On Algorithms for Obtaining a Maximum Transversal

I. S. DUFF
AERE Harwell, England

A computer program for obtaining a permutation of a general sparse matrix that places a maximum number of nonzero elements on its diagonal is described. The history of this problem and the main motivation for developing the algorithm are briefly discussed Comments are made on the use of cheap heuristics, and the complexity of the algorithm is examined both in terms of its worst case bound and its performance on typical examples. Finally, some initial attempts at implementing an algorithm with superior asymptotic complexity are described.

Key Words and Phrases unsymmetric permutations, maximum transversal, maximum assignment, block-triangular form, sparse matrices
CR Categories 5.0, 5 1, 5.3, 5.4
The Algorithm. Permutations for a Zero-Free Diagonal, *ACM Trans. Math Softw* 7, 3 (Sept. 1981), 387-390.

1. INTRODUCTION

There is a long history to the problem of computing permutations to place the maximum number of nonzeros on the diagonal of a sparse matrix. In this paper we give only principal references; further references are given in [5, sec. VIII E]. Although we indirectly discuss some of this history in Sections 2 and 6, the principal objective of this paper is to describe a computer program, Harwell subroutine MC21A, for computing such a permutation. We examine the complexity of the algorithm both in terms of its worst case bound and its performance on real and simulated problems and compare it with other approaches.

In Section 2 we give some indication of the wide range of application areas for such an algorithm and additionally introduce other terms commonly used to describe this ordering problem. Our main motivation for developing this algorithm is to obtain fast, reliable techniques for preordering a matrix to block-triangular form prior to its decomposition using Gaussian elimination [4]. We discuss this in Section 3.

The algorithm MC21A given in [6] is described in Section 4. Although MC21A was initially developed independently from Gustavson's ASSIGN ROW [11], both are related to an algorithm by Hall [12]. We have since incorporated two of

Author's address· Computer Science and Systems Division, Building 8.9, AERE Harwell, Oxon, England.

Gustavson's suggestions so that our resulting algorithm has similar features to an efficient implementation of ASSIGN ROW.

In Section 5 comments are made on the complexity of the algorithm. Throughout this paper we will use n and τ to denote the order of the matrix and its number of nonzeros, respectively. An example that causes the algorithm to exhibit its worst case bound of $O(n\tau)$, is described, although the behavior on more realistic examples is seen to be more like $O(n) + O(\tau)$. Its speed is seen to compare favorably with the other algorithms with which it is used in our block-triangularization context. In Section 6 we comment on other algorithmic approaches to this problem. In particular we compare our algorithm with a version of the Hopcroft and Karp algorithm [15] that has worst case complexity only $O(n^{1/2}\tau)$. Finally, in Section 7 we make some comments on the use of heuristics to reduce computing time.

2. OTHER FORMULATIONS OF THE PROBLEM

Since the problem of obtaining the permutation described in the introduction has appeared in various guises in many areas, we feel it is useful to devote a whole section to describing the terms we will use later in this paper, as well as the different formulation and terminologies used in some of these other areas. We stress that this section does not attempt to be exhaustive but rather is included to give the reader a flavor of this topic.

Most of our nomenclature is taken from management science (for example, Kettler and Weil [16]).

For example, we refer to the set of nonzeros on the diagonal of the permuted matrix as a *transversal*. Thus a transversal is characterized as a set of nonzeros, no two of which lie in the same row or column. We refer to the number of nonzeros in a transversal as the *length of the transversal*, and call a set containing the maximum number of such nonzeros a *maximum transversal*. When we include a nonzero in the transversal, we call it an *assignment* and, if we change the transversal by also removing nonzeros, we have performed a *reassignment*. Ford and Fulkerson [10] refer to a maximum transversal as a *maximum assignment*. We find it helpful during the remainder of this section to refer to the simple 3×3 example shown in Figure 1. In this figure, the circled elements are the (in this case, unique) maximum transversal.

Transversal selection has for many years been studied by researchers in combinatorics. If we have a set of n subsets of some universal set S, say, then a *system of distinct representatives (SDR)* is a subset of n distinct elements of S, each of which represents exactly one of the given subsets and belongs to the subset it represents. In the example of Figure 1, $S = \{1, 2, 3\}$ and the n (here equal to 3) subsets could consist of the column indices of nonzeros in each row and so might be denoted by $\{1, 3\}$, $\{2\}$, $\{1\}$. The SDR $\{3, 2, 1\}$ then would correspond to the selected transversal.

P. Hall [13] gave the following necessary and sufficient condition for an SDR to exist. "An SDR exists if and only if the union of any k subsets $(1 \leq k \leq n)$ contains at least k distinct elements." Although M. Hall [12] proposed an algorithm for obtaining an SDR, he gave little indication of its implementation as a computer program.

$$\begin{bmatrix} x & & \textcircled{x} \\ & \textcircled{x} & \\ \textcircled{x} & & \end{bmatrix}$$

Fig 1. Matrix indicating traversal

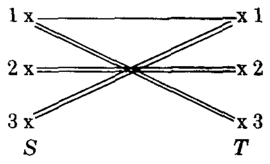

Fig 2 Bigraph of matrix of Figure 1

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$$

Fig. 3. Matrix representation of undirected graph obtained from Figure 2

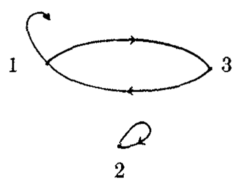

Fig.-4 Directed graph of matrix of Figure 1

Steward [20, 21] calls such an SDR *an output set*, a term commonly employed in engineering design or chemical engineering. The term is used because the chosen output set designates for which variable each equation is solved, the other variables being termed input variables.

