SIAM J. MATRIX ANAL. APPL.
Vol. 28, No. 3, pp. 805–821

REDUCING THE TOTAL BANDWIDTH OF A SPARSE UNSYMMETRIC MATRIX*

J. K. REID† AND J. A. SCOTT†

Abstract. For a sparse symmetric matrix, there has been much attention given to algorithms for reducing the bandwidth. As far as we can see, little has been done for the unsymmetric matrix A, which has distinct lower and upper bandwidths l and u. When Gaussian elimination with row interchanges is applied, the lower bandwidth is unaltered, while the upper bandwidth becomes $l+u$. With column interchanges, the upper bandwidth is unaltered, while the lower bandwidth becomes $l+u$. We therefore seek to reduce $\min(l,u)+l+u$, which we call the *total bandwidth*. We compare applying the reverse Cuthill–McKee algorithm to $A+A^T$, to the row graph of A, and to the bipartite graph of A. We also propose an unsymmetric variant of the reverse Cuthill–McKee algorithm. In addition, we have adapted the node-centroid and hill-climbing ideas of Lim, Rodrigues, and Xiao to the unsymmetric case. We have found that using these to refine a Cuthill–McKee-based ordering can give significant further bandwidth reductions. Numerical results for a range of practical problems are presented and comparisons made with the recent lexicographical method of Baumann, Fleischmann, and Mutzbauer.

Key words. matrix bandwidth, sparse unsymmetric matrices, Gaussian elimination, Cuthill–McKee algorithm

AMS subject classification. 65F50

DOI. 10.1137/050629938

1. Introduction.

If Gaussian elimination is applied without interchanges to an unsymmetric matrix $A=\{a_{ij}\}$ of order n, each fill-in takes place between the first entry of a row and the diagonal or between the first entry of a column and the diagonal. It is therefore sufficient to store all the entries in the lower triangle from the first entry in each row to the diagonal and all the entries in the upper triangle from the first entry in each column to the diagonal. This simple structure allows straightforward code using static data structures to be written. We will call the sum of the lengths of the rows the *lower profile* and the sum of the lengths of the columns the *upper profile*.

We will also use the term *lower bandwidth* for $l=\max_{a_{ij}\neq 0}(i-j)$ and the term *upper bandwidth* for $u=\max_{a_{ij}\neq 0}(j-i)$. For a symmetric matrix, these are the same and are called the *semibandwidth*. A particularly simple data structure is available by taking account of only the bandwidths l and u. If row interchanges are used for stability reasons during the factorization, it may be readily verified that the lower bandwidth remains l but the upper bandwidth may increase to $l+u$. With column interchanges (or row interchanges applied while factorizing A^T), the upper bandwidth is unaltered, while the lower bandwidth becomes $l+u$. We may therefore always have one triangular factor of bandwidth $\min(l,u)$ and the other of bandwidth $l+u$. Thus we seek to reduce $\min(l,u)+l+u$, which we call the *total bandwidth*.

Many algorithms for reducing the bandwidth of a sparse symmetric matrix A have been proposed in the literature, most of which make extensive use of the adjacency graph $\mathcal{G}$ of the matrix. This is an undirected graph that has a node for each row (or

*Received by the editors April 25, 2005; accepted for publication (in revised form) by V. Simoncini March 22, 2006; published electronically October 4, 2006.
http://www.siam.org/journals/simax/28-3/62993.html
†Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, England (j.k.reid@rl.ac.uk, j.a.scott@rl.ac.uk). The work of the second author was supported by the EPSRC grant GR/S42170.

column) of the matrix, and node i is a neighbor of node j if a_{ij} (and by symmetry a_{ji}) is an entry (nonzero) of A. An important and well-known example of an algorithm that uses $\mathcal{G}$ is that of Cuthill and McKee [2]. The main aim of this paper is to consider how variants of the Cuthill–McKee algorithm can be used to order an unsymmetric matrix for small total bandwidth.

In some circumstances, reordering the matrix and then using a band solver will be the method of choice for solving large sparse linear systems. However, in many situations it is more appropriate to use other sparse direct methods. In this study, we concentrate solely on the reduction of the bandwidth of unsymmetric matrices and do not address the question of when a band solver is the best choice.

The rest of this paper is organized as follows. We begin (in section 2) by commenting on the importance of reordering a matrix to block form prior to applying a bandwidth reduction algorithm. In section 3, we briefly describe the Cuthill–McKee algorithm and the variant that reverses the order (RCM). Then, in section 4, we discuss three undirected graphs that can be associated with an unsymmetric matrix A and that can be reordered using RCM. We then propose in section 5 an unsymmetric variant of RCM. In section 6, we look at modifying the hill-climbing algorithm of Lim, Rodrigues, and Xiao [10] to improve a given ordering, and in section 7, we propose a variant of the node-centroid algorithm of [10] for the unsymmetric case. In section 8, we discuss the recently published algorithm of Baumann, Fleischmann, and Mutzbauer [1] for reducing the bandwidth of an unsymmetric matrix. In section 9, we use our proposed algorithms to reorder a set of matrices that arise from a range of practical problems; we report the total bandwidths before and after reordering, and we summarize our findings in section 10.

2. The block triangular form. In the symmetric case, it may be possible to preorder the matrix A to block diagonal form

$$\begin{bmatrix} A_{11} & & & & & \\ & A_{22} & & & & \\ & & A_{33} & & & \\ & & & A_{44} & & \\ & & & & \dots & \end{bmatrix}. \tag{2.1}$$

In this case, each block may be permuted to band form, and the overall matrix is a band matrix; the profile is the sum of the profiles of the blocks, and the bandwidth is the greatest bandwidth of a block.

The unsymmetric case is not so straightforward because we need also to exploit the block triangular form

$$\begin{bmatrix} A_{11} & & & & & \\ A_{21} & A_{22} & & & & \\ A_{31} & A_{32} & A_{33} & & & \\ A_{41} & A_{42} & A_{43} & A_{44} & & \\ \dots & \dots & \dots & \dots & \dots & \end{bmatrix}, \tag{2.2}$$

where the blocks A_{ll}, $l = 1, 2, \dots, N$, are all square. A matrix that can be permuted to this form with $N > 1$ diagonal blocks is said to be *reducible*; if no block triangular form other than the trivial one with a single block ($N = 1$) can be found, the matrix is *irreducible*. The advantage of the block triangular form (2.2) is that the corresponding

set of equations $Ax = b$ may be solved by the block forward substitution

$$A_{ii}x_i = b_i - \sum_{j=1}^{i-1} A_{ij}x_j, \quad i = 1, 2, \ldots, N. \tag{2.3}$$

There is no fill in the off-diagonal blocks, which are involved only in matrix-by-vector multiplications. It therefore suffices to permute each diagonal block A_{ii} to band form. We will take the upper and lower profiles to be the sums of the upper and lower profiles of the diagonal blocks, and the upper and lower bandwidths to be the greatest of the upper and lower bandwidths of the diagonal blocks.

