An Implementation of Tarjan's Algorithm for the Block Triangularization of a Matrix

I. S. DUFF and J. K. REID
AERE Harwell

An implementation of Tarjan's algorithm for symmetrically permuting a given matrix to block triangular form is described. The discussion includes a flowchart of the algorithm, a complexity analysis, and a comparison with the earlier widely used algorithm of Sargent and Westerberg. Timing results are presented from several experiments using the code developed by the authors.

Key Words and Phrases: block triangular form, sparse matrices, partitioning, reducibility, depth first search
CR Categories: 5.14, 5 25, 5.32, 5.41
The Algorithm: Permutations to Block Triangular Form. *ACM Trans. Math. Software 4*, 2 (June 1978), 189–192.

1. INTRODUCTION

Given an $n \times n$ matrix A with τ nonzeros, a permutation matrix Q is sought, such that

$$QAQ^{T} = \begin{bmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ A_{31} & \cdot & \cdot & \mathrm{O} \\ \cdot & & \cdot & \\ \cdot & & & \cdot \\ A_{N1} & \cdot & \cdot & \cdot \; A_{NN} \end{bmatrix} \tag{1}$$

where the blocks A_{ii} are square and cannot themselves be symmetrically permuted to block triangular form. Duff [2] has shown that, if the matrix A has a zero-free diagonal, then the matrices A_{ii} cannot be reduced to block triangular form even by unsymmetric permutations. In addition, eq. (1) is independent of which nonzeros are on the diagonal apart from permutations within the diagonal blocks themselves and changes in the ordering of the blocks. Matrices with zeros on the diagonal should preferably first be unsymmetrically permuted to have nonzeros on the

Authors' address: Computer Science and Systems Division, Building 8.9, AERE Harwell, Didcot, Oxfordshire, OX11 ORA, England.

diagonal, using Harwell subroutine MC21A (Duff [3]), for example. The form illustrated by eq. (1) is block lower triangular. If the algorithms in this paper are applied to A^T, then the resulting permutations will give a block upper triangular form for A, as used by Duff [2].

Harary [7] first examined the permutation and partitioning eq. (1) and suggested algorithms for obtaining Q which involved computing powers of Boolean matrices. Unfortunately these algorithms involve at least $O(n^3)$ operations and so are prohibitive in computing time for large systems. Steward [12] suggested an algorithm which uses selective OR operations on the Boolean pattern of the rows and was shown by Duff [1] to be typically 10 to 20 times faster than Harary's method. A somewhat better method than Steward's is Sargent and Westerberg's [11], which was found to be on the average twice as fast as Stewards' by Duff [1]. This algorithm, which is described in Section 3, involves $O(n^2)$ operations in adverse cases although some modifications by Tarjan [14] result in an $O(n \log^* n) + O(\tau)$ bound, where τ is the number of off-diagonal nonzeros in A and

$$\log^* n = \min \{i : \overbrace{\log \log \ldots \log}^{i \text{ times}}(n) \leq 1\}.$$

However, Tarjan's algorithm [13] reduces this bound to the optimal $O(n) + O(\tau)$, and it is an implementation of this algorithm (also considered by Gustavson [6]) which is the concern of this paper. Duff [4] presents a fuller account of the history of this problem and surveys the allied question of making the diagonal zero free by unsymmetric permutations.

Relevant terms from graph theory are defined in Section 2 while the algorithms of Sargent and Westerberg and Tarjan are described in Sections 3 and 4, with some comments on their asymptotic behavior. In Section 5 some remarks on the choice of data structure are made, and a complexity analysis and some computer timings are presented. The code analyzed here is that developed by Duff and Reid and is given in this issue [5].

2. GRAPH THEORETIC TERMS

In order to describe the algorithms of Sargent and Westerberg [11] and Tarjan [13], it is very helpful to associate with the matrix A a *directed graph* (hereafter called a *graph*) consisting of a set of n *nodes* (or *vertices*) labeled $1, 2, \ldots, n$ and a set of *edges* (i, j) each of which is an ordered pair of nodes corresponding to an off-diagonal nonzero a_{ij}. Edge (i, j) is said to *leave* node i and *enter* node j. In graph theoretic terminology, these graphs have no multiple edges (that is, no two distinct edges have the same two nodes as *endpoints*), and no self-loops (that is, edges of the form (i, i)). Performing a symmetric permutation of A corresponds to relabeling the nodes of the graph. Figure 1 shows a matrix and its associated graph.