The structure of a sparse matrix can be represented as a graph in several ways (see, for example, Harary [14]). We show, in Figure 2, the *bipartite graph (bigraph)* associated with our example of Figure 1. The vertices of the bigraph are divided into two sets S and T, those in S representing rows and those in T columns. An edge exists from vertex s_i in S to vertex t_j in T if and only if the nonzero (s_i, t_j) is present in the matrix. The edges of the bigraph corresponding to nonzeros in the maximum transversal are indicated by double lines. In such a context, the maximum transversal is called a *maximum or perfect matching.* The problem of finding a maximum matching in a bipartite graph has been studied by Dulmage and Mendelsohn [8, 9].

If, in Figure 2, we ignore the distinction between sets S and T and renumber the vertices 1, 2, 3 of T to be 4, 5, and 6, respectively, we obtain the undirected graph associated with the symmetric matrix (Figure 3), where A is the matrix in Figure 1.

A general sparse matrix can also be represented by a directed graph, where an edge exists from vertex i to vertex j if and only if the element (i, j) is nonzero. The directed graph corresponding to the matrix of Figure 1 is shown in Figure 4. Finding a maximum transversal is equivalent to finding a factor where a *factor of a directed graph* is defined as a set of edges of the graph such that each vertex has one edge of the set entering it and one leaving it. Although such a factor can be easily found in the example of Figure 4, the algorithm suggested by Berge [1]

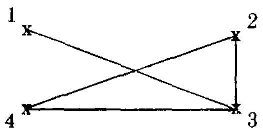

Fig 5. Graph for the rooks' problem.

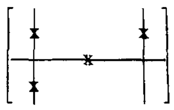

Fig 6. Lines showing minimum cover

requires the solution of two sets of n Boolean equations in τ Boolean variables when the original matrix is of order n with τ nonzeros.

Notice that we can similarly define a factor for an undirected graph, in which case our problem is equivalent to finding a factor of the graph associated with the matrix of Figure 3.

Another graphical interpretation is obtained from the rooks' problem. The *rooks' problem* is a popular puzzle where one is required to place 8 rooks (or castles) on a chess board such that no rook can take any other. The more general problem allows the chess board to have n squares in each direction and restricts the squares on which rooks may be placed. This can be seen to be identical to our maximum transversal problem. The rooks' problem is often rephrased in the following way. We form an undirected graph on τ vertices (τ is the number of permitted locations for the rook) where each edge corresponds to the fact that its endpoints do not lie on the same row or column of the original board. In the graph thus obtained (Figure 5), we wish to find a complete subgraph (a subgraph whose vertices are all pairwise adjacent) with the maximum possible number of vertices. For our example of Figure 1, the graph on τ (=4) vertices has the form shown in Figure 5, where the complete subgraph on vertices 2, 3, 4 yields the maximum transversal. However, since we have rephrased the problem to that of finding a complete subgraph on n vertices from a graph on τ vertices with $\frac{1}{2}(\tau^2 - 2\tau n + \tau)$ edges, any algorithms based on such an approach are clearly $O(\tau^2) + O(n\tau)$ at best and will usually be worse.

Finally, a related topic is that of obtaining a *minimum cover* of a matrix. This minimum number of lines (that is, rows or columns) necessary to include all the nonzeros of a matrix has been shown by Konig [17] to be equal to the length of a maximum transversal. Of course, in the nonsingular case, the minimum cover can consist trivially of all rows or all columns of the matrix, but we show in Figure 6 a different minimum cover for our given example.

3. TRANSVERSAL SELECTION AND BLOCK TRIANGULARIZATION

Our principal motivation for developing a transversal algorithm has been its application in the first step of a method for permuting an arbitrary sparse matrix to block-triangular form. This form is illustrated in Figure 7, where each A_{ii} is square and cannot itself be permuted to block-triangular form.

It is evident that the solution of a system whose coefficient matrix can be thus preordered is greatly simplified. Indeed, if we are using Gaussian elimination to

$$\begin{pmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ \vdots & \vdots & \ddots & \\ A_{N1} & A_{N2} & \cdots & A_{NN} \end{pmatrix}$$

Fig 7 Block-triangular form

Fig 8 Irreducible, bireducible matrix $\begin{pmatrix} 0 & x \\ x & 0 \end{pmatrix}$

solve such a system, the forward elimination phase can be confined to the blocks on the diagonal, A_{ii}. Thus the system is solved as a sequence of subsystems with coefficient matrices A_{ii}, $i = 1, 2, \ldots, N$.

