and is a modification of the code given by George and Liu. Upon exit, the level structure is held in the list LS and pointer vector XLS. The nodes on level $i$ are found in LS($j$), where $j = XLS(i), XLS(i) + 1, \ldots, XLS(i + 1) - 1$. In all cases the length of LS is equal to NC, which has an upper bound of $N$. The length required for XLS, however, depends on the number of levels in LS. Since there are a maximum of $N$ levels for a graph with $N$ nodes, an upper bound on the storage required for XLS is $N + 1$. Note that the assembly of the level structure is aborted if any level is found to have a width which is greater than or equal to the parameter MAXWID. In this case, all output from ROOTLS should be ignored. The depth and width of the rooted level structure, assuming that it has not been aborted during assembly, are given by the output parameters DEPTH and WIDTH.

Subroutine NUMBER

The scheme for labelling the nodes, as described previously in the section entitled ‘Node labelling algorithm’, is implemented in this subroutine. A key feature of the labelling process is the list of active and preactive nodes, which is maintained as a priority queue. The nodes currently in the queue are stored in the array Q, whilst a separate list named P is used to record the priorities of all of the nodes. The status of each node during the ordering process (i.e. whether it is postactive, active, preactive or inactive) is kept in the array S. The list Q has a maximum length of NC, whilst the lists $P$ and $S$ have lengths equal to $N$. Since the list $Q$ is kept unordered, locating the node with
the maximum priority requires $O(n)$ operations for a queue of length $n$. Deleting, inserting or changing the priority of a node, however, is trivial. Although simple, this implementation of the priority queue has proved highly efficient. Its success follows from the fact that the number of nodes in the queue is typically a small fraction of the total number of nodes in the graph (or component). Another elegant method for implementing a priority queue, which is efficient for large queues, is based on a variant of a binary tree data structure called a heap.\cite{12} A heap data structure permits an item to be deleted, inserted or have its priority changed in $O(\log_2 n)$ operations, whilst the searching step is trivial. In order to change the priority of a node in a heap efficiently, it is necessary to keep track of its position. Consequently, the heap implementation requires an additional array of length $N$. Detailed comparisons of the two methods for managing a priority queue suggest that the simple unordered list implementation is generally superior to the heap implementation, at least for small and medium sized problems involving several thousand nodes. For larger graphs, where the priority queue grows to an appreciable size, the relative performance of the heap data structure improves and it eventually becomes the method of choice. Preliminary numerical experiments suggest that the unordered list data structure is to be preferred for problems where the root-mean-square wavefront does not exceed several hundred nodes.

In subroutine NUMBER, we note that the status array $S$ is also used to hold the new node numbers, and thus shares the same storage as the permutation array $NNN$. This economy of storage is achieved by adopting the convention that all postactive nodes have a status which is greater than zero. Note that, in step 9 of the node labelling algorithm, it is unnecessary to update the priority of node $k$ if it is postactive (since the node has already been assigned a new label and its priority is no longer of interest). In NUMBER, we choose to update the priority of node $k$ regardless of its status, as this leads to cleaner code which avoids many redundant tests.

Subroutine PROFIL

This subroutine is straightforward and uses the original and new node numbers to compute the original and new profiles.

RESULTS

In order to provide direct comparisons between various algorithms, we employ the thirty test matrices collected by Everstine.\cite{2} These correspond to a diverse range of finite element grids and have been widely studied. Of particular interest to the present work are the results given by Armstrong,\cite{13} who obtained near-optimal profiles and wavefronts for Everstine’s problems by using a simulated annealing algorithm. Although it is much too slow for practical use, this scheme establishes valuable benchmarks for gauging the performance of faster but more approximate algorithms.

Armstrong’s results, together with those obtained from the code of Appendix II, are shown in Tables I and II. These tables also list results for the reverse Cuthill–McKee algorithm, as implemented in the FORTRAN program of George and Liu,\cite{4} and the Gibbs–King algorithm as described by Lewis.\cite{6} The code for Lewis’s implementation is distributed by the Association for Computing Machinery as Algorithm 582 and is available on request.

From Table I it is clear that the reverse Cuthill–McKee algorithm gives poorer results than any of the other schemes since, on average, it yields a profile which is 31 per cent in excess of the simulated annealing profile. This performance does not compare well with that of the Sloan algorithm where, typically, the computed profile is just 10 per cent above the simulated annealing profile. The average reduction achieved by the Gibbs–King scheme is between those of the
reverse Cuthill–McKee and Sloan schemes. As practical applications often deal with a single type of matrix which has a particular sparsity pattern, it is not only the average performance of an algorithm which is of interest but also its worst performance. The results for the reverse Cuthill–McKee and Gibbs–King schemes are not encouraging in this regard since they show worst case profiles which are, respectively, at least 114 per cent and 76 per cent above the optimum profiles. Moreover, on another 3 occasions, these two schemes give profiles which are at least 50 per cent above the optimum. The performance of the Sloan algorithm, on the other hand, is much less erratic and in the worst case its profile exceeds the simulated annealing profile by 24 per cent. Excluding the results for the simulated annealing scheme, which are not bettered by any of the three algorithms tested, the Sloan scheme yields the lowest profile on 24 occasions whilst the Gibbs–King scheme gives the lowest profile on 4 occasions. The reverse Cuthill–McKee scheme is unable to achieve the lowest profile for any of the examples.

