Symmetric orderings for unsymmetric sparse matrices

H.A.G. Wijshoff

RUU-CS-91-19
June 1991

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Symmetric orderings for unsymmetric sparse matrices

H.A.G. Wijshoff

Technical Report RUU-CS-91-19
June 1991

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Symmetric Orderings for Unsymmetric Sparse Matrices*

Harry A.G. Wijshoff

Department of Computer Science
Utrecht University

Abstract

The efficient solution of large sparse systems of linear equations is one of the key tasks in many computationally intensive scientific applications. The parallelization and efficiency of codes for solving these systems heavily depends on the sparsity structure of the matrices. As the variety of different structured sparse matrices is large, there is no uniform method which is well suited for the whole range of sparse matrices. Investigations in the problem of parallelizing direct methods for solving large, structurally unsymmetric, sparse systems of linear equations has led to the development of a collection of new ordering algorithms (heuristics) for structurally unsymmetric, sparse matrices.

1 Introduction

Implementations of direct solvers for solving general sparse linear systems of equations:

$$Ax = b$$

mostly perform very poorly on vector/parallel architectures. This is mainly caused by the fact that the exploitation of parallelism and vectorization of such a code is strongly dependent on the sparsity structure of the matrix A. As the sparsity pattern of the matrix A is assumed to be arbitrary and unstructured, the utilization of any parallelism and/or vectorization will lead in most cases to very inefficient code. It is here that orderings can play an important role, because they transform the sparsity structure of a sparse matrix into a special form which renders the possibility of efficient parallelization or vectorization. There are essentially two types of orderings, unsymmetric orderings and symmetric orderings.

Definition 1.1
i) An ordering of a sparse matrix A is called symmetric *if the ordering can be represented by*

$$\tilde{A} = PAP^T,$$

*This work was done in part while the author was visiting CSRD, University of Illinois, 305 Talbot Lab., 104 S. Wright St., Urbana, Illinois 61801 under funding of the National Science Foundation under Grant No. US NSF CCR-8717942, and the US Department of Energy under Grant No. US DOE DE-FG02-85ER25001.

with P a permutation matrix.
ii) An ordering of a sparse matrix is called unsymmetric *if the ordering can be represented by*

$$\tilde{A} = PAQ^T,$$

with P and Q permutation matrices.

Note that symmetric ordering have the property that the associated graphs of A and $\tilde{A}$ are isomorphic, i.e., only the numbering of the nodes differs. For a formal definition of the associated graph of a sparse matrix, see the next section. Unsymmetric orderings are obtained by independent row and column interchanges of the matrix, with P representing the row interchanges and Q representing the column interchanges. So, where the unsymmetric orderings change certain properties of the sparse matrix, like, for instance, the eigenvalues, symmetric orderings maintain these. Also, if A is a diagonally dominant matrix, then after a symmetric ordering the resulting matrix will still be diagonally dominant, whereas an unsymmetric ordering destroys this property. One can very well ask whether it makes any sense to maintain properties of an arbitrary sparse matrix A, which in most cases does not have any of these nice properties. However, as will be pointed out in section 4, these symmetric orderings can be preceded by possibly an unsymmetric ordering which brings the matrix into a more (numerically) acceptable form, which is maintained throughout the symmetric ordering.

In this paper we will describe some new algorithms (heuristics) for symmetric orderings which bring a general sparse matrix into "bordered upper triangular block form". The advantage of such a form is that it allows large grain parallelism when computing the LU factor of the matrix. Orderings for unsymmetric[1] matrices received less attention in the literature than orderings for symmetric matrices. Lately a number of unsymmetric orderings for unsymmetric matrices have been proposed [EGL$^+$87], which bring a matrix in bordered upper triangular block form. These orderings, however, do not allow an effective factorization. This is caused by the fact that the numerical values of the entries of the matrix are not taken into account when ordering a matrix. The orderings as described in this paper are based on two algorithms commonly used for ordering matrices: *Tarjan's algorithm* [Tar72] and *Nested, One-way Dissection* [Geo73]. From these two orderings which are very different in nature (Tarjan's algorithm brings an unsymmetric matrix into lower triangular block form and Nested, One–way Dissection brings a symmetric matrix into arrowhead form, i.e., doubly bordered diagonal block form) two orderings are derived **H1** and **H2** which both bring an unsymmetric matrix in bordered upper triangular block form. The composition of these two orderings appears to be very effective and is used in the hybrid ordering: **H***, which will be described in section 4.

2 The H1 Ordering

For describing the orderings in this section and in the other sections we need some graph theoretical concepts. We will briefly introduce the relationship between sparse matrices and graphs. For a more detailed account of this relationship the reader is referred to [BM76, DER86].

[1]Whenever we refer to a sparse matrix as (un)symmetric we mean throughout the paper: (un)symmetric in structure.