3. The Cuthill–McKee algorithm. The Cuthill–McKee algorithm is a well-known and successful algorithm for reducing the bandwidth of a symmetric matrix of order n. It does this for a given starting node s by relabeling the nodes of the adjacency graph $\mathcal{G}$ in order of increasing distance from s. The algorithm is outlined in Figure 1. Here the degree of a node i is defined as the number of its neighbors. If $\mathcal{G}$ has more than one component, the procedure is repeated from a starting node in each component.

```
ALGORITHM CUTHILL–MCKEE.
Label s as node 1; l_1 = {s}; i = 1
do k = 2, 3, ... until i = n
    l_k = {}
    do for each v ∈ l_{k-1} in label order
        do for each neighbor u of v that has not been labeled,
                in order of increasing degree
            add u to l_k; i = i + 1; label u as node i
        end do
    end do
end do
```

FIG. 1. *Cuthill–McKee ordering algorithm.*

Ordering the nodes in this way groups them into "level sets," that is, nodes at the same distance from the starting node. Since nodes in level set l_k can have neighbors only in level sets l_{k-1}, l_k, and l_{k+1}, the reordered matrix is block tridiagonal with blocks corresponding to the level sets. It is therefore desirable that the level sets be small, which is likely if there are many of them. The size of the largest level set is called the *width* of the level structure. The width and number of level sets (*height* of the level structure) are dependent on the choice of the starting node s. Algorithms for finding a good starting node are usually based on finding a pseudodiameter (pair of nodes that are a maximum distance apart or nearly so). Much effort has gone into efficiently finding a pseudodiameter; see, for example, [7] and [12] and the references therein. The modified Gibbs Poole Stockmeyer (MGPS) algorithm of Reid and Scott [12] is outlined in Figure 2. For efficiency, the test on $w(r)$ may be performed during the formation of the level set structure so that the structure can be discarded as soon as a large value of $w(r)$ is found. In the inner loop, the choice of 5 nodes was made on the basis of numerical experimentation.

George [6] found that the profile may be reduced if the Cuthill–McKee ordering is reversed (the bandwidth is unchanged). The reverse Cuthill–McKee (RCM) algorithm and variants of it remain in common use. For example, an implementation is available

ALGORITHM MGPS.
Construct $\mathcal{G}$ and choose a starting node s of smallest degree
Form the level structure rooted at s, of height $h(s)$ and width $w(s)$
outer: do
  $w_{least} = \infty$
  inner: do for up to 5 nodes r of the final level set that are not neighbors, in order of increasing degree
    Form the level set structure rooted at r,
    of height $h(r)$ and width $w(r)$
    if $w(r) \geq w_{least}$ **cycle inner**
    if $h(r) > h(s)$ **then**
      $s = r$; **cycle outer**
    end if
    $e = r$; $w_{least} = w(r)$;
  end do inner
  exit outer
end do outer
Pseudodiameter is defined by the pair (s, e).

FIG. 2. *Modified Gibbs Poole Stockmeyer algorithm.*

within MATLAB as the function `symrcm`, and RCM is included as an option within the package `MC60` from the mathematical software library HSL [8]. We note that both these implementations apply RCM directly to the supplied matrix A, without attempting to first reorder the matrix to block diagonal form (2.1).

4. Undirected graphs for unsymmetric matrices. In this section, we consider three adjacency graphs that can be associated with an unsymmetric matrix A. In each case, we employ the RCM algorithm to reduce the semibandwidth of the graph, and this permutation is then used to reorder A.

4.1. Using $A + A^T$. For a matrix whose structure is nearly symmetric, an effective strategy is to find a symmetric permutation that reduces the bandwidth of the structure of the symmetric matrix $A+A^T$. The MATLAB function `symrcm` applies RCM to the adjacency graph of $A + A^T$. If the symmetric permutation is applied to A, the lower and upper bandwidths are no greater than the semibandwidth of the permuted $A + A^T$. Of course, the same algorithm may be applied to a matrix that is far from symmetric, and the same results apply, but the effectiveness is uncertain. It is likely to be helpful to permute A to make it more symmetric. We will judge this by its *symmetry index*, that is, the number of off-diagonal entries a_{ij} for which a_{ji} is also an entry, divided by the total number of off-diagonal entries. Permuting a large number of off-diagonal entries onto the diagonal reduces the number of unmatched off-diagonal entries, which in turn generally increases the symmetry index (see, for example, [5], [9]).

Note that most algorithms for preordering the matrix to block triangular form (2.2) begin with a permutation that places entries on the diagonal. Thus, in this case, permuting entries onto the diagonal is not available as a strategy for improving the symmetry index.

4.2. Bipartite graph. The bipartite graph of A, which we will denote by $\mathcal{G}_{bipart}$, has a node for each row and a node for each column, and row node i and column node

j are neighbors if a_{ij} is an entry. It is straightforward to see that $\mathcal{G}_{bipart}$ is actually the adjacency graph of the $2n \times 2n$ symmetric matrix

$$\hat{A} = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}. \tag{4.1}$$

Starting the Cuthill–McKee algorithm with any node of $\mathcal{G}_{bipart}$, the level sets are alternately sets of rows and sets of columns. If we start from a row node and perform the corresponding symmetric permutation on the matrix (4.1), we find the matrix

$$\begin{bmatrix} 0 & A_{11} & & & & & & \\ A_{11}^T & 0 & A_{21}^T & & & & & \\ & A_{21} & 0 & A_{22} & & & & \\ & & A_{22}^T & 0 & A_{32}^T & & & \\ & & & A_{32} & 0 & A_{33} & & \\ & & & & A_{33}^T & 0 & A_{34}^T & \\ & & & & & \cdots & \cdots & \cdots \end{bmatrix}, \tag{4.2}$$

where A_{lm} is the submatrix of A corresponding to the rows of row level set l and columns of column level set m.

