make the cheap pre-pass worthwhile. This was in part because it is common for the given matrix to have entries on most of its diagonal. The permutations corresponding to the cheap assignments could destroy this property. In the case of identifying singletons as we did in Section 6.3, however, no initial decrease in the original entries on the diagonal can take place (see Exercise 6.8). Duff et al. (2011) find that KSM and MDM usually find a transversal that is close to maximum and therefore recommend their use, whatever subsequent algorithm is used to complete the matching.

6.5 Symmetric permutations to block triangular form

6.5.1 *Background*

We assume that a row permutation $\mathbf{P}_1$ has been computed so that $\mathbf{P}_1\mathbf{A}$ has entries on every position on its diagonal (unless $\mathbf{A}$ is symbolically singular), and we now wish to find a symmetric permutation that will put the matrix into block lower triangular form. That is, we wish to find a permutation matrix $\mathbf{Q}$ such that $\mathbf{Q}^T(\mathbf{P}_1\mathbf{A})\mathbf{Q}$ has the form

$$\mathbf{Q}^T(\mathbf{P}_1\mathbf{A})\mathbf{Q} = \begin{bmatrix} \mathbf{B}_{11} & & & & & & \\ \mathbf{B}_{21} & \mathbf{B}_{22} & & & & & \\ \mathbf{B}_{31} & \mathbf{B}_{32} & \mathbf{B}_{33} & & & & \\ \cdot & \cdot & \cdot & \cdot & & & \\ \cdot & \cdot & \cdot & & \cdot & & \\ \cdot & \cdot & \cdot & & & \cdot & \\ \mathbf{B}_{N1} & \mathbf{B}_{N2} & \mathbf{B}_{N3} & \cdot & \cdot & \cdot & \mathbf{B}_{NN} \end{bmatrix}, \tag{6.5.1}$$

where each $\mathbf{B}_{ii}$ cannot itself be symmetrically permuted to block triangular form. Then, with $\mathbf{P} = \mathbf{Q}^T\mathbf{P}_1$, we will have achieved the desired permutation (6.1.2). We will show (Section 6.6) that symmetric permutations are all that is required once the transversal has been found. While any row and column singletons will usually have been removed, we do not rely on this.

It is convenient to describe algorithms for this process with the help of the digraphs (directed graphs) associated with the matrices (see Section 1.2). With only symmetric permutations in use, the diagonal entries play no role so we have no need for self-loops associated with diagonal entries. Applying a symmetric permutation to the matrix causes no change in the associated digraph except for the relabelling of its nodes. Thus, we need only consider relabelling the nodes of the digraph, which we find easier to explain than permuting the rows and columns of the matrix.

If we cannot find a closed path (cycle), as defined in Chapter 1, through all the nodes of the digraph, then we must be able to divide the digraph into two parts such that there is no path from the first part to the second. Renumbering the first group of nodes $1, 2, ..., k$ and the second group $k+1, ..., n$ will produce a

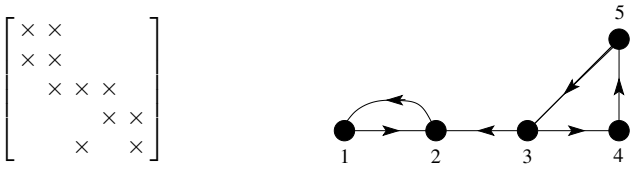

FIG. 6.5.1. A 5×5 matrix and its digraph.

corresponding (permuted) matrix in block lower triangular form. An example is shown in Figure 6.5.1, where there is no connection from nodes $(1, 2)$ to nodes $(3, 4, 5)$. The same process may now be applied to each resulting block until no further subdivision is possible. The sets of nodes corresponding to the resulting diagonal blocks are called **strong components**. For each, there is a closed path passing through all its nodes but no closed path includes these and other nodes. The digraph of Figure 6.5.1 contains just two strong components, which correspond to the two irreducible diagonal blocks.