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & x & & x & & x \\ 2 & & x & x & & \\ 3 & & & x & x & \\ 4 & x & & & x & \\ 5 & x & x & & & x \end{array}$$

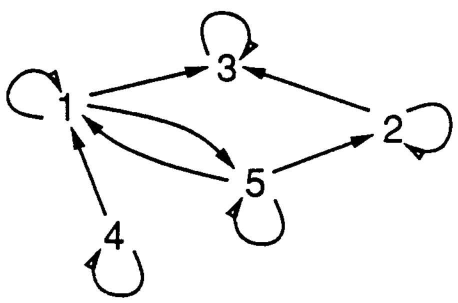


Figure 1: A 5 x 5 sparse matrix and its associated digraph.

Definition 2.1 *Given an unsymmetric (square) sparse matrix A ($N \times N$). The digraph associated with A is defined to be the graph $G(V, E)$ with $|V| = N$ such that $(x, y) \in E$ if and only if (x, y) is a non-zero entry in A.*

In other words A is the adjacency matrix of $G(V, E)$. An example of a 5 x 5 sparse matrix and its associated digraph are depicted in figure 1. A symmetric ordering is nothing else than a suitable numbering of the nodes of the associated digraph.

Tarjan's algorithm [Tar72] for bringing a sparse matrix into lower triangular block form is an efficient algorithm ($O(|V| + |E|)$) for finding the strongly connected components of the associated digraph.

Definition 2.2 *Given a digraph $G(V, E)$. A strongly connected component is a maximal connected subgraph $G'(V', E')$ of G such that all paths from one node in G' to another node in G' include only nodes from G'.*

The algorithm is based on a depth-first search on the graph G and makes use of a stack ST to store the current, strongly connected component. Whenever a strongly connected component is found the stack is emptied and the search for a new strongly connected component starts. A strongly connected component is found whenever all edges from any node of a subtree of the depth first search tree point only to nodes which are contained in that subtree. To verify this condition a node α of the tree is labeled with a *low* value indicating "the highest node in the tree which is pointed to by a node of the subtree with root α". If the depth-first search traverses in the upward direction (backtrack) the *low* value of the current node is updated by the minimum of the *low* value it already had and the *low* value of its son. Whenever the *low* value is equal to the *numb* value and all neighbors of a node have been considered, the condition of a strongly connected component is met.

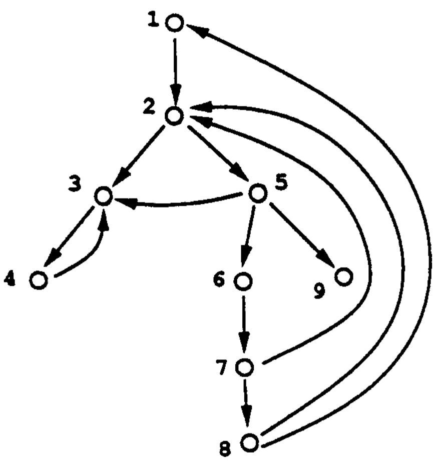
Tarjan's Algorithm

```
$i = 1$
1 Choose an arbitrary node $\alpha$ of $G$ which has not been considered yet
(If all nodes have been considered then the algorithm is finished);
2 Push $\alpha$ on the stack $ST$ and assign:
$numb(\alpha) = low(\alpha) = i$;
$i = i + 1$;
3 if not all neighbors of $\alpha$ have been considered yet;
  then
    Take a neighbor $\beta$ of $\alpha$ which has not been considered yet;
    if $\beta$ is on the stack;
      then
        $low(\alpha) = \min(low(\alpha), numb(\beta))$;
        goto 3;
      else ($\beta$ is not numbered yet)
        $\alpha = \beta$;
        goto 2;
  else
    if $low(\alpha)$ is equal to $numb(\alpha)$;
      then
        STRONGLY CONNECTED COMPONENT FOUND;
        remove $\alpha$ and all entries on top of $\alpha$ from the stack;
        discard all edges pointing to these nodes;
4 if stack is not empty;
  then
    take $\beta$ which caused $\alpha$ to be pushed on the stack and assign:
    $low(\beta) = \min(low(\alpha), low(\beta))$; and
    $\alpha = \beta$ (backtrack);
    goto 3;
  else
    goto 1;
```

In figure 2 the contents of the stack is depicted during the Tarjan's algorithm on a sample graph. Of every pair i–j, the first number is the *numb* value of a node and the second number indicates its *low* value. Below each stack the corresponding step of the algorithm is shown (for instance, 3te should be read as "the else–clause of the then–clause of step 3).