If we permute the rows of A by the row level sets and the orderings within them, and permute the columns by the column level sets and the orderings within them, we find the block bidiagonal form

$$\begin{bmatrix} A_{11} & & & & \\ A_{21} & A_{22} & & & \\ & A_{32} & A_{33} & & \\ & & A_{43} & A_{44} & \\ & & & \cdots & \cdots \end{bmatrix}, \tag{4.3}$$

which is also the submatrix of (4.2) consisting of block rows 1, 3, ... and block columns 2, 4, ... We illustrate with a small reordered example in Figure 3. Here there are four row level sets, of sizes 1, 3, 2, 2, and three column level sets, of sizes 3, 2, 3.

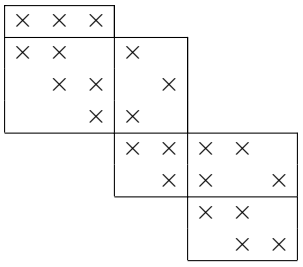

FIG. 3. *A matrix with rows and columns reordered using the permutations obtained by applying Cuthill–McKee to its bipartite graph.*

This example has entries on the whole of the diagonal, which will not necessarily be the case. However, if the matrix is structurally nonsingular, the diagonal will always intersect each of the blocks. This is because the leading k columns must be of full structural rank and similarly for the leading k rows.

The semibandwidth of the reordered matrix (4.2) is at most one less than the largest sum of the sizes of two adjacent level sets, that is, one less than the largest

sum of the sizes of a row level set and an adjacent column level set. The corresponding results for the reordered unsymmetric matrix (4.3) are that the lower bandwidth is at most one less than the sum of the sizes of two adjacent column level sets and the upper bandwidth is at most one less than the sum of the sizes of two adjacent row level sets. Note, however, that all these bounds are pessimistic; they do not take into account the ordering of the nodes within each level set (and RCM does well in this respect) and, in the case (4.3), of the position of the matrix diagonal within the blocks.

4.3. Row graph. Another alternative is to consider the *row graph* [11] of A, which is defined to be the adjacency graph of the symmetric matrix AA^T, where matrix multiplication is performed without taking cancellations into account (so that, if a coefficient of AA^T is zero as a result of numerical cancellation, it is still considered to be an entry). The nodes of the row graph correspond to the rows of A, and nodes i and j ($i \neq j$) are neighbors if and only if there is at least one column k of A for which a_{ik} and a_{jk} are both entries. The row graph has been used by Scott [13], [14] to order the rows of unsymmetric matrices prior to solving the linear system using a frontal solver. We can obtain an ordering for the rows of A by applying the RCM algorithm to AA^T. This will ensure that rows with entries in common are nearby; that is, the first and last entry of each column will not be too far apart. If the columns are now ordered according to their last entry, the lower bandwidth will be small and the upper bandwidth will not be large.

A potential disadvantage of computing and working with the pattern of AA^T is that it can be costly in terms of time and memory requirements. This is because AA^T may contain many more entries than A. It fails completely if A has a full column (AA^T is full), but such a matrix cannot be permuted to have small lower and upper bandwidths.

5. Unsymmetric RCM. Any reordering within a Cuthill–McKee level set of section 4.2 will alter the positions of the leading entries of the columns of a submatrix A_{ii} or the rows of a submatrix A_{ji}, $j = i+1$. It will make exactly the same change to the profile of the matrix (4.2) as it does to the sum of the upper and lower profiles of the matrix (4.3). If it reduces the bandwidth of the matrix (4.2), it will reduce either the upper or lower bandwidth of the matrix (4.3); however, the converse is not true: It might reduce the upper or lower bandwidth of the matrix (4.3) without reducing the bandwidth of the matrix (4.2). It follows that it may be advantageous for bandwidth reduction to develop a special-purpose code for the unsymmetric case, rather than giving the matrix (4.1) to a general-purpose code for reducing the bandwidth of a symmetric matrix. We have developed a prototype unsymmetric bandwidth reduction code of this kind. Our algorithm is based on the MGPS algorithm outlined in Figure 2. As in the bipartite approach discussed in section 4.2, we use the adjacency graph $\mathcal{G}_{bipart}$ so that the level sets alternate between a set of rows and a set of columns, but our unsymmetric algorithm bases its decisions on the total bandwidth of the unsymmetric matrix A rather than on the bandwidth of the matrix (4.2). Our algorithm is given in Figure 4.

The profile and bandwidth are likely to be reduced if the rows of each level set are ordered according to their leading entries. This happens automatically with the Cuthill–McKee algorithm and was done for the example in Figure 3.

Reversing the Cuthill–McKee ordering may reduce the profile. It is reduced if the column index of the trailing entry in a row is lower than the column index of the trailing entry in an earlier row. There is an example in block A_{22} of Figure 3.

ALGORITHM UNSYMMETRIC RCM.
Construct $\mathcal{G}_{bipart}$ and choose a starting node s of smallest degree
Apply Cuthill–McKee from s, finding height $h(s)$ and total bandwidth $t(s)$
$t_{least} = \infty$; $h_{most} = 1$
outer: do
  inner: do for up to 5 nodes r of the final level set that are not neighbors,
      in order of increasing degree
    Apply Cuthill–McKee from r, finding height $h(r)$
    and total bandwidth $t(r)$
    if $t(r) > t_{least}$ **cycle inner**
    if $t(r) < t_{least}$ or $h(r) > h(s)$ **then**
      $e = s$; $s = r$; $t_{least} = t(r)$; $h_{most} = h(s)$; **cycle outer**
    end if
  end do inner
  exit outer
end do outer
Order using the Cuthill–McKee ordering from e
Reverse the order

FIG. 4. *Unsymmetric RCM ordering algorithm.*

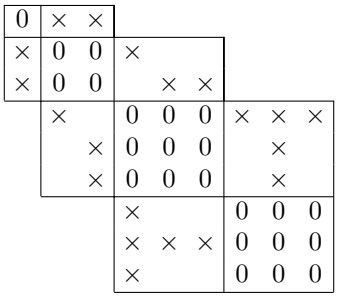

FIG. 5. *A symmetric matrix ordered by Cuthill–McKee.*

6. Hill climbing to improve a given ordering. In this and the next section, we consider algorithms that are not based on level sets in a graph and are therefore completely different.