### Table I. Results for profile reduction on Everstine's test problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>N</th>
<th>$P_0$</th>
<th>$P_{rcm}$</th>
<th>$P_{gk}$</th>
<th>$P_{slo}$</th>
<th>$P_{arm}$</th>
<th>$P_{rcm} / P_{arm}$</th>
<th>$P_{gk} / P_{arm}$</th>
<th>$P_{slo} / P_{arm}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59</td>
<td>464</td>
<td>314</td>
<td>314</td>
<td>294</td>
<td>273</td>
<td>1.15</td>
<td>1.15</td>
<td>1.08</td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td>640</td>
<td>217</td>
<td>193</td>
<td>193</td>
<td>193</td>
<td>1.12</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>72</td>
<td>244</td>
<td>244*</td>
<td>244*</td>
<td>244*</td>
<td>244*</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>4</td>
<td>87</td>
<td>2336</td>
<td>696</td>
<td>682</td>
<td>525</td>
<td>515</td>
<td>1.35</td>
<td>1.32</td>
<td>1.02</td>
</tr>
<tr>
<td>5</td>
<td>162</td>
<td>2806</td>
<td>1641</td>
<td>1759</td>
<td>1554</td>
<td>1554</td>
<td>1.09</td>
<td>1.09</td>
<td>1.09</td>
</tr>
<tr>
<td>6</td>
<td>193</td>
<td>7953</td>
<td>5505</td>
<td>4609</td>
<td>4618</td>
<td>4618</td>
<td>1.25</td>
<td>1.25</td>
<td>1.05</td>
</tr>
<tr>
<td>7</td>
<td>198</td>
<td>5817</td>
<td>1417</td>
<td>1313</td>
<td>1297</td>
<td>1297</td>
<td>1.02</td>
<td>1.02</td>
<td>1.02</td>
</tr>
<tr>
<td>8</td>
<td>209</td>
<td>9712</td>
<td>3819</td>
<td>4032</td>
<td>3316</td>
<td>2693</td>
<td>1.42</td>
<td>1.50</td>
<td>1.23</td>
</tr>
<tr>
<td>9</td>
<td>221</td>
<td>10131</td>
<td>2225</td>
<td>2154</td>
<td>2052</td>
<td>1848</td>
<td>1.20</td>
<td>1.17</td>
<td>1.11</td>
</tr>
<tr>
<td>10</td>
<td>234</td>
<td>1999</td>
<td>1539</td>
<td>1349</td>
<td>1089</td>
<td>1016</td>
<td>1.31</td>
<td>1.33</td>
<td>1.07</td>
</tr>
<tr>
<td>11</td>
<td>245</td>
<td>4179</td>
<td>4179*</td>
<td>3813</td>
<td>2676</td>
<td>2161</td>
<td>1.93</td>
<td>1.76</td>
<td>1.24</td>
</tr>
<tr>
<td>12</td>
<td>307</td>
<td>8132</td>
<td>8132*</td>
<td>8132*</td>
<td>7550</td>
<td>6535</td>
<td>1.24</td>
<td>1.24</td>
<td>1.16</td>
</tr>
<tr>
<td>13</td>
<td>310</td>
<td>3006</td>
<td>3006*</td>
<td>3006*</td>
<td>2982</td>
<td>2940</td>
<td>1.02</td>
<td>1.02</td>
<td>1.01</td>
</tr>
<tr>
<td>14</td>
<td>346</td>
<td>9054</td>
<td>8034</td>
<td>7681</td>
<td>6726</td>
<td>6136</td>
<td>1.31</td>
<td>1.25</td>
<td>1.10</td>
</tr>
<tr>
<td>15</td>
<td>361</td>
<td>5445</td>
<td>5075</td>
<td>5060</td>
<td>5062</td>
<td>4992</td>
<td>1.02</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td>16</td>
<td>419</td>
<td>40145</td>
<td>8649</td>
<td>8073</td>
<td>7255</td>
<td>6512</td>
<td>1.33</td>
<td>1.24</td>
<td>1.11</td>
</tr>
<tr>
<td>17</td>
<td>492</td>
<td>34282</td>
<td>7067</td>
<td>5513</td>
<td>3412</td>
<td>3304</td>
<td>2.14</td>
<td>1.67</td>
<td>1.03</td>
</tr>
<tr>
<td>18</td>
<td>503</td>
<td>36417</td>
<td>15319</td>
<td>15042</td>
<td>14436</td>
<td>11958</td>
<td>1.28</td>
<td>1.26</td>
<td>1.21</td>
</tr>
<tr>
<td>19</td>
<td>512</td>
<td>6530</td>
<td>5319</td>
<td>4970</td>
<td>4680</td>
<td>4384</td>
<td>1.21</td>
<td>1.13</td>
<td>1.07</td>
</tr>
<tr>
<td>20</td>
<td>592</td>
<td>23937</td>
<td>11440</td>
<td>10925</td>
<td>10073</td>
<td>9417</td>
<td>1.21</td>
<td>1.16</td>
<td>1.07</td>
</tr>
<tr>
<td>21</td>
<td>607</td>
<td>30615</td>
<td>17868</td>
<td>14760</td>
<td>14412</td>
<td>13065</td>
<td>1.37</td>
<td>1.13</td>
<td>1.10</td>
</tr>
<tr>
<td>22</td>
<td>758</td>
<td>23871</td>
<td>8580</td>
<td>8175</td>
<td>7598</td>
<td>7123</td>
<td>1.20</td>
<td>1.15</td>
<td>1.07</td>
</tr>
<tr>
<td>23</td>
<td>869</td>
<td>20397</td>
<td>19293</td>
<td>15728</td>
<td>14909</td>
<td>13207</td>
<td>1.46</td>
<td>1.19</td>
<td>1.13</td>
</tr>
<tr>
<td>24</td>
<td>878</td>
<td>26933</td>
<td>22391</td>
<td>19696</td>
<td>20537</td>
<td>17835</td>
<td>1.26</td>
<td>1.10</td>
<td>1.15</td>
</tr>
<tr>
<td>25</td>
<td>918</td>
<td>109273</td>
<td>23105</td>
<td>20498</td>
<td>17236</td>
<td>15949</td>
<td>1.45</td>
<td>1.29</td>
<td>1.08</td>
</tr>
<tr>
<td>26</td>
<td>992</td>
<td>263298</td>
<td>38128</td>
<td>34068</td>
<td>33928</td>
<td>32528</td>
<td>1.17</td>
<td>1.05</td>
<td>1.04</td>
</tr>
<tr>
<td>27</td>
<td>1005</td>
<td>122075</td>
<td>43068</td>
<td>40141</td>
<td>36396</td>
<td>32512</td>
<td>1.32</td>
<td>1.23</td>
<td>1.10</td>
</tr>
<tr>
<td>28</td>
<td>1007</td>
<td>26793</td>
<td>24703</td>
<td>22465</td>
<td>22669</td>
<td>19913</td>
<td>1.24</td>
<td>1.13</td>
<td>1.14</td>
</tr>
<tr>
<td>29</td>
<td>1242</td>
<td>111430</td>
<td>50052</td>
<td>52952</td>
<td>36822</td>
<td>33098</td>
<td>1.51</td>
<td>1.60</td>
<td>1.11</td>
</tr>
<tr>
<td>30</td>
<td>2680</td>
<td>590543</td>
<td>105663</td>
<td>99271</td>
<td>89686</td>
<td>84900</td>
<td>1.24</td>
<td>1.17</td>
<td>1.06</td>
</tr>
</tbody>
</table>