A *path* from node v_1 to node v_k in the graph is a sequence of edges (v_1, v_2), $(v_2, v_3), \ldots, (v_{k-1}, v_k)$ and is called a *cycle* if $v_1 = v_k$. A *subgraph* consists of a subset of nodes and all edges that are pairs of nodes belonging to this subset. A subgraph is said to be *strongly connected* if there is a path from any of its nodes to any other and is a *strong component* if it is strongly connected and cannot be enlarged to another strongly connected subgraph by adding extra nodes and associated edges. Clearly, each node can belong to only one strong component (which may consist of a single node) so the strong components define a partition of the graph. There

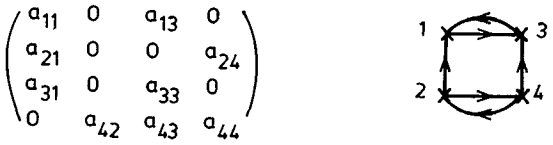

Fig. 1. A matrix and its associated graph

must be at least one strong component such that there is no path from any of its nodes to any node of another strong component. Let this strong component be called C_1. The remaining strong components $C_2, C_3, \ldots, C_k$ may be chosen similarly so that there is no path from any node of one strong component to a node of a strong component later in the sequence.

If these strong components and the ordering shown are identified and the nodes of C_1 labeled before those of C_2 and so on, then the associated matrix is block lower triangular with blocks corresponding to the strong components. Furthermore, because the subgraph associated with each submatrix on the diagonal of the block form is strongly connected, it is apparent that the blocks cannot themselves be permuted to block triangular form.

3. THE ALGORITHM OF SARGENT AND WESTERBERG

The algorithm of Sargent and Westerberg [11] makes use of the fact that all the nodes in any cycle of a graph must lie in the same strong component. A flowchart for this algorithm is given in Figure 2. The algorithm traces paths in the graph associated with A and in modifications of this graph. Starting from any node, a path is followed through the graph until a cycle is found (identified by encountering the same node twice) or a node is encountered with no edges leaving it. All nodes in a cycle must belong in the same strong component and when one is found the graph is modified by collapsing all the nodes in the cycle into a single "composite node"; edges between constituent nodes are ignored and edges entering or leaving constituent nodes from elsewhere in the graph are regarded as entering or leaving the new composite node. If a node (or composite node) is encountered with no edges leaving it, then it must correspond to a strong component of the original graph having no edge to any other strong component. This therefore corresponds to the first block of the required block triangular form. The composite node and its associated edges are then deleted from the graph to leave a graph corresponding to the permuted matrix A with its first block row and column removed. The algorithm continues from the last node on the remaining path or starts from another node if the path has become empty. In this way the blocks of the required form are obtained successively.

The algorithm has the great virtue that each nonzero off-diagonal element (corresponding to an edge in the original graph) has to be inspected only once. Its disadvantage lies in the overheads associated with recording which nodes belong in each composite node and to which composite node each node belongs. Referring to the flowchart in Figure 2, step (1) requires the former information and step (2) the latter. The formation of a new composite node in step (3) requires altering this information. In the example given by Figure 3 successive composite nodes are (4, 5), (3, 4, 5, 6), (2, 3, 4, 5, 6, 7), (1, 2, 3, 4, 5, 6, 7, 8).