Lim, Rodrigues, and Xiao [10] propose a hill-climbing algorithm for reducing the semibandwidth of a symmetric matrix. An entry a_{ij} in a matrix A with semibandwidth b is called *critical* if $|i-j| = b$. For each critical entry a_{ij} in the lower-triangular part, an interchange of i with $k < i$ or j with $k > j$ is sought that will reduce the number of critical entries. For example, a_{94} is critical in Figure 5, and the semibandwidth is reduced from 5 to 4 by interchanging column 4 with column 5 and row 4 with row 5. As a column is moved backwards, its first entry is moved away from the diagonal, while its last entry is moved nearer. If the distance of the first entry from the diagonal is d, we can therefore limit the choice of k to the range $j < k < j + b - d$ since we want $d + k - j$ to be smaller than the bandwidth b. Similarly, if the distance of the last entry in row i from the diagonal is l, we limit the choice of k to the range $i - b + l < k < i$. Each interchange while the semibandwidth is b reduces the number of critical entries by one. If the number of critical entries becomes zero, we recommence the algorithm for semibandwidth $b - 1$ and continue until none of the critical entries

for the current semibandwidth can be interchanged to reduce their number. The algorithm is summarized as Figure 6. Note that this hill-climbing algorithm cannot increase the semibandwidth.

```
ALGORITHM HC (SYMMETRIC).
outer: do
    Form the set V_c of critical nodes
    do until V_c is empty
        if there are nodes u ∈ V_c and v ∉ V_c such that
          swapping u and v leaves both noncritical then
          swap u and v and remove u from V_c
        else
          exit outer
        end if
    end do
end do outer
```

FIG. 6. *Hill climbing algorithm for symmetric matrices.*

We have adapted this idea to reduce the lower and upper bandwidths of an unsymmetric matrix. If the lower bandwidth is l and the upper bandwidth is u, we call an entry a_{ij} in the lower triangle for which $i - j = l$ a *critical lower* entry, and an entry a_{ij} in the upper triangle for which $j - i = u$ a *critical upper* entry. We have found it convenient to alternate between making row interchanges while the column permutation is fixed and making column interchanges while the row permutation is fixed. While making row interchanges to reduce the number of critical upper entries, we seek to exchange a row i containing a critical upper entry with another row so that the number of critical upper entries is reduced by one while the lower bandwidth is not increased. If the distance between the leading entry in the row and the diagonal is d, we limit our search to rows in the range $i - l + d \leq k < i$. For example, we do not exchange rows 3 and 4 in Figure 3, since this would increase the lower bandwidth.

Similarly, while making row interchanges to reduce the number of critical lower entries, we seek to exchange a row i containing a critical lower entry with another row so that the number of critical lower entries is reduced by one while the upper bandwidth is not increased. The row hill-climbing algorithm is outlined in Figure 7. Column hill climbing is analogous, using column interchanges to first reduce the upper bandwidth as much as possible and then to reduce the lower bandwidth as much as possible.

One complete iteration of our hill-climbing algorithm for unsymmetric matrices consists of row hill climbing followed by column hill climbing. We continue until a complete iteration fails to reduce one of the bandwidths or the total number of critical entries. This is illustrated in Figure 8.

7. Node centroid ordering. The hill-climbing algorithm of the previous section is essentially a local search and is very dependent on the initial order that it is given. To generate other initial orderings, Lim, Rodrigues, and Xiao [10] propose an algorithm that they call "node-centroid." For the graph of a symmetric matrix, they define $N_\lambda(i)$ to be the set of neighbors j of node i for which the distance $|i - j|$ is at least λb, where b is the semibandwidth and $\lambda \leq 1$ is a parameter for which they recommend a value of 0.95. They refer to such neighbors as λ-*critical*. $w(i)$ is then defined as the average node index over $i \cup N_\lambda(i)$, and the nodes are ordered by increas-

```
Algorithm HC (row).
rows: do
    Form the set V_u of rows that contain a critical upper entry
    do until V_u is empty
        if there are rows u ∈ V_u and v ∉ V_u such that swapping leaves both
                noncritical and does not increase the lower bandwidth then
            swap u and v and remove u from V_u
        else
            exit rows
        end if
    end do
end do rows
cols: do
    Form the set V_l of columns that contain a critical lower entry
    do until V_l is empty
        if there are columns u ∈ V_u and v ∉ V_u such that swapping leaves
            both noncritical and does not increase the upper bandwidth then
            swap u and v and remove u from V_l
        else
            exit cols
        end if
    end do
end do cols
```

Fig. 7. *Row hill climbing algorithm for unsymmetric matrices.*

```
Algorithm HC (unsymmetric).
do while lower bandwidth, upper bandwidth, or
        number of critical entries is reduced
    call HC(row)
    call HC(column)
end do
```

Fig. 8. *Hill climbing for unsymmetric matrices.*

ing $w(i)$. This will tend to move a row with a λ-critical entry in the lower triangle but no λ-critical entry in the upper triangle forward; hopefully, its new leading entry will be nearer the diagonal than the old one was, and its trailing entry will not have moved out so much that it becomes critical. Similar arguments apply to a row with a λ-critical entry in the upper triangle but no λ-critical entry in the lower triangle, which will tend to be moved back. The algorithm is outlined in Figure 9.

Lim, Rodrigues, and Xiao [10] apply a sequence of major steps, each of which consists of two iterations of node centroid ordering followed by one iteration of hill climbing, as illustrated in Figure 10. The decision to perform hill climbing after two steps of the node-centroid algorithm was taken on the basis of numerical experimentation. Using a Cuthill–McKee-type initial ordering with a random starting node, Lim, Rodrigues, and Xiao [10] report encouraging results for the DWT set of symmetric problems from the Harwell–Boeing Sparse Matrix Collection [4].

We have adapted this idea to the unsymmetric case by again alternating between permuting the rows while the column permutation is fixed and permuting the columns

ALGORITHM NC (SYMMETRIC).
choose $\lambda \leq 1$.
do $i = 1, n$
  $w(i) = i$; $c(i) = 1$; form $N_\lambda(i)$
  do for each $j \in N_\lambda(i)$
    $w(i) = w(i) + j$; $c(i) = c(i) + 1$
  end do
  $w(i) = w(i)/c(i)$
end do
sort entries of w into increasing order
reorder nodes in accord with the sorted sequence.