Notes: $N =$ number of nodes; $P_0 =$ original profile; $P_{rcm} =$ profile for reverse Cuthill–McKee; $P_{gk} =$ profile for Gibbs–King; $P_{slo} =$ profile for Sloan; $P_{arm} =$ profile for Armstrong; * = no improvement

Average: 1.31 1.22 1.10
The root-mean-square wavefront reductions for the various algorithms, given in Table II, exhibit similar trends to those shown in the profile results. The reverse Cuthill-McKee algorithm again performs less favourably than the Gibbs-King and Sloan schemes and, on average, yields a root-mean-square wavefront which is some 35 per cent above that of the simulated annealing algorithm. In the worst case, its root-mean-square wavefront is 124 per cent in excess of the simulated annealing root-mean-square wavefront. The Sloan scheme gives the most consistent reductions, its average and worst case results are respectively 11 per cent and 27 per cent above those given by simulated annealing, and compared with the reverse Cuthill-McKee and Gibbs-King schemes it yields the lowest root-mean-square wavefront on 24 occasions.

Including the storage needed to hold the structure of the graph, the reverse Cuthill-McKee and Sloan implementations require, respectively, $2E + 4N + 2$ and $2E + 5N + 2$ words of integer memory to compute their orderings. As implemented by Lewis, the storage required by the

<table>
<thead>
<tr>
<th>Problem</th>
<th>$N$</th>
<th>$F_0$</th>
<th>$F_{rm}$</th>
<th>$F_{gk}$</th>
<th>$F_{slo}$</th>
<th>$F_{arm}$</th>
<th>$F_{rm}$</th>
<th>$F_{gk}$</th>
<th>$F_{slo}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59</td>
<td>5.45</td>
<td>5.53</td>
<td>5.15</td>
<td>4.74</td>
<td>1.15</td>
<td>1.17</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td>3.34</td>
<td>2.94</td>
<td>2.94</td>
<td>2.94</td>
<td>1.14</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>72</td>
<td>3.46</td>
<td>3.46*</td>
<td>3.46*</td>
<td>3.12</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>87</td>
<td>8.50</td>
<td>6.30</td>
<td>6.16</td>
<td>1.38</td>
<td>1.34</td>
<td>1.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>162</td>
<td>19.96</td>
<td>10.09</td>
<td>7.97</td>
<td>1.32</td>
<td>1.27</td>
<td>1.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>193</td>
<td>30.48</td>
<td>24.73</td>
<td>23.70</td>
<td>1.29</td>
<td>1.05</td>
<td>1.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>198</td>
<td>7.64</td>
<td>6.88</td>
<td>6.83</td>
<td>1.12</td>
<td>1.02</td>
<td>1.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>209</td>
<td>19.23</td>
<td>16.77</td>
<td>13.33</td>
<td>1.44</td>
<td>1.53</td>
<td>1.26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>221</td>
<td>10.52</td>
<td>9.71</td>
<td>8.64</td>
<td>1.22</td>
<td>1.18</td>
<td>1.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>234</td>
<td>7.30</td>
<td>6.29</td>
<td>4.95</td>
<td>4.59</td>
<td>1.59</td>
<td>1.37</td>
<td>1.08</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>245</td>
<td>16.64</td>
<td>11.76</td>
<td>9.23</td>
<td>2.00</td>
<td>1.80</td>
<td>1.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>307</td>
<td>25.65</td>
<td>22.33</td>
<td>18.42</td>
<td>1.33</td>
<td>1.26</td>
<td>1.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>310</td>
<td>9.76</td>
<td>9.62</td>
<td>6.12</td>
<td>1.02</td>
<td>1.02</td>
<td>1.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>346</td>
<td>23.29</td>
<td>20.23</td>
<td>18.42</td>
<td>1.33</td>
<td>1.26</td>
<td>1.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>361</td>
<td>14.23</td>
<td>14.24</td>
<td>14.08</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>419</td>
<td>19.96</td>
<td>17.80</td>
<td>15.96</td>
<td>1.34</td>
<td>1.25</td>
<td>1.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>492</td>
<td>11.76</td>
<td>10.30</td>
<td>10.28</td>
<td>1.14</td>
<td>1.14</td>
<td>1.11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>503</td>
<td>18.32</td>
<td>16.65</td>
<td>16.65</td>
<td>1.24</td>
<td>1.18</td>
<td>1.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>512</td>
<td>13.16</td>
<td>11.44</td>
<td>10.30</td>
<td>1.28</td>
<td>1.14</td>
<td>1.11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>592</td>
<td>19.70</td>
<td>18.32</td>
<td>18.32</td>
<td>1.24</td>
<td>1.24</td>
<td>1.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>607</td>
<td>27.25</td>
<td>26.71</td>
<td>24.32</td>
<td>1.40</td>
<td>1.12</td>
<td>1.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>758</td>
<td>12.01</td>
<td>10.92</td>
<td>10.05</td>
<td>1.27</td>
<td>1.20</td>
<td>1.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>869</td>
<td>18.91</td>
<td>18.94</td>
<td>15.63</td>
<td>1.55</td>
<td>1.27</td>
<td>1.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>878</td>
<td>22.60</td>
<td>24.22</td>
<td>20.98</td>
<td>1.27</td>
<td>1.08</td>
<td>1.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>918</td>
<td>19.52</td>
<td>18.02</td>
<td>18.02</td>
<td>1.48</td>
<td>1.30</td>
<td>1.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>992</td>
<td>33.51</td>
<td>33.51</td>
<td>33.51</td>
<td>1.20</td>
<td>1.03</td>
<td>1.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1005</td>
<td>14.40</td>
<td>13.79</td>
<td>13.79</td>
<td>1.45</td>
<td>1.33</td>
<td>1.20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>1007</td>
<td>23.00</td>
<td>20.26</td>
<td>20.26</td>
<td>1.26</td>
<td>1.12</td>
<td>1.14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1242</td>
<td>27.30</td>
<td>27.30</td>
<td>1.55</td>
<td>1.69</td>
<td>1.11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>2680</td>
<td>32.27</td>
<td>32.27</td>
<td>1.26</td>
<td>1.18</td>
<td>1.06</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average 1.35 1.24 1.11