The approach we adopt for obtaining such a form is to first permute the matrix so that the diagonal is zero-free and then use symmetric permutations to produce the desired form. The author [4] has justified this two-stage approach and has shown that the final form is essentially independent of the first permutation. The author and Reid [7] have developed code, based on the work of Tarjan [22] and Gustavson [11], which performs the second phase in $O(n) + O(\tau)$ time. Wiberg [23] has suggested combining these two phases and intends to compare his algorithms with ours in the near future.

A matrix, which may have zeros on its diagonal, is *bireducible* if there exist row and column permutations that permute it to block-triangular form, as in Figure 7, and is *reducible* if this can be done using symmetric permutations (i.e., rows and columns permuted identically). The negation of these terms, bi-irreducible and irreducible, respectively, are defined in the obvious way. The author [4] has shown the equivalence of these terms when the matrix has a zero-free diagonal.

If we omit the first stage, then, as the example of Figure 8 illustrates, we will not be able to order an irreducible, bireducible matrix to block-triangular form. However, the application of the second phase to a matrix with zeros on the diagonal may give a partial decomposition in the sense that further unsymmetric permutations within a block may further decompose it. In other words, the presence of zeros on the diagonal may give us a more grainy decomposition than otherwise possible, although the block structure includes the possibility of finer decomposition.

4. ALGORITHM MC21A

In this section we first describe the basic techniques of our algorithm, making use of some terminology from graph theory. We then present a flowchart of the algorithm and give some details of our implementation. The FORTRAN code is given in [6].

This algorithm is usually described in terms of a bipartite graph [15], but we believe that our presentation is made clearer by using a less common graphical representation, which we now describe. The transversal is constructed in n major steps, after the kth of which we have a transversal for a submatrix of order k.

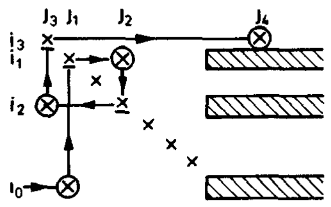
Fig. 9. Reassignment chain.

After the kth step, we associate with the matrix an unconventional directed graph (which usually changes from step to step). Each vertex of the graph corresponds to a row of the matrix, and there is an edge from vertex i_0 to vertex i_1 if there exists a column of the matrix, j_1 say, such that nonzero (i_1, j_1) is a current transversal element and element (i_0, j_1) is nonzero. We say we can reach vertex i_1 from vertex i_0 and define a path to be a sequence of edges of this kind. It is helpful to consider a path, from i_0 to i_k, say, as a sequence of nonzeros (i_0, j_1), $(i_1, j_2), \ldots, (i_{k-1}, j_k)$ where the present transversal includes the nonzeros (i_1, j_1), $(i_2, j_2), \ldots, (i_k, j_k)$. Now if there is a nonzero in position (i_k, j_{k+1}) and if no nonzero in row i_0 or column j_{k+1} is currently on the transversal, then the length of the transversal can be increased by 1 by removing nonzeros (i_r, j_r), $r = 1, \ldots, k$, from the transversal and adding nonzeros (i_r, j_{r+1}), $r = 0, 1, \ldots, k$, to it. In Figure 9 we illustrate this reassignment chain (called an augmenting path by Hopcroft and Karp [15]) on the matrix representation. We simplify our illustration by assuming, without loss of generality, that a column permutation has been performed so that the previously assigned rows have their transversal element on the diagonal. We comment on the shaded regions of this example later in the text.

The reassignment chain is shown by directed lines in Figure 9, the vertical lines from nonzeros (i_r, j_{r+1}) to nonzeros (i_r, j_r), and horizontal ones from (i_r, j_r) to (i_r, j_{r+1}). The reassignment corresponds to replacing the three underlined transversal elements by the four circled nonzeros (i.e., k in our earlier discussion is equal to 3).

There are several different techniques available for finding a reassignment chain. We use a depth first search with look-ahead technique that we now describe.

In accessing the vertices of a graph in a *depth first search* (DFS), we search edges from the current vertex and add to our path the first vertex encountered that we have not yet visited. This becomes the current vertex and we proceed from it as before. If all the vertices that can be reached from the current one at the end of the path are already visited, we retrace our steps (or backtrack) to the vertex added to the path immediately before this present one, make that the current vertex, and proceed as before. We illustrate the depth first search algorithm in Figure 10, where heavy lines denote edges in the path and the vertices are numbered in the order in which they are visited during the DFS. Clearly, when implementing a DFS we need to keep track of the path (required for backtracking) and also know which vertices have been visited during the search.

In the present context we define our edges, as before, to be of the form (i_1, i_2) where (i_1, j_2), say, is a nonzero and (i_2, j_2) a present assignment. We start from

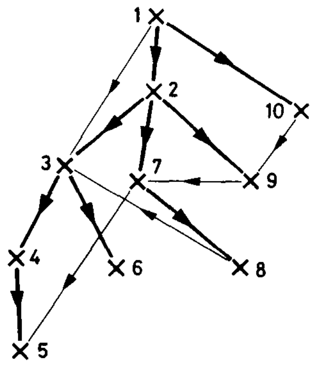

Fig. 10 Directed graph indicating depth first search ordering

any unassigned vertex (row) i_0 (in practice, this will be the next row in the matrix) and trace a path using a DFS technique until a vertex (row) i_k is reached where the path terminates because nonzero (i_k, j_{k+1}) exists and j_{k+1} is an unassigned column. We call vertex i_k a free vertex. If we then retrace our path from i_k to i_0 using the path information used in backtracking, we recover our reassignment chain (or augmenting path).