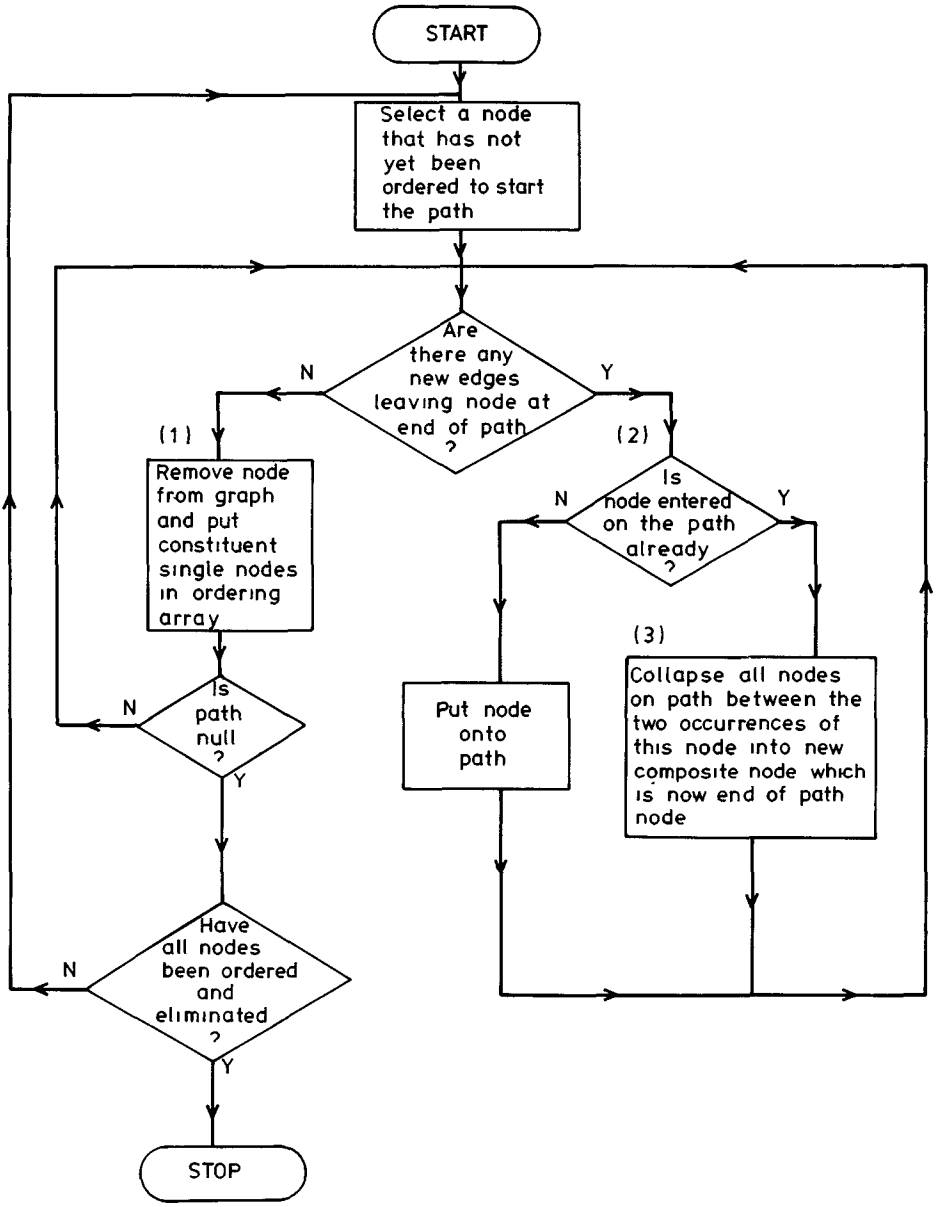

Fig. 2. Flowchart of Sargent and Westerberg's algorithm

A simple scheme such as labeling each composite node with the lowest node number of a constituent node could result in almost every node being identified with n different composite nodes necessitating $O(n^2)$ relabelings. The examples given in Figures 3–6, where n can be arbitrarily large, are likely to show such a behavior if an inferior implementation is adopted. Munro [8] used a common technique in the theory of algorithms when suggesting that successive amalgamations of composite nodes should be performed by relabeling only the one with least con-

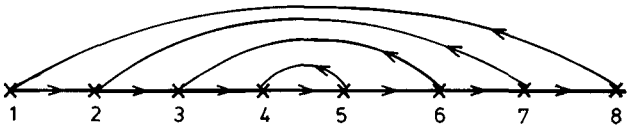

Fig. 3. Concentric semicircles ($n = 2m$, $\tau = 3n/2 - 1$)

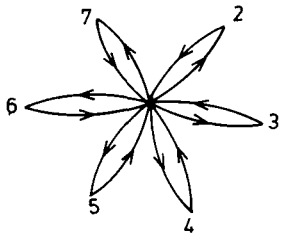

Fig. 4. Star pattern (any n, $\tau = 2n - 2$)