Notes: $N$ = number of nodes; $F_0$ = original root-mean-square wavefront; $F_{rm}$ = root-mean-square wavefront for reverse Cuthill-McKee; $F_{gk}$ = root-mean-square wavefront for Gibbs-King; $F_{slo}$ = root-mean-square wavefront for Sloan; $F_{arm}$ = root-mean-square wavefront for Armstrong; * = no improvement
Gibbs–King algorithm depends on the problem at hand. Empirical evidence suggests that its average storage requirement is roughly $2E + 7N$ words, but this increases to $2E + 9N + 3$ for the worst case.

With regard to efficiency, detailed timing statistics indicate that the reverse Cuthill–McKee algorithm is typically 140 per cent faster than the Gibbs–King algorithm and 50 per cent faster than the Sloan algorithm. All the schemes, however, are easily fast enough for most practical applications. We note in closing that a more comprehensive set of results and timing statistics, for a larger range of test problems which include those of Everstine, may be found in Sloan and Ng.\textsuperscript{14}

**CONCLUSION**

A FORTRAN 77 program for reducing the profile and root-mean-square wavefront of a sparse matrix has been described. The code has been tested extensively on a broad range of finite element problems and found to be superior to existing methods. It can be used to provide efficient orderings for both profile and frontal solution schemes.

**APPENDIX I**

```fortran
SUBROUTINE GRAPH(N,NE,INPN,NPN,XNPN,IADJ,ADJ,XADJ)

* PURPOSE:
* Form adjacency list for a graph corresponding to a finite element
* mesh

* INPUT:
* N - Number of nodes in graph (finite element mesh)
* NE - Number of elements in finite element mesh
* INPN - Length of NPN = XNPN(NE+1)-1
* NPN - List of node numbers for each element
* XNPN - Index vector for NPN
  * nodes for element I are found in NPN(J), where
    * J = XNPN(I), XNPN(I)+1, ..., XNPN(I+1)-1
* IADJ - Length of vector ADJ
  * - Set IADJ=NE*NEN*(NEN-1) for a mesh of a single type of
  * element with NEN nodes
  * - IADJ=(NEN(1)*(NEN(1)-1)+,....,NEN(NE)*(NEN(NE)-1))
  * for a mesh of elements with varying numbers of nodes
* ADJ - Undefined
* XADJ - Undefined

* OUTPUT:
* N - Unchanged
* NE - Unchanged
* INPN - Unchanged
* NPN - Unchanged
```

[Image of a page from a document]
* XNPN - Unchanged
* IADJ - Unchanged
* ADJ - Adjacency list for all nodes in graph
  - List of length 2E where E is the number of edges in
    the graph (note that 2E = XADJ(N+1)-1)
* XADJ - Index vector for ADJ
  - Nodes adjacent to node I are found in ADJ(J), where
    J = XADJ(I), XADJ(I)+1, ..., XADJ(I+1)-1
* - Degree of node I given by XADJ(I+1)-XADJ(I)

NOTES:
------

This routine typically requires about 25 percent elbow room for
assembling the ADJ list (i.e. IADJ/2E is typically around 1.25).
In some cases, the elbow room may be larger (IADJ/2E is slightly
less than 2 for the 3-noded triangle) and in other cases it may be
zero (IADJ/2E = 1 for bar elements)