In practice, such an algorithm might be very inefficient because the DFS scheme does not specify which unvisited vertex reached from the present current vertex should be added to the path and such a choice could be quite critical. We alleviate this problem, at each stage, by checking all unvisited vertices reached from the current one to see if any are free. If so, our reassignment chain has been found and we can extend the transversal by one. We say we have found a *cheap assignment* and call this part of the algorithm the *look-ahead technique*. We therefore see that if DFS with look ahead was used to create the assignment chain in Figure 9, then the shaded areas must contain all zeros or a shorter chain would have been found.

Naturally, this requires a little more work at each stage, but if we implement this efficiently (see use of array ARP, below), then the total cost of all the look aheads is only $O(\tau)$, where τ is the number of nonzeros in the matrix. We see in the next section that the shortening of the length of the reassignment more than compensates for this small amount of extra computation.

We summarize this description of our algorithm in the flow diagram in Figure 11. The labels on the symbols in the flowchart are used in the complexity investigations of the next section.

We now discuss some of the details of our implementation. The reader may find it helpful to refer to the code in [6]. On Gustavson's suggestion, we introduce an auxiliary pointer array for the cheap assignment phase (ACP) as well as for the depth first search. This ensures that the total number of nonzeros accessed during cheap-assignment searches is τ. Gustavson [11] indicated that the number of accesses could be much larger when this was not done; for example, on a matrix of order 199 with 701 nonzeros, the accesses increased from 661 to 1706. However, we adopt the same policy as in [7], and use ACP to denote one less than the number of unsearched nonzeros in a particular row (the same is done for the auxiliary array OUT during the depth first search). This facilitates checking for

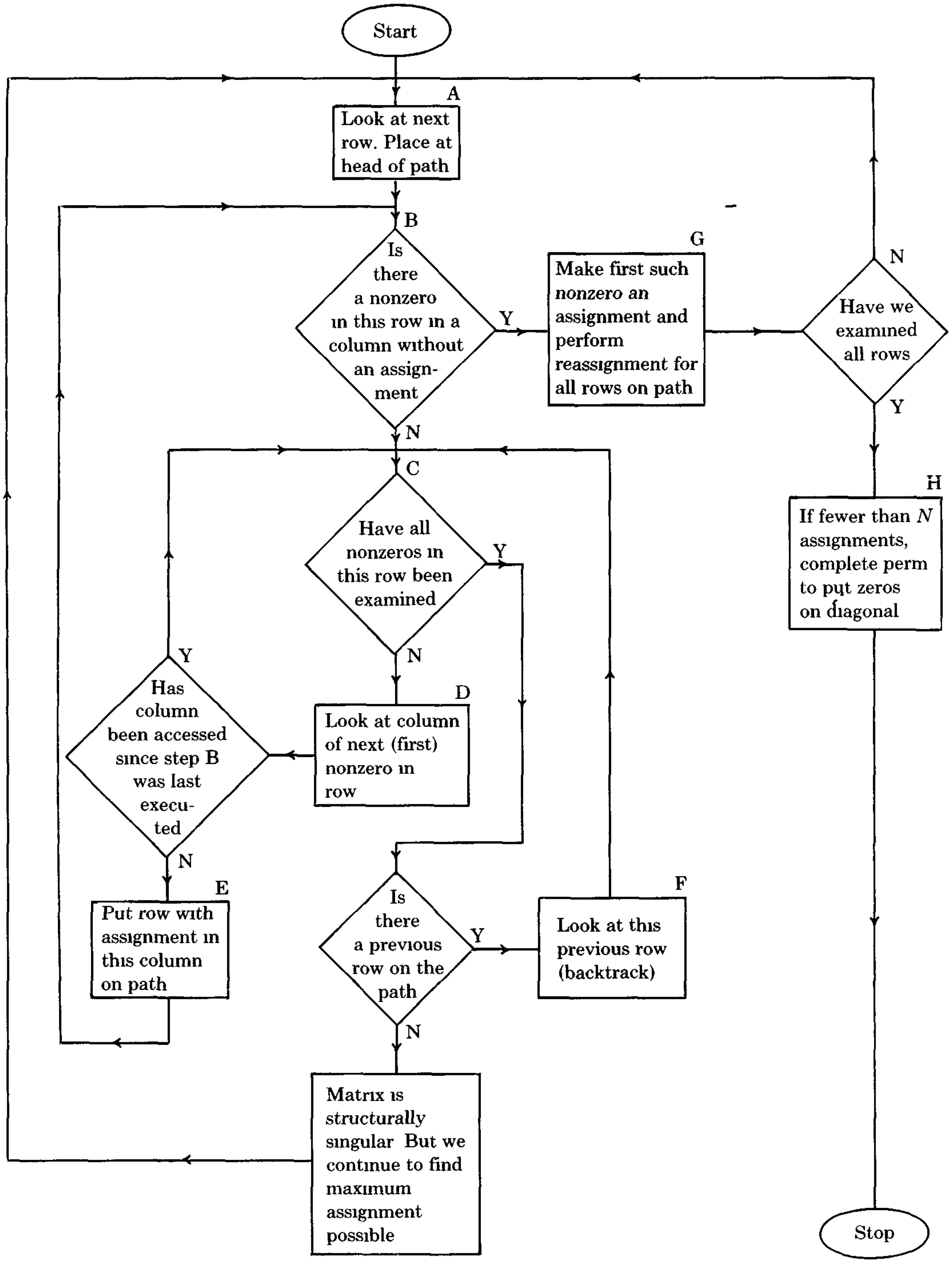
Start
A
Look at next row. Place at head of path
B
Is there a nonzero in this row in a column without an assignment
Y
N
G
Make first such nonzero an assignment and perform reassignment for all rows on path
Have we examined all rows
N
Y
H
If fewer than N assignments, complete perm to put zeros on diagonal
Stop
C
Have all nonzeros in this row been examined
Y
N
D
Look at column of next (first) nonzero in row
Has column been accessed since step B was last executed
Y
N
E
Put row with assignment in this column on path
Is there a previous row on the path
Y
N
F
Look at this previous row (backtrack)
Matrix is structurally singular But we continue to find maximum assignment possible