FIG. 9. *Node centroid algorithm for symmetric matrices.*

ALGORITHM NCHC (SYMMETRIC).
choose an initial ordering
do while semibandwidth is reduced
  call NC(symmetric)
  call NC(symmetric)
  call HC(symmetric)
end do

FIG. 10. *Node centroid plus hill climbing for symmetric matrices.*

while the row permutation is fixed. Suppose that the lower bandwidth is l and the upper bandwidth is u. While permuting the rows, only the leading and trailing entries of the rows are relevant, since they will still have these properties after the row permutation. If the leading or trailing entry of row i is λ-critical, it is desirable to move the row. If its leading entry is in column l_i and its trailing entry is in column u_i, the gap between the upper band and the trailing entry is $u + i - u_i$, and the gap between the lower band and the leading entry is $l_i - (i - l) = l_i - i + l$. If we move the row forward to become row $i + \delta$, the gaps become $u + i + \delta - u_i$ and $l_i - i - \delta + l$. If $l > u$, it would seem desirable to make the gap at the trailing end greater than the gap at the leading end. We choose a parameter $\alpha > 1$ and aim for the gap at the trailing end to be α times greater than the gap at the leading end; that is,

$$u + i + \delta - u_i = \alpha(l_i - i - \delta + l) \tag{7.1}$$

or

$$\delta = \frac{(u_i - i - u) + \alpha(l_i - i + l)}{1 + \alpha}. \tag{7.2}$$

Similar calculations for $l = u$ and $l < u$ lead us to conclude that a desirable position for the row is given by the equation

$$w(i) = \begin{cases} i + \frac{(u_i - i - u) + \alpha(l_i - i + l)}{1 + \alpha} & \text{if } l > u, \\ i + \frac{(u_i - i - u) + (l_i - i + l)}{2} & \text{if } l = u, \\ i + \frac{\alpha(u_i - i - u) + (l_i - i + l)}{1 + \alpha} & \text{if } l < u. \end{cases} \tag{7.3}$$

For other rows, we set $w(i) = i$. We sort the rows in increasing order of $w(i)$, $i = 1, 2, \ldots, n$. This is summarized in Figure 11. In our numerical experiments (see section 9), we found that a suitable value for α is 2.

ALGORITHM NC (ROW).
choose $\lambda \leq 1$ and $\alpha > 1$.
compute l, u.
if $(l > u)$ **then**
  $\beta = 1/(1+\alpha)$; $\gamma = \alpha/(1+\alpha)$
else if $(l < u)$ **then**
  $\beta = \alpha/(1+\alpha)$; $\gamma = 1/(1+\alpha)$
else
  $\beta = 1/2$; $\gamma = 1/2$
end if
do $i = 1, n$
  $w(i) = i$
  compute l_i, u_i
  if $u_i - i > \lambda u$ **or** $i - l_i > \lambda l$ **then**
    $w(i) = i + \beta * (u_i - i - u) + \gamma * (l_i - i + l)$
  end if
end do
sort entries of w into increasing order
reorder rows in accord with the sorted sequence.

FIG. 11. *Node centroid algorithm for ordering the rows of an unsymmetric matrix.*

Similar considerations apply to ordering the columns of the matrix with the row order fixed. We apply a sequence of major steps, each consisting of two iterations of the node-centroid row ordering followed by row hill climbing, then two iterations of the node-centroid column ordering followed by column hill climbing. We continue until the total bandwidth ceases to decrease. This is illustrated in Figure 12. We found in our numerical experiments that it is sufficient to limit the number of cycles of the do loop to 10.

ALGORITHM NCHC (UNSYMMETRIC).
choose an initial ordering
do
  call NC(row)
  call NC(row)
  call HC(row)
  if total bandwidth not reduced **exit**
  call NC(column)
  call NC(column)
  call HC(column)
  if total bandwidth not reduced **exit**
end do

FIG. 12. *Node centroid plus hill climbing for unsymmetric matrices.*

8. Relaxed double ordering. Before presenting numerical results for our proposed algorithm, in this section we briefly discuss the recently published algorithm of Baumann, Fleischmann, and Mutzbauer [1] for reducing the bandwidth of an unsymmetric matrix. The sparsity pattern of the matrix is represented by a (0,1)-matrix, that is, a matrix which is the same as the original matrix except that each nonzero

entry is replaced by 1. Each row and column of the (0,1)-matrix then defines a binary number. The algorithm proceeds by alternating between ordering the rows in decreasing order and ordering the columns in decreasing order. The authors [1] show that this converges to a limit and call it a "double ordering." Following reverse Cuthill–McKee, they reverse the converged ordering. Since only the leading entries of the rows or columns affect the bandwidths and profiles, we have implemented an efficient variant in which no attempt is made to order the rows or columns with the same leading entry. We call this a *relaxed double ordering* (RDO). Results for the RDO algorithm are included in section 9.

Unfortunately, there are huge numbers of double orderings, and Baumann, Fleischmann, and Mutzbauer [1] have no strategy for choosing a good one. For example, a Cuthill–McKee ordering produces a relaxed double ordering regardless of the starting node, since the leading entries of the rows (or columns) form a monotonic sequence. There is scope for the double ordering to reduce the profile of an RCM ordering, but our experience is that the improvement is slight and is often at the expense of the bandwidths (see section 9.2).

9. Numerical experiments. In this section, we first describe the problems that we use for testing the algorithms discussed in this paper and then present numerical results.

TABLE 9.1
The test problems; see text for details.

Identifier	Order	Number of entries	Symmetry index
4cols†	11770	43668	0.0159
10cols†	29496	109588	0.0167
bayer01	57735	277774	0.0002
bayer03	6747	56196	0.0031
circuit_3	12127	48137	0.7701
ethylene-1†	10673	80904	0.2973
extr1	2837	11407	0.0042
g7jac200sc	59310	837936	0.0323
fidapm11	22294	623554	1.0000
hydr1	5308	23752	0.0041
impcol_d	425	1339	0.0567
jan99jac020sc	6774	38692	0.0037
lhr71c	70304	1528092	0.0015
mark3jac140	64089	399735	0.0740
poli_large	15575	33074	0.0035
radfr1	1048	13299	0.0537
rdist1	4134	94408	0.0588
sinc15	11532	568526	0.0138
Zhao2	33861	166453	0.9225

9.1. Test problems. The test problems are listed in Table 9.1. Each arises from a real engineering or industrial application. Problems marked with a † are chemical process engineering problems that were supplied to us by Mark Stadtherr of the University of Notre Dame. The remaining problems are available through the University of Florida Sparse Matrix Collection [3]. Most of the test problems were chosen on the grounds of being highly unsymmetric, because working with the symmetrized matrix $A + A^T$ will be satisfactory for near-symmetric matrices. We include two (nearly) symmetric matrices to illustrate this. We have chosen problems of different sizes since, in our experience, for some users it is not just the very large

TABLE 9.2

Details of the block triangular form for our test problems. n_1 and n_2 are the numbers of 1×1 and 2×2 blocks; $n_{>2}$ is the number of larger blocks; n_{off} is the total number of entries in the off-diagonal blocks. For the largest diagonal block A_{kk}, m is the order, me is the number of entries, avg is the average number of entries per row, si is the symmetry index, and me_{kk} is the number of entries in $A_{kk}A_{kk}^T$.