PROGRAMMER: Scott Sloan
--------------

LAST MODIFIED: 10 March 1989 Scott Sloan
--------------

********************************************************************************
INTEGER N,NE,NODEJ,NODEK,MSTRT,IADJ,I,J,K,JSTRT,JSTOP,LSTRT,LSTOP,
+ L,NEN1,MSTOP,M,INPN,XNPN(NE+1),NPN(INPN),ADJ(IADJ),XADJ(N+1)
* Initialise the adjacency list and its index vector
* DO 5 I=1,IADJ
  ADJ(I)=0
  5 CONTINUE
DO 10 I=1,N
  XADJ(I)=0
  10 CONTINUE

* Estimate the degree of each node (always an overestimate)
* DO 30 I=1,NE
  JSTRT=XNPN(I)
  JSTOP=XNPN(I+1)-1
  NEN1=JSTOP-JSTRT
  DO 20 J=JSTRT,JSTOP
    NODEJ=NPN(J)
    XADJ(NODEJ)=XADJ(NODEJ)+NEN1
  20 CONTINUE
DO 30 CONTINUE

* Reconstruct XADJ to point to start of each set of neighbours
* L=1
DO 40 I=1,N
  L=L+XADJ(I)
  XADJ(I)=L-XADJ(I)
  40 CONTINUE
CONTINUE
XADJ(N+1)=L
*
* Form adjacency list (which may contain zeros)
*
DO 90 I=1,NE
  JSTRT=XNPN(I)
  JSTOP=XNPN(I+1)-1
DO 80 J=JSTRT,JSTOP-1
  NODEJ=NPN(J)
  LSTRT=XADJ(NODEJ)
  LSTOP=XADJ(NODEJ+1)-1
  DO 70 K=J+1,JSTOP
     NODEK=NPN(K)
     DO 50 L=LSTRT,LSTOP
        IF(ADJ(L).EQ.NODEK)GOTO 70
        IF(ADJ(L).EQ.O)GOTO 55
  50 CONTINUE
  WRITE(6,1000)
  STOP
55 CONTINUE
ADJ(L)=NODEK
MSTRT=XADJ(NODEK)
MSTOP=XADJ(NODEK+1)-1
DO 60 M=MSTRT,MSTOP
   IF(ADJ(M).EQ.O)GOTO 65
  60 CONTINUE
  WRITE(6,1000)
  STOP
65 CONTINUE
ADJ(M)=NODEJ
70 CONTINUE
80 CONTINUE
90 CONTINUE
*
* Strip any zeros from adjacency list
*
K=0
JSTRT=1
DO 110 I=1,N
  JSTOP=XADJ(I+1)-1
  DO 100 J=JSTRT,JSTOP
     IF(ADJ(J).EQ.O)GOTO 105
     K=K+1
     ADJ(K)=ADJ(J)
  100 CONTINUE
105 CONTINUE
XADJ(I+1)=K+1
JSTRT=JSTOP+1
110 CONTINUE
*
* Error message
*
1000 FORMAT(/,1X,'***ERROR IN GRAPH***',
+     /,1X,'CANNOT ASSEMBLE NODE ADJACENCY LIST',
+     /,1X,'CHECK NPN AND XNPN ARRAYS')
END
SUBROUTINE LABEL(N,E2,ADJ,XADJ,NNN,IW,OLDPRO,NEWPRO)

********************************************************************************

* PURPOSE:  
*  
*  Label a graph for small profile and rms wavefront  
*  
*  INPUT:  
*  
*  N  - Total number of nodes in graph  
*  E2  - Twice the number of edges in the graph = XADJ(N+1)-1  
*  ADJ  - Adjacency list for all nodes in graph  
*  - List of length 2E where E is the number of edges in  
*  the graph and 2E = XADJ(N+1)-1  
*  XADJ  - Index vector for ADJ  
*  - Nodes adjacent to node I are found in ADJ(J), where  
*  J = XADJ(I), XADJ(I)+1, ..., XADJ(I+1)-1  
*  - Degree of node I given by XADJ(I+1)-XADJ(I)  
*  NNN  - Undefined  
*  IW  - Undefined  
*  OLDPRO  - Undefined  
*  NEWPRO  - Undefined  
*  
*  OUTPUT:  
*  
*  N  - Unchanged  
*  E2  - Unchanged  
*  ADJ  - Unchanged  
*  XADJ  - Unchanged  
*  NNN  - List of new node numbers  
*  - New number for node I given by NNN(I)  
*  - If original node numbers give a smaller profile then  
*  NNN is set so that NNN(I)=I for I=1,N  
*  IW  - Not used  
*  OLDPRO  - Profile using original node numbering  
*  NEWPRO  - Profile for new node numbering  
*  - If original profile is smaller than new profile, then  
*  original node numbers are used and NEWPRO=OLDPRO  
*  
*  SUBROUTINES CALLED: DIAMTR, NUMBER, PROFIL  
*  
*  PROGRAMMER: Scott Sloan  
*  
*  LAST MODIFIED: 10 March 1889 Scott Sloan  
*  
********************************************************************************
INTEGER N, I1, I2, I3, I, SNODE, LSTNUM, NC, OLDPRO, NEWPRO, E2, XADJ(N+1),
  + ADJ(E2), NNN(N), IW(3*N+1)