A triangular matrix may be regarded as the limiting case of the block triangular form in the case when every diagonal block has size 1×1. Conversely, the block triangular form may be regarded as a generalization of the triangular form with strong components in the digraph corresponding to generalized nodes. This observation forms the basis for the algorithms in the next three sections.

6.5.2 *The algorithm of Sargent and Westerberg*

Algorithms for finding the triangular form may be built upon the observation that if $\mathbf{A}$ is a symmetric permutation of a triangular matrix, there must be a node in its digraph from which no path leaves. This node should be ordered first in the relabelled digraph (and the corresponding row and column of the matrix permuted to the first position). Eliminating this node and all edges pointing into it (corresponding to removing the first row and column of the permuted matrix) leaves a remaining subgraph, which again has a node from which no paths leave. Continuing in this way, we eventually permute the matrix to lower triangular form.

To implement this strategy for a matrix that may be permuted to triangular form, we may start anywhere in the digraph and trace a path until we encounter a node from which no paths leave. This is easy since the digraph contains no closed paths (such a digraph is called acyclic); any available choice may be made at each node and the path can have length at most $n - 1$, where n is the matrix order. We number the node at the end of the path first, and remove it and all edges pointing to it from the digraph, then continue from the previous node on the path (or choose any remaining node if the path is now empty) until once again we reach a node with no path leaving it. In this way, the triangular form is identified and no edge is inspected more than once, so the algorithm is economical. We illustrate this with the digraph of Figure 6.5.2. The sequence of paths is illustrated in Figure 6.5.3.

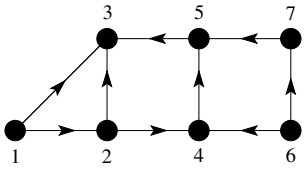

FIG. 6.5.2. A digraph corresponding to a triangular matrix.

Path					**5**						
			3	4	4	**4**					
		2	2	2	2	2	**2**			**7**	
	1	1	1	1	1	1	1	**1**	6	6	**6**
Step	1	2	3	4	5	6	7	8	9	10	11

FIG. 6.5.3. The sequence of paths used for the Figure 6.5.2 case, where nodes selected for ordering are shown in bold.

It may be verified (see Figure 6.5.4) that the relabelled digraph $3 \to 1$, $5 \to 2$, $4 \to 3$, $2 \to 4$, $1 \to 5$, $7 \to 6$, $6 \to 7$ corresponds to a triangular matrix. Observe that at step 5 there was no path from node 5 because node 3 had already been removed. Note also that at step 9 the path tracing was restarted because the path became empty. For large problems known to have triangular structure, this algorithm is very useful in its own right. It allows users to enter data in any convenient form and automatically finds the triangular structure.

We observe that this is a graph-theoretic interpretation of the algorithm that we used in Section 6.3 for the case where the matrix may be reduced to triangular form.

Sargent and Westerberg (1964) generalized this idea to the block case. They define as a **composite node** any group of nodes through which a closed path has been found. Starting from any node, a path is followed through the digraph until:

(i) A closed path is found (identified by encountering the same node or composite node twice), or

(ii) a node or composite node is encountered with no edges leaving it.

In case (i), all the nodes on the closed path must belong to the same strong component and the digraph is modified by collapsing all nodes on the closed path into a single composite node. Edges within a composite node are ignored, and edges entering or leaving any node of the composite node are regarded as entering or leaving the composite node. The path is now continued from the composite node.

In case (ii), as for ordinary nodes in the triangular case, the composite node is numbered next in the relabelling. It and all edges connected to it are removed, and the path now ends at the previous node or composite node, or starts from any remaining node if it would otherwise be empty.