Identifier	n_1	n_2	$n_{>2}$	n_{off}	Largest block A_{kk}				
					m	me	avg	si	me_{kk}
4cols	0	0	1	0	11770	43668	3.71	0.0159	210026
10cols	0	0	1	0	29496	109588	3.72	0.0167	527124
bayer01	8858	0	3	28228	48803	240222	4.92	0.0812	1236678
bayer03	1772	2	6	19575	4776	33555	7.02	0.1066	252236
circuit_3	4520	0	1	9593	7607	34024	4.47	0.5579	76178
ethylene-1	2137	0	7	12865	8336	65375	7.84	0.3000	203920
extr1	424	0	1	464	2413	10519	4.35	0.0935	34118
fidapm11	0	0	1	0	22294	623554	28.0	1.0000	4067228
g7jac200sc	0	0	1	0	59310	837936	14.1	0.0323	6377172
hydr1	968	0	6	1420	2370	11738	4.95	0.0730	47946
impcol_d	226	0	1	551	199	562	2.82	0.0275	1350
jan99jac020sc	0	0	1	0	6774	38692	5.71	0.00376	603720
lhr71c	7038	0	28	95912	7663	173683	22.7	0.07396	2877995
mark3jac140	0	0	1	0	64089	399735	6.24	0.4225	773752
poli_large	15450	12	4	17266	90	286	3.18	0.1633	680
radfr1	97	0	1	969	951	12233	12.9	0.4751	34032
rdist1	198	0	1	3959	3936	90251	22.9	0.4821	280538
sinc15	652	0	1	24343	10880	543531	50.0	0.2615	11063488
Zhao2	0	0	1	0	33861	166453	4.92	0.9225	549692

problems that are important: In their applications they must repeatedly factorize and solve many small or medium-sized problems efficiently, and so spending time and effort on getting a good ordering is essential.

In Table 9.2, we give details of the block triangular form for each of our test matrices. We note that 4cols, 10cols, fidapm11, g7jac200sc, jan99jac020sc, mark3jac140, and Zhao2 are irreducible, while a number of problems (including rdist1 and circuit_3) have only one block of order greater than 1. Most of the remaining problems have fewer than 10 blocks of order greater than 1. As expected, the matrix $A_{kk}A_{kk}^T$ contains many more entries than A_{kk}. We also note that, for the reducible examples, the symmetry index of A_{kk} is usually larger than that of the original matrix.

9.2. Test results. We first present results for applying the HSL [8] implementation of the RCM algorithm (MC60) to the following matrices: (i) $A + A^T$; (ii) $B + B^T$, where $B = PA$ is the permuted matrix after employing the HSL routine MC21 to put entries on the diagonal; (iii) AA^T; (iv) the matrix $\hat{A}$ given by (4.1); and (iv) Unsymmetric RCM code (this is column 8, which is headed A). The total bandwidth for each ordering and for the initial ordering is given in Table 9.3. Results are also given for the RDO algorithm (section 8). A blank entry in the $B + B^T$ column indicates that the matrix A has no zeros on the diagonal, and in these cases, MC21 is not applied. We see that for some problems applying MC21 prior to the reordering with RCM can significantly reduce the bandwidths, but narrower bandwidths are achieved by working with either the row graph (AA^T) or the bipartite graph ($\hat{A}$) or using the unsymmetric RCM. For many of our test examples, the RDO orderings are poorer. However, they are often a significant improvement on the initial ordering, and for a small number of problems (notably radfr1a and rdist1) RDO produces good orderings.

TABLE 9.3
The total bandwidth for the RDO and RCM ordering algorithms.

Identifier	Initial	RDO	RCM				
			$A+A^T$	$B+B^T$	AA^T	$\hat{A}$	A
4cols	11770	4768	846		460	565	541
10cols	29496	13855	1052		546	572	557
bayer01	57735	45332	52201	4117	2232	2236	2331
bayer03	6747	3660	6747	1074	651	651	612
circuit_3	12127	10979	12127	12127	10441	11157	11324
ethylene-1	10664	8301	7797		5114	5093	5306
extr1	2837	1610	2575	298	171	169	169
fidapm11	19515	3315	3189		3299	3211	3202
g7jac200sc1	44611	22822	41198		19554	20384	19248
hydr1	5308	2726	5308	559	337	334	324
impcol_d	425	153	241	219	123	117	102
jan99jac020s	6774	4708	6264		5016	4891	5885
lhr71c	58291	18181	27064	5802	3620	3771	3893
mark3jac140	6825	12126	6550		6106	6112	6159
poli_large	15575	6400	15575		6381	6316	5995
radfr1a	1048	95	970	142	71	98	147
rdist1	4134	195	3421	336	223	215	175
sinc15	11532	9686	11532	11532	11036	9623	10833
Zhao2	33861	33861	1476		1464	1476	565

TABLE 9.4
The total bandwidth for the HC, NCHC, RDO, and RCM ordering algorithms applied to the diagonal blocks of the block triangular form.