* Set all new node numbers = 0
* This is used to denote all visible nodes

DO 10 I=1,N
  NNN(I)=0
10 CONTINUE

* Define offsets
* I1=1
I2=I1+N
I3=I2+N+1

* Loop while some nodes remain unnumbered
* LSTNUM=0
20 IF(LSTNUM.LT.N)THEN
  * Find end points of p-diameter for nodes in this component
  * Compute distances of nodes from end node
  CALL DIAMTR(N, E2, ADJ, XADJ, NNN, IW(I1), IW(I2), IW(I3), SNODE, NC)
  * Number nodes in this component
  CALL NUMBER(N, NC, SNODE, LSTNUM, E2, ADJ, XADJ, NNN, IW(I1), IW(I2))
  GOTO 20
END IF

* Compute profiles for old and new node numbers
* CALL PROFIL(N, NNN, E2, ADJ, XADJ, OLDPRO, NEWPRO)
* Use original numbering if it gives a smaller profile
* IF(OLDPRO.LT.NEWPRO)THEN
  DO 30 I=1,N
    NNN(I)=I
 30 CONTINUE
NEWPRO=OLDPRO
END IF
END

SUBROUTINE DIAMTR(N, E2, ADJ, XADJ, MASK, LS, XLS, HLEVEL, SNODE, NC)
******************************************************************************
* PURPOSE:
* --------
* Find nodes which define a pseudo-diameter of a graph and store
* distances from end node
******************************************************************************
INPUT:

- N: The total number of nodes in the graph
- E2: Twice the number of edges in the graph = XADJ(N+1)-1
- ADJ: Adjacency list for all nodes in the graph
- XADJ: Index vector for ADJ
- MASK: Masking vector for graph
- LS: List of distances of nodes from the end node
- XLS: List of nodes in the component
- HLEVEL: Not used
- SNODE: Starting node for numbering
- NC: The number of nodes in this component of graph

OUTPUT:

- N: Unchanged
- E2: Unchanged
- ADJ: Unchanged
- XADJ: Unchanged
- MASK: List of distances of nodes from the end node
- LS: List of nodes in the component
- XLS: Not used
- HLEVEL: Not used
- SNODE: Starting node for numbering
- NC: The number of nodes in this component of graph

SUBROUTINES CALLED: ROOTLS, ISORTI

NOTE: SNODE and ENODE define a pseudo-diameter

PROGRAMMER: Scott Sloan

LAST MODIFIED: 10 March 1989 Scott Sloan
DEGREE=XADJ(I+1)-XADJ(I)
IF(DEGREE.LT.MINDEG)THEN
  SNODE=I
  MINDEG=DEGREE
END IF
END IF
10 CONTINUE

* Generate level structure for node with min degree
* CALL ROOTLS(N,SNODE,N+1,E2,ADJ,XADJ,MASK,LS,XLS,SDEPTH,WIDTH)
* Store number of nodes in this component
* NC=XLS(SDEPTH+1)-1
* Iterate to find start and end nodes
* 15 CONTINUE
* Store list of nodes that are at max distance from starting node
* Store their degrees in XLS
* HSIZE=0
ISTRT=XLS(SDEPTH)
ISTOP=XLS(SDEPTH+1)-1
DO 20 I=ISTRT,ISTOP
  NODE=LS(I)
  HSIZE=HSIZE+1
  HLEVEL(HSIZE)=NODE
  XLS(NODE)=XADJ(NODE+1)-XADJ(NODE)
20 CONTINUE

* Sort list of nodes in ascending sequence of their degree
* Use insertion sort algorithm
* IF(HSIZE.GT.1)CALL ISORTI(HSIZE,HLEVEL,N,XLS)
* Remove nodes with duplicate degrees
* ISTOP=HSIZE
HSIZE=1
DEGREE=XLS(HLEVEL(1))
DO 25 I=2,ISTOP
  NODE=HLEVEL(I)
  IF(XLS(NODE).NE.DEGREE)THEN
    DEGREE=XLS(NODE)
    HSIZE=HSIZE+1
    HLEVEL(HSIZE)=NODE
  ENDIF
25 CONTINUE
* Loop over nodes in shrunken level
* EWIDTH=NC+1
DO 30 I=1,HSIZE
  NODE=HLEVEL(I)
Form rooted level structures for each node in shrunken level

CALL ROOTLS(N,NODE,EWIDTH,E2,ADJ,XADJ,MASK,LS,XLS,DEPTH,WIDTH)

IF(WIDTH.LT.EWIDTH)THEN

Level structure was not aborted during assembly

IF(DEPTH.GT.SDEPTH)THEN

Level structure of greater depth found
Store new starting node, new max depth, and begin
a new iteration

SNODE=NODE
SDEPTH=DEPTH
GOTO 15

ENDIF

Level structure width for this end node is smallest so far
store end node and new min width

ENODE=NODE
EWIDTH=WIDTH
END IF

30 CONTINUE

Generate level structure rooted at end node if necessary

IF(NODE.NE.ENODE)THEN

ENDIF

Store distances of each node from end node

DO 50 I=1,DEPTH
JSTRT=XLS(I)
JSTOP=XLS(I+1)-1
DO 40 J=JSTRT,JSTOP
MASK(LS(J))=I-1
40 CONTINUE
50 CONTINUE
END

SUBROUTINE ROOTLS(N,ROOT,MAXWID,E2,ADJ,XADJ,MASK,LS,XLS,DEPTH,WIDTH)

******************************************************************************

PURPOSE:
--------

Generate rooted level structure using a FORTRAN 77 implementation
of the algorithm given by George and Liu
INPUT:
-----