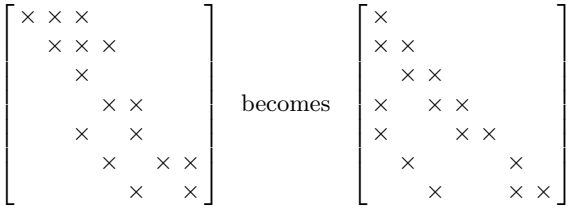

FIG. 6.5.4. The matrices before and after renumbering.

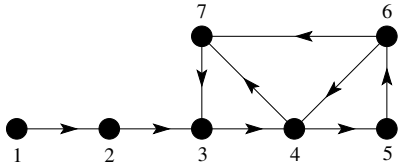

FIG. 6.5.5. A digraph illustrating the algorithm of Sargent and Westerberg.

Thus, the blocks of the required form are obtained successively. This generalization of the triangularization algorithm shares the property that each edge of the original digraph is inspected at most once.

We illustrate with the example shown in Figure 6.5.5. Starting the path at node 1, it continues $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 4$, then $(4, 5, 6)$ is recognized as a closed path, and nodes 4, 5, and 6 are relabelled as composite node $4'$. The path is continued from this composite node to become $1 \to 2 \to 3 \to 4' \to 7 \to 3$. Again, a closed path has been found and $(3, 4', 7)$ is labelled as $3'$ and the path becomes $1 \to 2 \to 3'$. Since there are no edges leaving $3'$, it is numbered first and removed. The path is now $1 \to 2$ and no edges leave node 2, so this is numbered second. Finally, node 1 is numbered as the last block. The corresponding original and reordered matrices are shown in Figure 6.5.6.

The difficulty with this approach is that there may be large overheads associated with the relabelling in the node collapsing step. A simple scheme such as labelling each composite node with the lowest label of its constituent nodes can result in $\mathcal{O}(n^2)$ relabellings. For instance, in Figure 6.5.7 successive composite nodes are $(4, 5)$, $(3, 4, 5, 6)$, $(2, 3, 4, 5, 6, 7)$, $(1, 2, 3, 4, 5, 6, 7, 8)$; in general, such

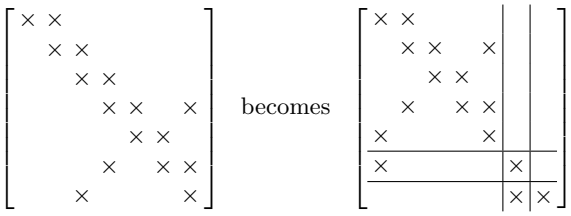

FIG. 6.5.6. The matrices before and after renumbering.

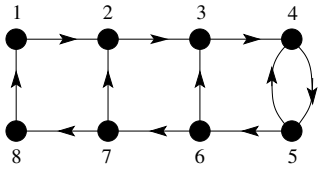

FIG. 6.5.7. A case causing many relabellings.

a digraph with n nodes will involve $2+4+6+...+n = n^2/4 + n/2$ relabellings. Various authors (for example, Munro 1971*a*, 1971*b*, Tarjan 1975) have proposed schemes for simplifying this relabelling, but the alternative approach of Tarjan (1972) eliminates the difficulty. This is described in the next section.

Even though the Sargent and Westerberg algorithm is not as good as Tarjan's, we introduced it for two reasons. First, it is very simple. Secondly, it was developed very early and demonstrated excellent performance on many problems. It provides an intuitive motivation for the more elegant, optimal algorithm of Tarjan.

6.5.3 *Tarjan's algorithm*

The algorithm of Tarjan (1972) follows the same basic idea as that of Sargent and Westerberg, tracing paths and identifying strong components. It eliminates the relabelling through the clever use of a stack very like that in Figure 6.5.3, which was used to find the triangular form. The stack is built using a depth-first search and records both the current path and all the closed paths so far identified. Each strong component eventually appears as a group of nodes at the top of the stack and is then removed from it. We first illustrate this algorithm with two examples and then explain it in general.