Fig. 11 Flowchart for MC21A

exhausted rows and reduces by half the number of statements in the inner loop. ACP need only be set once at the beginning of the subroutine and updated whenever a cheap assignment is made or a row exhausted. Notice that we only set OUT when a particular row enters the depth first search and that, as in Gustavson [11], we indicate previous paths in the present depth first search by setting CV equal to the number of the assignment being made to avoid the resetting of this array. On the 199 example, Gustavson [11] shows that this saves 8000 operations. The path in the depth first search is held in the linked list PR; this never need be reset, is unchanged during backtracking, and is only altered when a new row is added during the forward scan. The algorithm will work on structurally singular matrices (including ones with null rows or columns), in which case the permutation is completed in a postprocessing phase for which the number of operations is a small multiple of n.

5. COMPLEXITY OF ALGORITHM MC21A

When studying the complexity of our algorithm, we will refer to the flowchart given in Figure 11. In this flowchart, we have labeled the symbols that are critical to an estimation of the complexity of the algorithm by the letters A to H. All represent simple operations, except G and H, but we have already explained that the total number of operations in H is a small multiple of n. At the very worst we must, for each of our n new assignments, reassign all the previous assignments and the work involved in G may be $O(n^2)$. Since the flow of control is like an electrical circuit, an application of Kirchhoff's second law indicates that we now need only examine B and C/D to determine the complexity of the algorithm as a whole.

In the last section, we remarked that our use of the array ACP would limit the number of executions of B to τ. If we could avoid searching an edge of the graph more than once, then we could restrict the number of times C/D is executed to τ also. However, the example in Figure 12 illustrates that our algorithm may need to scan the same edge more than once. Here edge (1, 2) is scanned when making both the fourth and fifth assignment. If each edge were scanned on every assignment, then step D and hence our algorithm would perform as $O(n\tau)$. An example for which MC21A exhibits this behavior is given in Figure 13, where the blocks are of order $n/3$.

We show, in the results of Table I, that this behavior is reflected in the computation times for MC21A when run on matrices of type NCUBE of different orders. Particularly at higher n, when the low-order counts lose significance, the $O(n\tau)$ behavior of the algorithm is evident.

However, we must emphasize that this is a worst case example and we see from the runs on random matrices in Tables II and III that the behavior is nowhere near as bad as the matrices NCUBE suggest.

In these tables, we also give times for a straightforward application of a depth first search algorithm that does not involve a look ahead (DFSA). In Table II, it is interesting to observe that the time does not increase monotonically with τ and takes its maximum value at $\tau = 300$ and the same lowest value for $\tau = 100$ and $\tau = 500$. This is not that surprising since, although we would expect the algorithm to be very fast on a permutation of the identity matrix, when the number of

Fig. 12 Example requiring two scans of the same edge.

$$\begin{bmatrix} x & x & & & \\ & x & & x & \\ & & x & & x \\ x & & x & & \\ & x & & & \end{bmatrix}$$

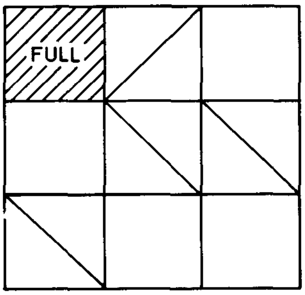

Fig. 13. Example NCUBE

Table I. Times for MC21A on NCUBE

n	τ	Time (milliseconds on IBM 370/168)	Time $\times 10^3/n\tau$
120	1760	75	0.36
240	6720	440	0.27
360	14880	1327	0.25
480	26240	2914	0.23
600	40800	5503	0.22
660	49280	7206	0.22
720	58560	9232	0.22
780	68640	11624	0.22

nonzeros increases above a certain point (in practice about five nonzeros per row), it becomes increasingly easy to obtain quick cheap assignments and any further increase in time is simply due to more nonzeros being scanned during the cheap-assignment phase. Notice that the DFSA algorithm does increasingly badly because the extra number of nonzeros does not help in getting cheap assignments since we do not look ahead for them. Indeed, they cause more paths to be searched and the time is proportional to τ.