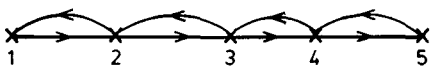

Fig. 5. Graph of tridiagonal matrix (any n, $\tau = 2n - 2$)

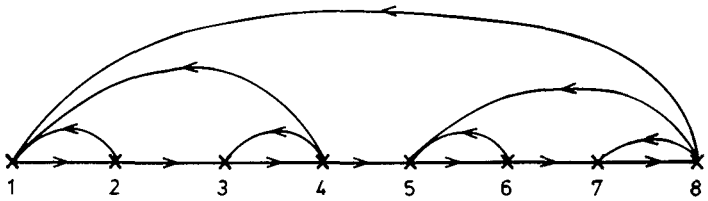

Fig. 6. Munro's example ($n = 2^m$, $\tau = 2n - 2$)

stituent nodes. Munro [9] showed that the resulting algorithm involves at most $O(n \log n) + O(\tau)$ operations and that this bound is attained by the example given in Figure 6. Tarjan [14] holds each composite node as a tree. He amalgamates two composite nodes by assigning the root of one as a son of the root of the other. Finding the composite node to which a given node belongs requires only that the path through successive fathers is followed to the root and much subsequent work is saved if, once the root is found, all nodes on this path are regarded as having the root as father. In this way, Tarjan obtains an algorithm with an $O(n \log^* n) + O(\tau)$ operation count, where $\log^* n$ is as defined in Section 1.

4. TARJAN'S ALGORITHM

Tarjan [13] proposed an algorithm similar to that of Sargent and Westerberg but with the advantage of avoiding the potentially expensive node collapsing steps. The reader may find it helpful to refer to the flowchart in Figure 7 when following the description of this algorithm. Again, paths through the graph are followed, but now a stack of nodes is maintained. This stack holds those nodes which are on the current path or from which a "backtrack" has taken place, and it will be shown that the required strong components can be read successively