Stack							7	7_3	**7**	**7**	**7**	**7**			
					6	6_4	6_4	6_4	6_3	**6**	**6**	**6**			
				5	5	5	5	5	5	5_3	**5**	**5**			
			4	4	4	4	4	4	4	4	4_3	**4**			
		3	3	3	3	3	3	3	3	3	3	3			
	2	2	2	2	2	2	2	2	2	2	2	2	2		
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

FIG. 6.5.8. The stack corresponding to Figure 6.5.5.

Figure 6.5.8 shows the stack at all steps of the algorithm for the digraph of Figure 6.5.5 starting from node 1. In the first six steps, the stack simply records the growing path $1 \to 2 \to 3 \to 4 \to 5 \to 6$. At step 7, we find an edge connecting the node at the top of the stack (node 6) to one lower down (node 4). This is recorded by adding a link, shown as a subscript. Since we know that there is a path up the stack, $4 \to 5 \to 6$, this tells us that (4,5,6) lie on a closed path, but

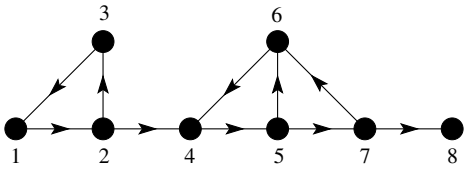

FIG. 6.5.9. An example with two nontrivial strong components.

we do not relabel the nodes. Similarly, at step 9, we record the link $7 \rightarrow 3$ and this indicates that (3, 4, 5, 6, 7) lie on a closed path. There are no more edges from node 7 so it is removed from the path. However it is not removed from the stack because it is part of the closed path $3-7$. We indicate this in Figure 6.5.8 by showing 7 in bold. Now node 6, which is immediately below node 7 on the stack, is at the path end so we return to it, but the only relabelling we do is to make its link point to 3 and we discard the link from 7, thereby recording the closed path 3–7.

We now look for unsearched edges at node 6 and find that there are none, so we label 6 in bold, set the link from 5 to 3, and discard the link from 6 (see column 11 of Figure 6.5.8). At the next step we treat node 5 similarly and in the following loop we label 4 in bold and discard its link, but no change is made to the link at 3 which may be regarded as pointing to itself. Column 13 of Figure 6.5.8 still records that nodes 3, 4, 5, 6, and 7 lie on a closed path. Next, we find that 3 has no unsearched edges. Since it does not have a link to a node below it, there cannot be any path from it or any of the nodes above it to a node below it. We have a 'dead-end', as in the triangular case. The nodes 3–7 constitute a strong component and may be removed from the stack. The trivial strong components (2) and (1) follow. The algorithm also works starting from any other node. Notice that the strong component was built up gradually by relabelling stack nodes in bold once their edges have been searched and each addition demands no relabelling of nodes already in the composite node.

This example was carefully chosen to be simple in the sense that the path always consists of adjacent nodes in the stack. This is not always so, and we illustrate the more general situation in the next example.

Stack											8					
									7	7	7_6	**7**	**7**			
						6	6_4	**6**	**6**	**6**	**6**	**6**	**6**			
					5	5	5	5_4	5_4	5_4	5_4	5_4	**5**			
				4	4	4	4	4	4	4	4	4	4			
		3	3_1	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	
	2	2	2	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2_1	2_1	**2**
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

FIG. 6.5.10. The stack corresponding to Figure 6.5.9.

Consider the digraph shown in Figure 6.5.9. The stack, starting from node 1, is shown in Figure 6.5.10. At step 5, node 3 is removed from the path because it has no unsearched edges, but is not removed from the stack because of its link to node 1. Node 4 is added at step 6 because of the edge (2,4) and the path is $1 \rightarrow 2 \rightarrow 4$. Similarly node 7 is added because an edge connects to it from node 5. At step 12, node 8 is identified as a strong component because it has no edges leading from it and it does not have a link pointing below it in the stack. At step 13, node 7 has no more edges and is recognized as a member of a strong component because of its link to node 6 and is therefore labelled in bold. The strong components (4,5,6,7) and (1,2,3) are found in steps 15 and 17. A useful exercise is to carry through this computation from another starting node (Exercises 6.9 and 6.10).