In Table III, we hold the number of nonzeros per row constant because we have found this to be the other main parameter in the times. Here we see that the behavior is much better than proportional to n^2, a much more satisfactory performance than could be expected from the upper bounds. The algorithm DFSA performs much worse, largely because it is more affected by the increasing total number of nonzeros.

We will see later that this behavior on random systems is reflected in the performance on systems arising in practice, thus indicating that the worst case bound is very unlikely to be realized.

Table II. *Behavior with Respect to Number of Nonzeros*[a]

Number of nonzeros, τ	100	150	200	300	400	500
MC21A	1.3	1.9	2.6	2.8	1.4	1.3
DFSA (no look ahead)	1.8	3.5	7.1	12.9	16.3	21.9

[a] Times in milliseconds on IBM 370/168 (order = 50).

Table III Behavior with Respect to Order[a]

Order, n	25	50	100	150	200
MC21A	0.6	1.9	4.9	8.7	12.4
DFSA	1.4	3.5	14.4	27.6	40.3

[a] Times in milliseconds on an IBM 370/168 (three nonzeros per row).

Table IV. Complexity of MC21A

		Number of times group of statements executed			
				Actual execution counts $n = 60$	
Block in Figure 11 flowchart	Number FORTRAN statements in block	Bound	Value on NCUBE $s = n/3$ $\tau = s(s + 4)$	NCUBE $\tau = 480$	RANDOM $\tau = 480$
A	2	n	n	60	60
B	2	τ	$s^2 + 4s$	480	290
C	2	n^2	$\frac{1}{2}s(3s + 1)$	610	6
D	2	$n\tau$	$\frac{1}{6}s(s + 1)(2s + 7)$	3290	6
E	6	n^2	$s(s + 1)$	420	6
F	2	n^2	$\frac{1}{2}s(s - 1)$	190	0
G	5	n^2	$\frac{1}{2}s(s + 3)$	230	60
H	3	n	0	0	0

In Table IV, we illustrate the complexity of the algorithm, where we again observe that the $O(n\tau)$ behavior can only come from the accessing of the edges during the forward pass of the depth first search. Although $O(n^2)$ counts abound (and some are realized by NCUBE), we see from the execution counts on a random matrix of the same order and density that it is only the very particular structure of NCUBE that causes the bound to be achieved.

When discussing the performance and complexity of our algorithm, it is important to examine it in the context of the other algorithms with which it will be used. We are therefore interested in comparing its behavior with that of a subroutine for block triangularization and of subroutines for the decomposition of sparse systems by Gaussian elimination. In practice the times of these algorithms are approximately proportional to $n + \tau$ and τ^2/n, respectively, so the $O(n\tau)$ performance of our algorithm on NCUBE gives us some cause for concern.

Table V. Comparative Timings of Transversal Selection with Block-Triangularization and Gaussian Elimination[a]

Matrix:	n	147	199	292	822
	τ	2449	701	2208	4841
Transversal selection		4.3	10.4	6.2	154.7
Block triangularization		10.1	7.2	13.1	36.6
Gaussian elimination (using Harwell subroutine MA30A)		1318.2	203.2	761.6	1426.4

[a] Times in milliseconds on an IBM 370/168.

However, we are primarily concerned with the performance of our algorithm on genuine problems and, when we examine the results in Table V, we observe that not only is the transversal selection sometimes faster than the asymptotically superior block-triangularization routines, but the combined steps take very little time in comparison with that for the decomposition of the matrix.

Because of our remarks in the Introduction, we would expect a coded version of ASSIGN ROW to perform very similarly to MC21A, but Gustavson [11] gives no exact implementation details, no code, and no timings for his algorithm.

6. COMMENTS ON OTHER ALGORITHMS

In Section 2 we gave a brief description of several of the areas in which the problem of transversal selection arises. Naturally, a variety of algorithms have been suggested by some of the researchers in these areas and, in this section, we comment on some of these.

In our description of MC21A, we defined a path and the depth first search technique for searching the vertices and edges of a graph. Another commonly employed search technique is that of *breadth first search.* Here, we explore *all* the edges leaving the current vertex, adding any unvisited vertices reached from this vertex onto a stack before processing the next vertex on the stack. We illustrate this technique by examining its application on the graph in Figure 14. The graph in Figure 14 is the same as that in Figure 10 and, additionally, the vertices are in the same position, but they are now numbered in the order accessed during a breadth first search. Notice that one property of a breadth first search is that it will always find the shortest path between the starting vertex and any other. Clearly this will enable us to get short reassignment chains (augmenting paths), but the total number of edges visited before finding such a chain may well be greater than in the depth first search method. Observe that our "look ahead" (Section 4) is really one step of a breadth first search. We compare our algorithm with one based on a breadth first search technique in Table VI.

In 1957 Kuhn [19] examined some of the algorithms used up to that time. We continue his classification and rephrase such comments in the language of this paper.