A disadvantage of this ordering is that most sparse matrices do not allow a nice decomposition into strongly connected components[2]. This is particularly the case whenever the matrix contains a large cycle. This is caused by the fact that, if the associated digraph has a long cycle $\alpha_1, \alpha_2, \ldots, \alpha_n, \alpha_1$, then all the nodes of this cycle have to be in one strongly connected component. In figure 3 and figure 4 a sparse matrix (west2021 from the Harwell/Boeing collection [DGLP82]) is depicted together with the reordered matrix after using Tarjan's algorithm combined with a transversal algorithm to obtain a zero–free diagonal (see section 4). As can be seen the resulting matrix has one very large diagonal

[2]Strongly connected components form a unique decomposition of a graph.



```
                                                                                     9-9  9-9
                                                  8-8  8-2  8-1  8-1  8-1  8-1  8-1  8-1  8-1  8-1  8-1  8-1
                                             7-7  7-7  7-7  7-7  7-1  7-1  7-1  7-1  7-1  7-1  7-1  7-1  7-1
               4-4  4-3  4-3            6-6  6-6  6-6  6-6  6-6  6-6  6-1  6-1  6-1  6-1  6-1  6-1  6-1  6-1
          3-3  3-3  3-3  3-3       5-5  5-5  5-5  5-5  5-5  5-5  5-5  5-5  5-5  5-1  5-1  5-1  5-1  5-1  5-1
     2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-2  2-1  2-1
1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1  1-1
-------------------------------------------------------------------------------------------------------------
2    2    2    2    3tt  3et  4t   2    2    2    2    3tt  3tt  4t   3tt  4t   4t   2    3et  4t   4t   3et
   3te  3te  3te                 3te  3te  3te  3te                                3te
-------------------------------------------------------------------------------------------------------------
1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20   21   22
```

Figure 2: The stack contents during Tarjan's algorithm on a sample graph.

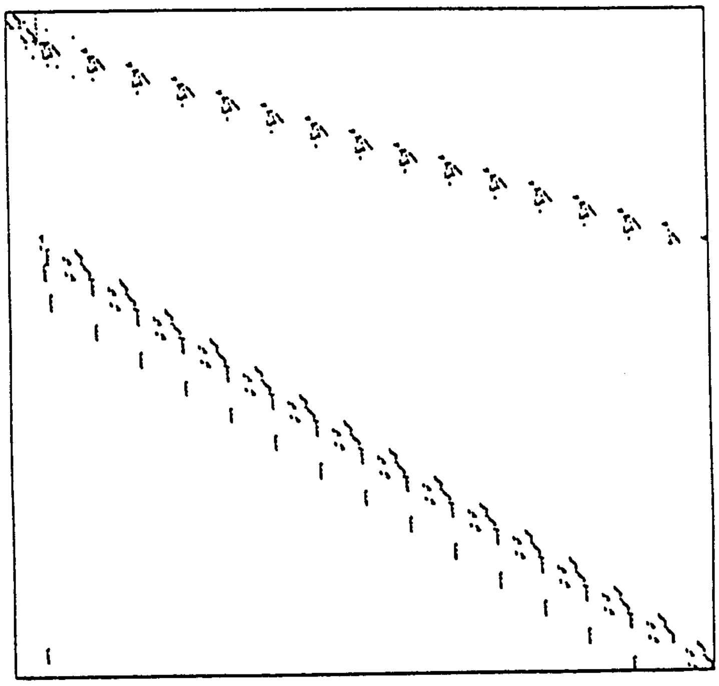


Figure 3: The west2021 matrix of the Harwell/Boeing collection.

block. This leads to a bad load balance whenever the diagonal block structure is used to parallelize the LU factorization.

Because of the above mentioned we developed an ordering **H1** based on Tarjan's algorithm, which extracts from the digraph a small set of nodes such that the remaining graph allows a better decomposition into strongly connected components. In the ordering the size of each strongly connected component is monitored during construction, and, whenever the size grows larger than a certain threshold (Tsize), an attempt is made to delete a small number of nodes from the graph such that the strongly connected component will not grow any further. The number of nodes to be removed from the graph for each strongly connected component has to be smaller than another threshold (Tseparator), which is mostly expressed as a fraction of the size of the current strongly connected component. The total set of nodes which have been removed from the graph will form a border in the resulting matrix when they are numbered last.