Identifier	Initial	HC	NCHC	RDO	RCM			
					$A+A^T$	AA^T(+RDO)	$\hat{A}$	A
4cols	11770	5134	3281	4768	846	**460** (1001)	565	504
10cols	29496	13801	6530	13855	1052	546 (1600)	572	**528**
bayer01	48803	48860	22394	34581	3483	**1768** (3056)	1823	**1776**
bayer03	4776	4792	2279	3617	740	527 (547)	**500**	**506**
circuit_3	7607	7607	3698	4776	1903	**1330** (1394)	**1321**	**1297**
ethylene-1	8336	8336	3540	4967	323	**179** (432)	**184**	230
extr1	2413	2413	1114	1660	240	**145** (266)	**149**	**148**
fidapm11	19515	7168	3880	3315	**3189**	3299 (3987)	**3211**	**3240**
g7jac200sc	44611	**16548**	**16149**	22822	41198	19554 (19279)	20384	19248
hydr1	2370	2370	1151	1640	198	129 (**112**)	134	129
impcol_d	199	194	133	70	98	79 (82)	67	**59**
jan99jac020s	6774	6478	**4046**	4708	6264	5016 (5321)	4891	5652
lhr71c	7663	7663	2911	5135	991	741 (2173)	**727**	**720**
mark3jac140	6825	**5055**	5857	12126	6550	6106 (8846)	6112	6123
poli_large	90	85	**59**	84	90	90 (79)	84	77
radfr1a	621	186	98	132	130	88 (93)	**85**	93
rdist1	341	129	**120**	155	346	188 (193)	189	192
sinc15	10880	10880	**9195**	10880	10880	10491 (10880)	10880	10880
Zhao2	33861	33861	14015	33861	1471	**1454** (2196)	**1467**	**1424**

Table 9.4 shows the effect of applying the ordering algorithms to the diagonal blocks of the block triangular form (2.2). As already noted, the construction of the block triangular form ensures that there are no zeros on the diagonal, so we do not preorder using MC21. Apart from this, the algorithms featured in Table 9.3 are featured here too. We also show results for the hill-climbing algorithm (HC) and hill-climbing plus the node-centroid algorithm (NCHC). For the node-centroid algorithm we have experimented with using values of λ in the range $[0.8, 1]$ and values of α in the range $[1.5, 2.5]$. Our experience was that the bandwidths were not very sensitive

TABLE 9.5

The total bandwidth after hill climbing and the node-centroid algorithm. All are applied to the diagonal blocks of the block triangular form.

Identifier	RCM + HC				RCM + NCHC			
	$A+A^T$	AA^T	$\hat{A}$	A	$A+A^T$	AA^T	$\hat{A}$	A
4cols	718	435	549	481	502	**395**	458	443
10cols	902	498	553	479	625	**448**	462	**447**
bayer01	3241	1739	1742	1756	2243	**1659**	**1675**	**1659**
bayer03	668	446	445	452	411	**381**	**384**	**377**
circuit_3	1715	1228	1227	1123	1356	**1065**	**1074**	**1095**
ethylene-1	271	172	173	216	174	169	**162**	203
extr1	190	119	120	131	130	**115**	119	**116**
fidapm11	**3154**	3261	3183	**3156**	**3123**	3336	3286	**3085**
g7jac200sc	37042	19244	19782	18660	22290	**17383**	**17451**	**17530**
hydr1	133	101	101	120	**89**	**91**	**91**	**89**
impcol_d	66	61	56	55	**50**	51	**49**	52
jan99jac020s	5758	4258	4401	5190	3883	3665	**3249**	3953
lhr71c	862	626	598	576	**540**	572	557	**540**
mark3jac140	6192	**6035**	**6044**	**6053**	**5951**	**5959**	**5946**	**5978**
poli_large	56	70	70	66	54	**50**	61	**52**
radfr1a	**57**	63	72	76	**58**	**58**	**57**	**58**
rdist1	148	133	156	158	**123**	**121**	**124**	**119**
sinc15	10880	8819	10880	10880	**7428**	8866	10648	8097
Zhao2	1471	1454	1467	**1420**	1473	**1446**	**1462**	**1442**

to the precise choice of λ, and for most examples 0.85 gave results that were within three percent of the best. For α, we found that a value of 2 gave slightly narrower bandwidths than either 1.5 or 2.5. We therefore used $\lambda = 0.85$ and $\alpha = 2$.

In Table 9.4 we have highlighted the narrowest bandwidths and those within three percent of the narrowest. As expected, the larger symmetry index for the diagonal blocks of the block triangular form results in an improvement in the performance of RCM applied to $A + A^T$, but it is still better to use the other RCM variants. There appears to be little to choose between RCM applied to the row graph, RCM applied to the bipartite graph, and our Unsymmetric RCM algorithm; for some of the examples, each produces the narrowest total bandwidth. In general, combining hill-climbing with the node-centroid algorithm is better than using hill-climbing alone, but this is not guaranteed. For a small number of problems (including `pol_large` and `rdist1`), the NCHC ordering has the smallest total bandwidth, but for many of the test examples it gives results that are significantly poorer than the RCM variants.

To see whether RDO can be successfully used to refine our RCM orderings, we have experimented with running RDO after RCM applied to AA^T; the results are in parentheses in Table 9.4 in the column headed AA^T(+RDO). For a number of problems (including `poli_large`) the bandwidth is reduced, but for others the results are much worse (for example, `4cols` and `bayer01`). Indeed, using RDO after RCM can be worse than using RDO on the original ordering (for example, `jan99jac020s` and `rdist1`). This illustrates that RDO is extremely sensitive to the initial ordering, and our findings lead us not to recommend its use.

In Table 9.5, we present results for applying the different RCM variants to the block triangular form, followed by applying either hill climbing alone (denoted by RCM + HC) or the node-centroid algorithm plus hill climbing (denoted by RCM + NCHC). Again, the narrowest total bandwidths (and those within three percent of the narrowest) are highlighted. Comparing the results in columns 2–5 of Table 9.5 with the corresponding results in Table 9.4, we see that hill climbing (which never increases the total bandwidth) can significantly improve the RCM orderings. However,

TABLE 9.6
The best and worse total bandwidths using the given ordering and nine random permutations.