The Hungarian algorithm of Kuhn [18] was similar to ours inasmuch as he used a depth first search technique. However, it differed in so far as he did not attempt to do a cheap assignment at each stage. By examining the simple example in Figure 15, we can see that this "look-ahead" capacity is very important.

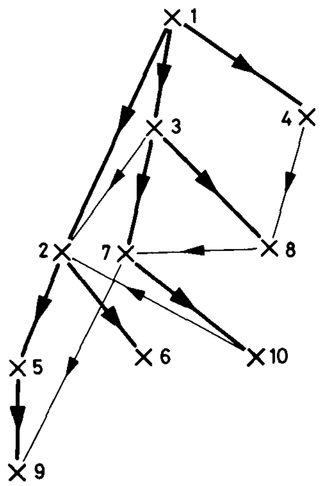

Fig. 14. Directed graph indicating the breadth first search ordering.

Table VI A Comparison of MC21A with Hopcroft and Karp Algorithm[a]

	RANDOM	STRUCTURED		NCUBE
Matrix: n	100	199	822	300
τ	300	701	4534	10400
MC21A	4.9	10.4	124.9	802
Hopcroft and Karp	32.3	52.8	253.6	127

[a] Times in milliseconds on an IBM 370/168.

$$\begin{bmatrix} x & x & & x \\ & x & x & \\ & & x & \\ x & & & \end{bmatrix}$$

Fig. 15 Indication of power of "look ahead."

Here the assignment of the fourth transversal element requires access to every nonzero if no "look ahead" is employed, whereas, by attempting a cheap assignment at every stage, the transversal is completed after accessing only three nonzeros. This is reinforced by our computational results in Tables II and III.

Kuhn's [19] algorithm employs a breadth first search, but the search is only undertaken from one unassigned row at a time and can be very inefficient. The original algorithm of Hall [12] was similar to this.

The algorithm of Ford and Fulkerson [10] again uses a breadth first search, this time starting simultaneously from all unassigned rows; however, they restart the breadth first search after each reassignment. Hopcroft and Karp [15] perform reassignments corresponding to disjoint augmenting paths for each level of the breadth first search before continuing to the next level. They have shown that their algorithm, which uses depth first searches for finding the disjoint augmenting paths, has an $O(n^{1/2}\tau)$ bound, the lowest worst case complexity of any proposed algorithm to date. Now, although we are reassured by the fact that the

$O(n\tau)$ behavior of MC21A is unlikely to be realized in practice, it is interesting to compare an implementation of the $O(n^{1/2}\tau)$ algorithm with MC21A. The implementation is ours because no published code for this algorithm is available. We have written it with care and believe it gives a fair representation of their algorithm, but our results should be interpreted in this light.

One glance at the results of Table VI is sufficient to convince us that our implementation of the Hopcroft and Karp algorithm is not competitive with MC21A on realistic problems, although it is better than MC21A on the worst case example, NCUBE.

7. CHEAP HEURISTICS

Although we could have first attempted a cheap assignment on all of the rows, thus guaranteeing that at least half of our transversal would be found immediately, we have decided against this because our approach makes the algorithm simpler and does not affect its complexity bound. However, if one could design a heuristic such that the cheap assignment, guided by this heuristic, obtained nearly all the transversal, then little work would be left to the depth first search. Both the author [3] and Gustavson [11] have experimented with such heuristics without reaching any firm conclusions. We do not use heuristics here because they will certainly complicate the algorithm and normally require additional work space for their implementation. Indeed our experience with heuristics has been that it is impossible to implement them with the same degree of efficiency as exhibited in MC21A and, additionally, no single heuristic can be guaranteed to work well on all examples. At one time we thought the heuristic M4 of [3], namely, first choose singletons in the active matrix (i.e., the matrix left after eliminating rows and columns already assigned) and, if there are none, choose the nonzero whose combined number of nonzeros in its row and column is least (we call this the element count), would nearly always give a complete assignment. We illustrate below a 6×6 matrix on which this heuristic fails (and it can be shown to be the smallest possible such counterexample). In Figure 16, the number representing each nonzero is its element count in the original matrix.

Al Erisman (Boeing Computer Services) drew our attention to the fact that since we can preorder the rows in order of ascending row count in only $O(n)$ time, it might be an advantage to first do such a preordering before starting the assignment process. At first glance, this seems attractive since our main algorithm is $O(n\tau) + O(n^2)$. However, as we have seen in practice, MC21A performs as $O(n) + O(\tau)$, and so the cost of the preordering may be significant. Indeed, the results in Table VII (the times being the average of ten runs with different permutations of the original matrix) indicate that the algorithm performs noticeably worse if it is preceded by a preordering phase. Additionally, this preprocessing requires extra integer work space of length n. However, the most severe drawback to such a preordering is that, while the original matrix may have a zero-free diagonal, a permutation of it with the rows in order of increasing number of nonzeros may have zeros on the diagonal; hence it may not be possible to cheaply and quickly assign to all the rows in order. A striking example of this is seen in the original matrix of order 1176 with 18,552 nonzeros. Here the times for runs with and without the preordering phase are 383 and 37 milliseconds,

$$\begin{bmatrix} 4 & 5 & & & & \\ 5 & & 6 & 6 & & \\ & & 5 & 5 & & \\ & & 5 & 5 & & \\ & 6 & & & 5 & 5 \\ & 6 & & & 5 & 5 \end{bmatrix}$$