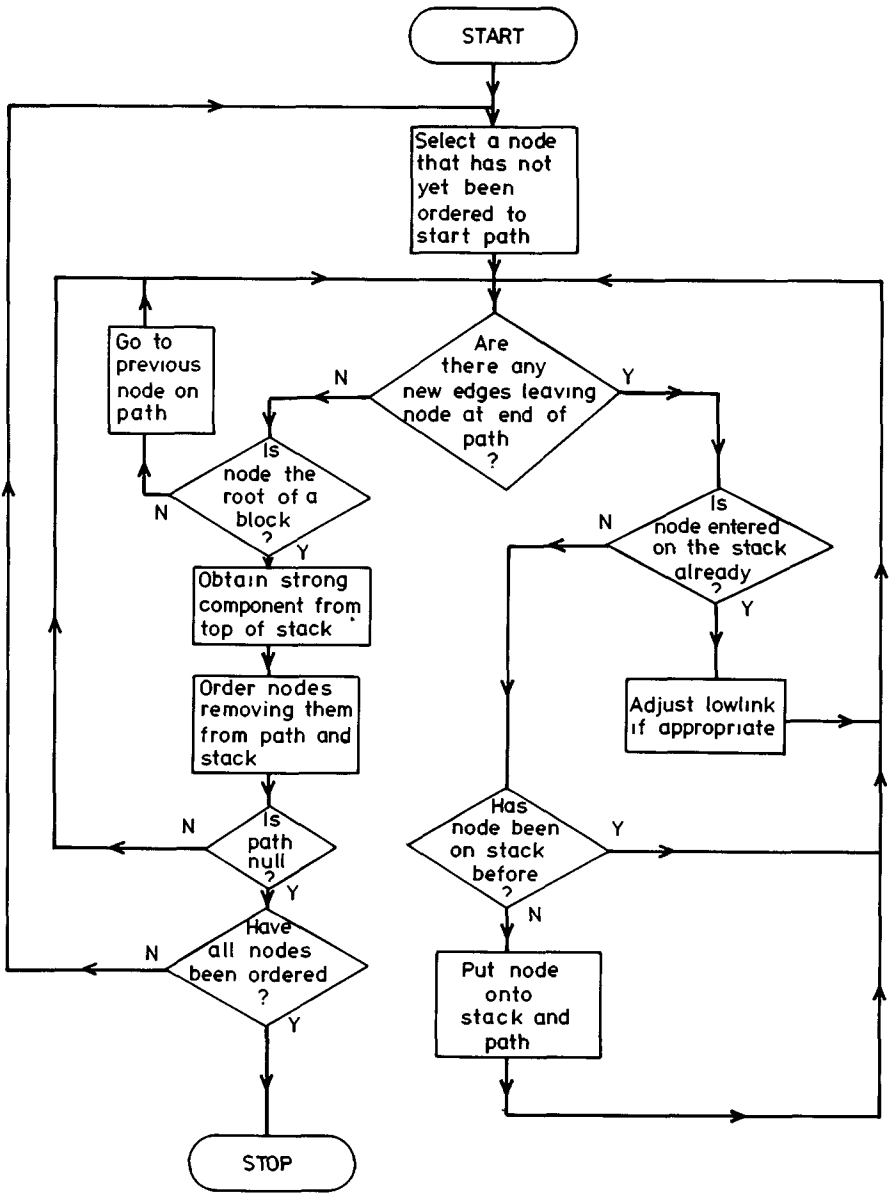

Fig. 7. Flowchart of Tarjan's algorithm

from the top of the stack. Nodes are put on the path and stack when first encountered and are removed from the path but not the stack when backtracking takes place. With each node on the stack a pointer, called lowlink, is held to the node lowest on the stack to which a path has so far been found. Lowlink is initialized to the stack position of the node itself. Note that there may be reachable nodes that are lower in the stack than lowlink.

The algorithm consists of a number (possibly one) of major steps, each comprised of a sequence of minor steps. A major step begins by placing on the stack and on the path any node that has not been on the stack in a previous major step. It then executes a sequence of minor steps, each of which extends the path by one more node or reduces its length by one by backtracking. The major step terminates when the path and stack become empty. A minor step begins with a search of the edges leading from the node v at the end of the path, excluding those previously searched. If an edge points to a node w whose lowlink points lower in the stack than the lowlink of v, then the lowlink of v is reset to the lowlink of w. (In the original algorithm the lowlink of v is reset to point to w if w is lower in the stack than lowlink of v. Our strategy may avoid some assignments to lowlink.) The search continues from the node at the end of the path until either of two possibilities occur:

(1) An edge is found entering a node not on the stack; in this case the node is added to the stack, the path is extended by putting the node at the end of it, and the minor step is complete.

(2) The list of edges leaving the node at the end of the path is exhausted; in this case lowlink is examined and either (a) or (b) occur:

   (a) Lowlink points to the node v currently at the end of the path; in this case v is called the root of a block and v and all nodes above it on the stack consist of the next strong component (to be shown below). They are therefore removed from the stack and from further consideration and are numbered next in the ordering array. The minor step is completed by backtracking to the node previous to v on the path unless the path, and stack, have become empty (ending the major step).

   (b) Lowlink points to a node lower on the stack than the node v currently at the end of the path. In this case the step is completed by backtracking to the node w prior to v on the path. Lowlink for w is reset to that for v if this results in its pointing to a node lower in the stack.

The major step ends when a minor step produces an empty stack, which must eventually happen. If all the nodes have not been ordered another major step is commenced. Eventually the algorithm terminates when all the nodes have been ordered, making it impossible to start another major step.

It remains to show that if lowlink = v in (2)(a) above then v and the nodes above it on the stack constitute a strong component from which there is no connection to any node outside it (apart from those in previously identified strong components). Since no backtracking through v has yet occurred (because it is on the path and can only get there once), v must have been on the path when each node w above v on the stack was placed there. It follows that there is a path from v to every such node w. In addition, backtracking from every such node w must by now have occurred, so there must be a path from every such w to a node below w on the stack; furthermore, there can be no path to a node below v because then lowlink would now point below v. There is therefore a path from v to every node w above it on the stack and back. Finally, there cannot be any connection to any other node so far unordered. This is the required result.

Notice that, as in the Sargent and Westerberg algorithm, each edge is examined

just once. Notice also that each node is placed on the stack just once and is involved in just one backtracking step so the main loop cannot be executed more than $2n$ times. A flowchart is given in Figure 7.