As with the Sargent and Westerberg algorithm, we start with any node and if ever the stack becomes empty we restart with any node not so far used. It is convenient to give such starting nodes links to themselves. At a typical step we look at the next unsearched edge of the node at the end of the path (we will call this node the current node) and distinguish these cases:

(i) The edge points to a node not on the stack, in which case this new node is added to the top of the stack and given a link that points to itself.

(ii) The edge points to a node lower on the stack than the node linked from the current node, in which case the link is reset to the link of this lower node.

(iii) The edge points to a node higher on the stack than the node linked from the current node, in which case no action is needed.

(iv) There are no unsearched edges from the current node and the link from the current node points below it. In this case, the node is left on the stack but removed from the path. The link for the node before it on the path is reset to the lesser of its old value and that of the link for the current node.

(v) There are no unsearched edges from the current node and the link from the current node does not point below it. In this case, the current node and all those above it on the stack constitute a strong component and so are removed from the stack.

It is interesting (see Exercise 6.11) to see that the excessive relabelling associated with the digraph of Figure 6.5.7 is avoided in the Tarjan algorithm.

A formal proof of the validity of the Tarjan algorithm can readily be developed from the following observations:

(i) At every step of the algorithm, there is a path from any node on the stack to any node above it on the stack.

(ii) For any contiguous group of nodes on the stack, but not on the path (nodes marked in bold), there is a closed path between these nodes and the next node on the stack below this group.

(iii) If nodes α and β are a part of the same strong component, α cannot be removed from the stack by the algorithm unless β is removed at the same time.

Together, (i) and (ii) say that nodes removed from the stack at one step must be a part of the same strong component, while (iii) says that all of the strong component must be removed at the same step.

6.5.4 *Implementation of Tarjan's algorithm*

To implement Tarjan's algorithm, it is convenient to store the matrix as a collection of sparse row vectors, since following paths in the digraph corresponds to accessing entries in the rows of the matrix.

It was convenient for illustration in the last section to label bold those nodes on the stack whose edges have all been searched, but for efficiency we need a rapid means of backtracking from the current node to the previous node on the path. This is better done by storing a pointer with each path node. Hence the following arrays of length n, the matrix order, are needed:

(i) One holding the nodes of the stack.
(ii) One holding, for each path node, a link to the lowest stack node to which a path has so far been found.
(iii) One holding, for each path node, a pointer to the previous path node.
(iv) One holding for each node, its position on the stack if it is there, its position in the final order if it has been ordered, or zero if neither.
(v) One holding, for each path node, a record of how far the search of its edges has progressed.

In addition, the starts of the blocks must be recorded.

Note that each edge is referenced only once and that each of the steps (i) to (v) of Section 6.5.3 that is associated with an edge involves an amount of work that is bounded by a fixed number of operations. There will also be some $\mathcal{O}(n)$ costs associated with initialization and accessing the rows, so the overall cost is $\mathcal{O}(\tau) + \mathcal{O}(n)$ for a matrix of order n with τ entries.

Further details of the algorithm and its implementation are provided by Tarjan (1972) and Duff and Reid (1978*a*). Duff and Reid (1978*b*) also provide an implementation in Fortran; it is interesting to note that their implementation involves less than 75 executable statements.

6.6 Essential uniqueness of the block triangular form

In the last three sections, we have described two alternative algorithms for finding a block triangular form given a matrix with entries on its diagonal. There remains, however, some uncertainty as to whether the final result may depend significantly on the algorithm or the set of entries placed on the diagonal in the first stage. We show in this section that for nonsingular matrices, the result is essentially unique in the sense that any one block form can be obtained from any other by applying row permutations that involve the rows of a single block