Fig. 16 The 6 × 6 example on which M4 fails to give a maximum transversal

Table VII. A Comparison of Times with and Without Preordering[a]

Matrix: n / τ	1176 / 18552	199 / 701	822 / 4790	300 / 7275	900 / 4970	180 / NCUBE
MC21A	280	12	140	6	15	257
MC21A + PREORDER	459	13	194	9	24	359

[a] Times in milliseconds on an IBM 370/168.

respectively. Thus, we do not recommend that one first permutes the matrix so that the rows are in order of increasing number of nonzeros.

8. CONCLUSION

We have presented a very simple, fast computer program for finding an ordering for permuting a matrix so that its diagonal has a minimum number of zeros. We have examined the complexity of this algorithm both in terms of its worst case bound and its behavior on more realistic examples and have compared it with some other methods for solving the same problem.

ACKNOWLEDGMENTS

I would like to thank D. Gay, F.G. Gustavson, M.A. Saunders, A.M. Erisman, and J.K. Reid for their comments on earlier versions of the code, and to thank the Swedish Science Research Council for its support during the latter stages of this work. I would also like to thank T. Wiberg, J.K. Reid, and A.R. Curtis for their comments on earlier drafts of the manuscript.

REFERENCES

(Note References [2, 24] are not cited in the text.)

1 BERGE, C. *The Theory of Graphs* Methuen, London, 1962.
2 BUNCH, J.R., AND ROSE, D J., Eds In *Proc Conf. Sparse Matrix Computations* (Argonne National Laboratory, Sept 9-11, 1975), Academic Press, New York, 1976.
3. DUFF, I.S. Analysis of sparse systems. Ph.D. thesis, Oxford, England, 1972.
4. DUFF, I.S On permutations to block triangular form. *J. Inst Math. Appl. 19* (1977), 339-342
5 DUFF, I S. A survey of sparse matrix research. *Proc IEEE 65* (1977), 500-535.
6 DUFF, I S Algorithm 575 Permutations for a zero-free diagonal. *ACM Trans. Math. Softw 7,* 3 (Sept 1981), 387-390
7. DUFF, I.S , AND REID, J K. An implementation of Tarjan's algorithm for the block triangularization of a matrix *ACM Trans Math. Softw 4,* 2 (June 1978), 137-147.
8. DULMAGE, A L , AND MENDELSOHN, N.S. A structure theory of bipartite graphs of finite exterior dimension *Trans Roy Soc. Can 53,* Sec. 3 (1959), 1-13
9. DULMAGE, A L., AND MENDELSOHN, N.S. Two algorithms for bipartite graphs. *J. SIAM 11* (1963), 183-194.

10. FORD, L.R , JR., AND FULKERSON, D.R. *Flows in Networks.* Princeton University Press, Princeton, N.J., 1962.
11. GUSTAVSON, F.G. Finding the block lower triangular form of a matrix. In *Sparse Matrix Computations,* J.R. Bunch and D J. Rose (Eds.), Academic Press, New York, 1976.
12. HALL, M. An algorithm for distinct representatives. *Am. Math. Monthly 63* (1956), 716-717.
13. HALL, P. On representatives of subsets. *J. London Math. Soc. 10,* 37, pt. 1 (1935), 26-30
14. HARARY, F. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969
15 HOPCROFT, J.E., AND KARP, R.M. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs *SIAM J. Comput. 2* (1973), 225-231.
16. KETTLER, P., AND WEIL, R. An algorithm to provide structure for decomposition. In *Proc Symp on Sparse Matrices and their Applications,* R A. Willoughby (Ed.), IBM Rep. RA (11707), Yorktown Heights, N Y , 1969.
17 KONIG, D. *Theorie der endlichen und unendlichen Graphen* Chelsea, New York, 1950.
18. KUHN, H W. The Hungarian method for solving the assignment problem. *Naval Res. Logist. Quart. 2* (1955), 83-97.
19. KUHN, H W Variants of the Hungarian method for assignment problems *Naval Res Logist Quart 3* (1957), 253-258.
20 STEWARD, D.V. On an approach to techniques for the analysis of the structure of large systems of equations. *SIAM Rev. 4* (1962), 321-342
21. STEWARD, D.V Partitioning and tearing systems of equations. *SIAM J Numer. Anal. 2* (1965), 345-365.
22. TARJAN, R. Depth-first search and linear graph algorithms. *SIAM J Comput 1* (1972), 146-160
23. WIBERG, T. Permutation of an unsymmetric matrix to block triangular form. Ph.D. dissertation, Dep of Information Processing, Univ. Umeå, Umeå, Sweden, March 1977
24. WILLOUGHBY, R.A. Ed., *Proc. Symp. on Sparse Matrices and their Applications,* IBM Rep. RA 1 (11707), Yorktown Heights, N.Y., 1969.

Received October 1978; revised January 1980; accepted April 1980