5. FORTRAN IMPLEMENTATION OF TARJAN'S ALGORITHM

It is intended that the Fortran subroutine can be used simply by referring to our paper on the code itself [5]. The purpose of this section of this paper is to explain the choice of data structure and to present complexity results and some computing timings.

All of the edges leaving a node of the graph have to be searched. This corresponds to inspecting the nonzeros in a row of the matrix. It follows that row-by-row access to the positions of the nonzeros is required and the structure chosen is to store the column numbers of each row consecutively in array ICN with array IP being used to point to the beginning of each row. Harwell subroutine MC20A may be used to order a sparse matrix in such a fashion.

Clearly some storage is required to hold information about the stack and the path. The stack is held in an array with a stack pointer which indicates the current position on the stack while the backtracking information related to the current path is held in the linked list PREV(ious vertex). Additionally, arrays are necessary to hold the information required to identify strong components. This is accomplished by having two arrays NUMB(ER) and LOWL(INK). An array giving the final permutation to block triangular form is also required as is an array to indicate the position of the blocks in the final form.

Some reduction in work space has been achieved by making certain integer work vectors share arrays. Since the sum of the number of blocks so far identified and the number of nodes on the stack cannot exceed n, the row numbers of the block starts are held at the beginning of array IB and the list of nodes on the stack at the end of this array. Also the single array NUMB can be used to hold the position in the ordering of those nodes that have been ordered, the position in the stack for the nodes which are on it, and an indication of which nodes have not yet been referenced. In this way the number of work vectors has been reduced to 3. For the user's convenience a single work array of length $3n$ is used, which is divided into three by subroutine MC13D. The real work is performed by subroutine MC13E.

The program is written in ANSI Fortran, as verified by the Bell Laboratories verifier [10]. It includes special comment cards that permit Harwell subroutine OEØ4A to produce an IBM Fortran version that saves storage by using INTEGER*2 arrays. Notice that array IP remains INTEGER so that this IBM version limits to $2^{15} - 1 = 32767$ the order rather than the number of nonzeros. To make the structure apparent and to facilitate complexity analysis, all backward jumps have been avoided.

The complexity of the program is now examined. The loop DO 80 . . . either backtracks from a node (which happens once for each node) or places a node on the stack (which happens for all nodes not placed on the bottom of the stack at the beginning of the loop DO 90 . . .). Therefore the loop DO 80 . . . cannot be commenced more than $2n - 1$ times. Also, because array ARP is used to record how far the search in each row has progressed and because no backtracking occurs

Table I. Complexity of Tarjan Algorithm

Statement Number	Phase or Operation	Theoretical Count	Basic Operations (I) = IF			Profiler Value	
						Fig.3 example n=100	Random n=100 τ=300
350 to 380	Initialization of arrays	n	3	2		100	100
410 to 430	Look for a starting node	s≤n	1		1	1	67
440 to 500	Number of starting nodes required	tree≤s	3	5		1	11
530 to 560	Loop entered whenever backtrack or new node added to stack	2n-tree	1	1	1	199	189
620 to 650	Examination of edge	τ+n	2	1	1	249	400
670	Comparison of low-link	τ-tree	2	1	1	150	311
760 to 780	Ordering nodes in a block	≥tree,≤n		3		1	15
810 to 860	Putting node on ordering array	n	3	4	1	100	100
970 to 1010	Backtrack	n-tree	3	3	1	99	89
1040 to 1110	Putting new node on stack	n-tree	5	8		99	89
1180 to 1210	Putting permutation in required form	n	2	2		100	100

until the row is exhausted, the loop DO 20 . . . is commenced exactly τ times (where τ is the number of nonzeros). The slightly unusual definition of ARP is made in order to let it have values less than N, to reduce the number of statements in the inner loop, and to facilitate checking for exhausted rows. The array ARP is also used to output the permutation in its desired final form. These considerations lead to the detailed complexity results shown in Table I. To assist in determining the computational complexity, three columns on basic operations have been included which refer to array look-ups, assignments, and decisions, respectively. They have been verified by several runs of the program when modified by the Harwell profiling subroutine OEØ2A, two such results are given in Table I.