Identifier	RCM + NCHC			
	$A + A^T$	AA^T	$\hat{A}$	A
4cols	[457,570]	[384,456]	[385,463]	[380,460]
10cols	[575,625]	[448,463]	[445,462]	[438,461]
bayer01	[1963,2243]	[1659,1680]	[1665,1694]	[1655,1682]
bayer03	[410,463]	[380,400]	[374,401]	[368,397]
circuit_3	[1298,1356]	[1033,1105]	[1041,1159]	[969,1106]
ethylene-1	[169,184]	[156,169]	[158,169]	[157,211]
extr1	[119,134]	[114,127]	[114,131]	[114,125]
fidapm11	[3119,3288]	[3205,3336]	[3172,3290]	[3085,3252]
g7jac200sc	[18849,22290]	[16793,18132]	[16460,18516]	[16396,17827]
hydr1	[89,98]	[89,93]	[89,92]	[88,93]
impcol_d	[50,57]	[43,56]	[42,54]	[46,52]
jan99jac020s	[3396,4107]	[3257,3665]	[3230,3724]	[3209,3953]
lhr71c	[540,633]	[555,600]	[554,578]	[506,700]
mark3jac140	[5951,7385]	[5909,5959]	[5946,6062]	[5941,6011]
poli_large	[49,55]	[50,59]	[55,66]	[49,53]
radfr1a	[56,62]	[58,58]	[58,62]	[58,61]
rdist1	[120,127]	[121,126]	[121,125]	[119,125]
sinc15	[6903,8652]	[7752,8866]	[8265,11532]	[7413,8523]
Zhao2	[1473,2055]	[1446,1467]	[1453,1464]	[1442,1521]

looking also at columns 6–9, it is clear that for all problems except fidapm11 and Zhao2 (the two nearly symmetric problems), the smallest bandwidths are achieved by using RCM + NCHC. For problems with an unsymmetric sparsity structure, the largest improvements resulting from using the node-centroid algorithm are to the orderings obtained using RCM applied to $A + A^T$; for some problems (including the bayer examples and lhr71c) the reductions resulting from including the node-centroid algorithm are more than 30 percent. However, for many of our unsymmetric examples, one of the other variants generally produces orderings with a smaller total bandwidth.

Finally, we note that for a small number of problems, none of our proposed algorithms was successful in significantly reducing the bandwidth. In particular, we were not able to reorder the problems g7jac200sc, jan99jac020s, mark3jac140, and sinc15 to have a small bandwidth. We are not able to predict a priori which problems we are able to reorder to have a small bandwidth using our algorithms.

9.3. The effect of random initial permutations. Finally, we tried applying the algorithms after applying random row and column permutations to the given matrix ordering. The results are shown in Table 9.6. It is indeed the case that better total bandwidths can often be found in this way, which points the way towards finding better algorithms. Meanwhile, if many problems with the same structure are to be solved (so that the cost of reordering may be amortized over the repeated factorizations), it may be worthwhile to perform such random permutations and take the best resulting ordering. The conclusion that we drew from Table 9.5, that there is little to help us choose between the algorithms of the final three columns, is true here too.

10. Concluding remarks. We have considered algorithms for reducing the lower and upper bandwidths l and u of an unsymmetric matrix A, focusing on the total bandwidth, which we have defined as $l + u + \min(l, u)$, because this is relevant for the storage and work when sets of banded linear equations are solved by Gaussian elimination.

The least satisfactory results came from working with the lexicographical method of Baumann, Fleischmann, and Mutzbauer [1] and with the pattern of $A + A^T$, although for unsymmetrically structured matrices the use of the unsymmetric node-centroid algorithm plus hill climbing dramatically improved the results of applying the reverse Cuthill–McKee ordering to $A + A^T$. For the majority of our test problems, we achieved good results by applying the RCM algorithm to the matrices AA^T (whose graph is the row graph) and $\left[\begin{smallmatrix} 0 & A \\ A^T & 0 \end{smallmatrix}\right]$ (whose graph is the bipartite graph). Our unsymmetric variant of RCM gave comparable results. The results were improved by preordering A to block triangular form and applying one of these three RCM-based algorithms to the blocks on the diagonal. The rest of the matrix is used unaltered. The bandwidths were further reduced using our unsymmetric node-centroid and hill-climbing algorithms.

In general, the time taken to reorder an unsymmetric matrix using our algorithms is significantly less than the time required to subsequently factorize the matrix. However, since the codes used to generate the numerical results presented in this paper are prototypes, we have not reported the reordering times. In the future, we plan to include carefully designed efficient implementations of our new algorithms within the mathematical software library HSL [8].

Acknowledgments. We are grateful to Iain Duff of the Rutherford Appleton Laboratory and Yifan Hu of Wolfram Research for helpful comments on a draft of this paper, and to the anonymous referees for their suggestions.

REFERENCES

[1] M. Baumann, P. Fleischmann, and O. Mutzbauer, *Double ordering and fill-in for the LU factorization*, SIAM J. Matrix Anal. Appl., 25 (2003), pp. 630–641.

[2] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the 24th National Conference of the ACM, Brandon Systems Press, 1969, pp. 157–172.

[3] T. Davis, *University of Florida Sparse Matrix Collection*, NA Digest, 97, 1997; full details from www.cise.ufl.edu/research/sparse/matrices/.

[4] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Softw., 15 (1989), pp. 1–14.

[5] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 889–901.

[6] A. George, *Computer Implementation of the Finite-Element Method*, Ph.D. thesis, Department of Computer Science, Report STAN CS-71-208, Stanford University, Stanford, CA, 1971.

[7] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer, *An algorithm for reducing the bandwidth and profile of a sparse matrix*, SIAM J. Numerical Analysis, 13 (1976), pp. 236–250.

[8] *HSL, A collection of Fortran codes for large-scale scientific computation*, 2004; online at http://hsl.rl.ac.uk/.

[9] Y. F. Hu and J. A. Scott, *Ordering techniques for singly bordered block diagonal forms for unsymmetric parallel sparse direct solvers*, Numer. Linear Algebra Appl., 12 (2005), pp. 877–894.

[10] A. Lim, B. Rodrigues, and F. Xiao, *A centroid-based approach to solve the bandwidth minimization problem*, in Proceedings of the 37th Hawaii International Conference on System Sciences, IEEE Press, Piscataway, NJ, 2004, p. 30075a.

[11] B. H. Mayoh, *A graph technique for inverting certain matrices*, Math. Comp., 19 (1965), pp. 644–646.

[12] J. K. Reid and J. A. Scott, *Ordering symmetric sparse matrices for small profile and wavefront*, Internat. J. Numer. Methods Engrg., 45 (1999), pp. 1737–1755.

[13] J. A. Scott, *A new row ordering strategy for frontal solvers*, Numer. Linear Algebra Appl., 6 (1999), pp. 1–23.

[14] J. A. Scott, *Row ordering for frontal solvers in chemical process engineering*, Comput. Chem. Eng., 24 (2000), pp. 1865–1880.