N - Number of nodes
ROOT - Root node for level structure
MAXWID - Max permissible width of rooted level structure
   - Abort assembly of level structure if width is ge MAXWID
   - Assembly ensured by setting MAXWID = N+1
E2 - Twice the number of edges in the graph = XADJ(N+1)-1
ADJ - Adjacency list for all nodes in graph
   - List of length 2E where E is the number of edges in
     the graph and 2E = XADJ(N+1)-1
XADJ - Index vector for ADJ
   - Nodes adjacent to node I are found in ADJ(J), where
     J = XADJ(I), XADJ(I)+1, ..., XADJ(I+1)-1
   - Degree of node I is XADJ(I+1)-XADJ(I)
MASK - Masking vector for graph
   - Visible nodes have MASK = 0
LS - Undefined
XLS - Undefined
DEPTH - Undefined
WIDTH - Undefined

OUTPUT:
-----

N - Unchanged
ROOT - Unchanged
MAXWID - unchanged
E2 - Unchanged
ADJ - Unchanged
XADJ - Unchanged
MASK - Unchanged
LS - List containing a rooted level structure
   - List of length NC
XLS - Index vector for LS
   - Nodes in level I are found in LS(J), where
     J = XLS(I), XLS(I)+1, ..., XLS(I+1)-1
   - List of max length NC+1
DEPTH - Number of levels in rooted level structure
WIDTH - Width of rooted level structure

NOTE: If WIDTH ge MAXWID then assembly has been aborted

PROGRAMMER: Scott Sloan

LAST MODIFIED: 10 March 1989 Scott Sloan

INITIALISATION

INT INTEGER ROOT, DEPTH, NBR, MAXWID, LSTRT, LSTOP, LWDTH, NODE, NC, WIDTH, N,
+ JSTRT, JSTOP, I, J, E2, XADJ(N+1), ADJ(E2), MASK(N), XLS(N+1), LS(N)

Initialisation
* MASK(ROOT)=1  
  LS(1)=ROOT  
  NC =1  
  WIDTH=1  
  DEPTH=0  
  LSTOP=0  
  LWDTH=1  
10 IF(LWDTH.GT.0)THEN  
  * LWDTH is the width of the current level  
  * LSTRT points to start of current level  
  * LSTOP points to end of current level  
  * NC counts the nodes in component  
  * LSTRT=LSTOP+1  
  LSTOP=NC  
  DEPTH=DEPTH+1  
  XLS(DEPTH)=LSTRT  
  * Generate next level by finding all visible neighbours  
  * of nodes in current level  
  DO 30 I=LSTRT,LSTOP  
    NODE=LS(I)  
    JSTRT=XADJ(NODE)  
    JSTOP=XADJ(NODE+1)-1  
    DO 20 J=JSTRT,JSTOP  
      NBR=ADJ(J)  
      IF(MASK(NBR).EQ.0)THEN  
        NC=NC+1  
        LS(NC)=NBR  
        MASK(NBR)=1  
      END IF  
      CONTINUE  
20 CONTINUE  
30 CONTINUE  
* Compute width of level just assembled and the width of the  
* level structure so far  
* LWDTH=NC-LSTOP  
* WIDTH=MAX(LWDTH,WIDTH)  
* Abort assembly if level structure is too wide  
* IF(WIDTH.GE.MAXWID)GOTO 35  
GOTO 10  
END IF  
XLS(DEPTH+1)=LSTOP+1  
* Reset MASK=0 for nodes in the level structure  
35 CONTINUE  
  DO 40 I=1,NC  
    MASK(LS(I))=0  
40 CONTINUE  
END
SUBROUTINE ISORTI(NL, LIST, NK, KEY)

ORDER A LIST OF INTEGERS IN ASCENDING SEQUENCE OF THEIR KEYS USING INSERTION SORT

INPUT:

NL - Length of LIST
LIST - A list of integers
NK - Length of KEY (NK must be ge NL)
KEY - A list of integer keys

OUTPUT:

NL - Unchanged
LIST - A list of integers sorted in ascending sequence of KEY
NK - Unchanged
KEY - Unchanged

NOTE: Efficient for short lists only (NL <t 20)

PROGRAMMER: Scott Sloan

LAST MODIFIED: 10 March 1989 Scott Sloan
SUBROUTINE NUMBER(N, NC, SNODE, LSTNUM, E2, ADJ, XADJ, S, Q, P)

Number nodes in component of graph for small profile and rms wavefront

INPUT:
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* N - Number of nodes in graph
* NC - Number of nodes in component of graph
* SNODE - Node at which numbering starts
* LSTNUM - Count of nodes which have already been numbered
* E2 - Twice the number of edges in the graph = XADJ(N+1)-1
* ADJ - Adjacency list for all nodes in graph
  - List of length 2E where E is the number of edges in
    the graph and 2E = XADJ(N+1)-1
* XADJ - Index vector for ADJ
  - Nodes adjacent to node I are found in ADJ(J), where
    J = XADJ(I), XADJ(I)+1, .... , XADJ(I+1)-1
* S - List giving the distance of each node in this
  component from the end node
* Q - List of nodes which are in this component
  - Also used to store queue of active or preactive nodes
* P - Undefined

OUTPUT:
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* N - Unchanged
* NC - Unchanged
* SNODE - Unchanged
* LSTNUM - Count of numbered nodes (input value incremented by NC)
* E2 - Unchanged
* ADJ - Unchanged
* XADJ - Unchanged
* S - List of new node numbers
  - New number for node, I is S(I)
* Q - Not used
* P - Not used