These results were further confirmed by timing runs on test examples, some of which are shown in Table II. The random matrices were generated as block triangular forms of varying block sizes and number, the actual positions of the nonzeros within each block being determined by a pseudo-random number generator. These matrices then had their rows and columns permuted in a similar fashion before entry to the subroutine. In all cases the times were better than those obtained with an earlier Sargent and Westerberg code, the current code being typically twice as fast as the earlier one. Other advantages of the present code are that it is much less complicated, is about half as long, and requires about half as much work space.

The Tarjan algorithm has thus been implemented for use on a sparse matrix

Table II. Execution Times for the 370/168 (in milliseconds)

		Order, n	No.of off-diagonal non-zeros, τ	Time
Fig.3.	Concentric semi-circles	200	3n/2-1	7
		400		13
		600		20
		800		23
Fig.4.	Star pattern	200	2n-2	10
		400		17
		600		20
		800		27
Fig.5.	Tridiagonal	200	2n-2	7
		400		13
		600		20
		800		23
Fig.6.	Munro	512	2n-2	17
		1024		30
		2048		60
		4096		123
	Random	100	300	7
		100	700	7
		100	900	7
		100	1900	10
		200	200	10
		200	600	10
		200	800	10
		200	1800	13

data structure to find the permutations to block triangular form. Although we appreciate that any analysis of computation time or complexity is very machine and compiler dependent and can only be regarded as approximate, we feel it is useful to give an indication of the total work involved. By counting the number of operations performed in the whole subroutine (noting that only the principal parts are included in Table I) we arrive at a bound on the computation time t of

$$t \leq (2\text{EQ} + 4\text{ARRAY} + 2\text{IF})(\tau + n) + (30\text{EQ} + 28\text{ARRAY} + 6\text{IF})\, n + 8\text{EQ}$$

where EQ, ARRAY, and IF stand for units of time for assignments, array look-ups, and decision statements, respectively.

ACKNOWLEDGMENT

We would like to acknowledge the help of F.G. Gustavson in drawing our attention to the advantages of Tarjan's approach and in discussing the detailed design of the code.

REFERENCES

1. Duff, I S. Analysis of sparse systems. D.Phil. Th., Oxford U., Oxford, 1972.
2. Duff, I.S. On permutations to block triangular form. *J. Inst. Math. Appl. 19* (1977), 339-342.
3. Duff, I.S. On algorithms for obtaining a maximal transversal. To appear.
4. Duff, I.S. A survey of sparse matrix research. *Proc. IEEE 65* (1977), 500-535.
5. Duff, I.S., and Reid, J.K. Algorithm 529. Permutations to block triangular form. *ACM Trans. Math. Software 4*, 2 (June 1978), 189-192.

6. GUSTAVSON, F.G. Finding the block lower triangular form of a sparse matrix. In *Sparse Matrix Computations*, Bunch and Rose, Eds., Academic Press, New York, 1976.
7. HARARY, F. A graph theoretic method for the complete reduction of a matrix with a view toward finding the eigenvalues. *J. Math. Phys. 38*, (1959), 104–111.
8. MUNRO, I. Efficient determination of the transitive closure of a directed graph. *Inform. Proc. Lett. 1*, (1971), 56–58.
9. MUNRO, I. Some results in the study of algorithms. Tech. Rep 32, Dept. of Comptr. Sci., U. of Toronto, Toronto, 1971.
10. RYDER, B.G. The Fortran verifier: user's guide. Comptg. Sci. Tech. Rep. 12, Bell Telephone Labs., Murray Hill, N.J., 1973.
11. SARGENT, R.W.H., AND WESTERBERG, A.W. "Speed-up" in chemical engineering design. *Trans. Inst. Chem. Engrs. 42*, (1964), 190–197.
12. STEWARD, D.V Partitioning and tearing systems of equations. *SIAM J. Numer. Anal. 2*, (1965), 345–365.
13. TARJAN, R.E. Depth first search and linear graph algorithms. *SIAM J. Comptg. 1*, (1972), 146–160.
14. TARJAN, R.E. Efficiency of a good but not linear set union algorithm. *J. ACM. 22*, 2 (April 1975), 215–225.

Received April 1976; revised August 1976