NOTES:
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* S also serves as a list giving the status of the nodes
  during the numbering process:
  * S(I) gt 0 indicates node I is postactive
  * S(I) = 0 indicates node I is active
  * S(I) = -1 indicates node I is preactive
  * S(I) = -2 indicates node I is inactive
  * P is used to hold the priorities for each node
PARAMETER(W1=1,
+   W2=2)

Initialise priorities and status for each node in this component
Initial priority = W1*DIST - W2*DEGREE where:
W1 = a positive weight
W2 = a positive weight
DEGREE = initial current degree for node
DIST = distance of node from end node

DO 10 I=1,NC
   NODE=Q(I)
   P(NODE)=W1*S(NODE)-W2*(XADJ(NODE+1)-XADJ(NODE)+1)
   S(NODE)=-2
10 CONTINUE

Insert starting node in queue and assign it a preactive status
NN is the size of queue

NN=1
Q(NN)=SNODE
S(SNODE)=-1

Loop while queue is not empty

30 IF(NN.GT.0)THEN
   Scan queue for node with max priority
   ADDRES=1
   MAXPRT=P(Q(1))
   DO 35 I=2,NN
      PRTY=P(Q(I))
      IF(PRTY.GT.MAXPRT)THEN
         ADDRES=I
         MAXPRT=PRTY
      END IF
   35 CONTINUE

NEXT is the node to be numbered next

NEXT=Q(ADDRES)
Delete node NEXT from queue

Q(ADDRESS) = Q(NN)
NN = NN - 1
ISTRT = XADJ(NEXT)
ISTOP = XADJ(NEXT + 1) - 1
IF(S(NEXT) .EQ. -1) THEN

Node NEXT is preactive, examine its neighbours

DO 50 I = ISTR, ISTOP

* Decrease current degree of neighbour by -1
* NBR = ADJ(I)
  P(NBR) = P(NBR) + W2
* Add neighbour to queue if it is inactive
  assign it a preactive status
* IF(S(NBR) .EQ. -2) THEN
  NN = NN + 1
  Q(NN) = NBR
  S(NBR) = -1
  END IF
  CONTINUE
END IF

Store new node number for node NEXT
Status for node NEXT is now postactive
LSTNUM = LSTNUM + 1
S(NEXT) = LSTNUM

Search for preactive neighbours of node NEXT

DO 80 I = ISTR, ISTOP
  NBR = ADJ(I)
  IF(S(NBR) .EQ. -1) THEN
    * Decrease current degree of preactive neighbour by -1
    assign neighbour an active status
    P(NBR) = P(NBR) + W2
    S(NBR) = 0
  END IF
  Loop over nodes adjacent to preactive neighbour
  JSTR = XADJ(NBR)
  JSTOP = XADJ(NBR + 1) - 1
  DO 60 J = JSTR, JSTOP
    NABOR = ADJ(J)
    * Decrease current degree of adjacent node by -1
    END DO 60
END DO 80
P(NABOR)=P(NABOR)+W2
IF(S(NABOR).EQ.-2)THEN
  * * 
  Insert inactive node in queue with a preactive status
  * *
  NN=NN+1
  Q(NN)=NABOR
  S(NABOR)=-1
END IF
60 CONTINUE
END IF
80 CONTINUE
GOTO 30
END IF
END

SUBROUTINE PROFIL(N,NNN,E2,ADJ,XADJ,OLDPRO,NEWPRO)

Purpose:
Compute the profiles using both original and new node numbers

Input:
N - Total number of nodes in graph
NNN - List of new node numbers for graph
E2 - Twice the number of edges in the graph = XADJ(N+1)-1
ADJ - Adjacency list for all nodes in graph
XADJ - Index vector for ADJ
Nodes adjacent to node I are found in ADJ(J), where
J = XADJ(I), XADJ(I)+1, ..., XADJ(I+1)-1
Degree of node I given by XADJ(I+1)-XADJ(I)
OLDPRO - Undefined
NEWPRO - Undefined

Output:
N - Unchanged
NNN - Unchanged
E2 - Unchanged
ADJ - Unchanged
XADJ - Unchanged
OLDPRO - Profile with original node numbering
NEWPRO - Profile with new node numbering

Note: Profiles include diagonal terms

* PROGRAMMER: Scott Sloan
*----------------
* LAST MODIFIED: 10 March 1989 Scott Sloan
*----------------

** Integer NEWPRO, I, J, N, JSTRT, JSTOP, OLDPRO, NEWMIN, OLDMIN, E2, NNN(N),
+ XADJ(N+1), ADJ(E2)
* Set profiles and loop over each node in graph
* 
OLDPRO=0
NEWPRO=0
DO 20 I=1,N
   JSTRT=XADJ(I)
   JSTOP=XADJ(I+1)-1
   OLDMIN=ADJ(JSTRT)
   NEWMIN=NNN(ADJ(JSTRT))
* 
   Find lowest numbered neighbour of node I
   (using both old and new node numbers)
* 
   DO 10 J=JSTRT+1,JSTOP
      OLDMIN=MIN(OLDMIN,ADJ(J))
      NEWMIN=MIN(NEWMIN,NNN(ADJ(J)))
10 CONTINUE
* 
   Update profiles
* 
   OLDPRO=OLDPRO+DIM(I,OLDMIN)
   NEWPRO=NEWPRO+DIM(NNN(I),NEWMIN)
20 CONTINUE
* 
   Add diagonal terms to profiles
* 
   OLDPRO=OLDPRO+N
   NEWPRO=NEWPRO+N
END

REFERENCES
14. S. W. Sloan and W. S. Ng, 'A direct comparison of three algorithms for reducing profile and wavefront', To appear in *Comp. Struct. (Civil Engineering Report No. 026.07.1988, University of Newcastle).*