Suppose in Tarjan's algorithm the *numb* and *low* value of a current node α equals N and M respectively ($M < N$). In this case $N - M$ gives an indication of how many more nodes will be added to the nodes of the subtree of α before a strongly connected component is found. On the other hand, if all the nodes on the path from $low(\alpha)$ to α, not including α itself, were deleted from the graph, then either, the *low* value of α would become equal to α, or the *low* value of α would point to a node β for which the *low* and *numb* values would become equal. The first case happens when the nodes on the path from $low(\alpha)$ to α are numbered consecutively. If this condition is not met, the latter case could happen. In figure 5 these 2 cases are shown. For both cases the nodes numbered 1 and 2 are removed from the graph. In the first case the deletion of the nodes would result into α being the root of a strongly connected component. In the second case, more than one strongly connected component will be formed. However, because of the numbering, viewing the remaining nodes on the stack as one strongly connected component will still maintain the single strongly connected components as separate diagonal blocks in the

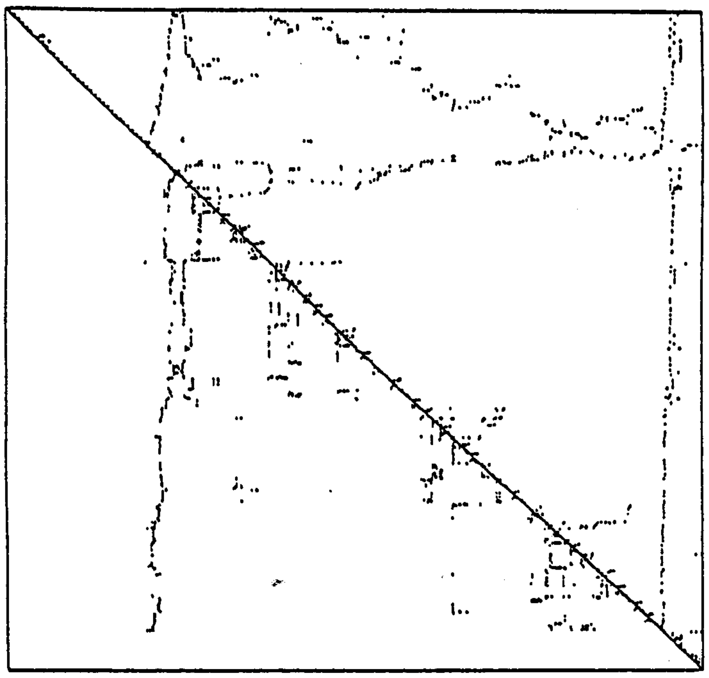


Figure 4: west2021 reordered by Tarjan's algorithm.

reordered matrix. To distinguish between "real" strongly connected components and the ones obtained by removing nodes from the graph, we call the later ones cut connected components.

Based on the previous observation a simple implementation of **H1** is obtained. This implementation requires checking the conditions whether the size M of the current strongly connected component is larger than Tsize and the condition that the number of nodes to be removed is smaller than Tseparator times M. In case 2 of figure 5 one can see that the contents of the stack would be: $1, 2, 3, 4, 5, 6, 7, 8, 9$ and $\alpha = 5$. The nodes of the current strongly connected component are $3, 4, 5, 6, 7, 8, 9$ and the nodes to be removed are $1, 2$. So, just taking the number of nodes on top of α as the size of the current strongly component will not work in this case. Hence, in order to monitor the size of the current strongly connected component Tarjan's algorithm is modified so that each element is removed from the stack and stored in a set CSCcomponent, whenever backtracking occurs. The resulting implementation of **H1** is obtained by changing step **3** and **4** of Tarjan's algorithm to:

```
3 if not all neighbors of $\alpha$ have been considered yet;
   then
      Take a neighbor $\beta$ of $\alpha$ which has not been considered yet;
      if $\beta$ is on the stack;
         then
            $low(\alpha) = \min(low(\alpha), numb(\beta))$;
            goto 3;
         else ($\beta$ is not numbered yet)
            $\alpha = \beta$;
            goto 2;
   else
      if $low(\alpha)$ is equal to $numb(\alpha)$;
         then
            STRONGLY CONNECTED COMPONENT FOUND;
            remove $\alpha$ from the stack;
            $\alpha$ together with the nodes of CSCcomponent form the
            strongly connected component;
            discard all edges pointing to these nodes;
            make CSCcomponent empty;
         else
            if the number ($M$) of elements in CSCcomponent
            is larger than Tsize;
               then
                  if the number of nodes on the stack
                  on top of $low(\alpha) <$ Tseparator $\times M$ (*);
                     then
                        CUT CONNECTED COMPONENT FOUND;
                        move all the nodes in the stack
                        below $\alpha$ and above $low(\alpha)$
                        (including $low(\alpha)$) to the border;
                        $\alpha$ together with the nodes of CSCcomponent
                        form the cut connected component;
                        remove $low(\alpha)$ and all entries on top
                        of $low(\alpha)$ from the stack;
                        discard all edges pointing to these nodes;
                        make CSCcomponent empty;
4 if stack is not empty;
   then
      take $\beta$ which caused $\alpha$ to be pushed on the stack and assign:
      $low(\beta) = \min(low(\alpha), low(\beta))$;
      remove $\alpha$ from the stack and store it in CSCcomponent;
      $\alpha = \beta$ (backtrack);
      goto 3;
   else
      goto 1;
```

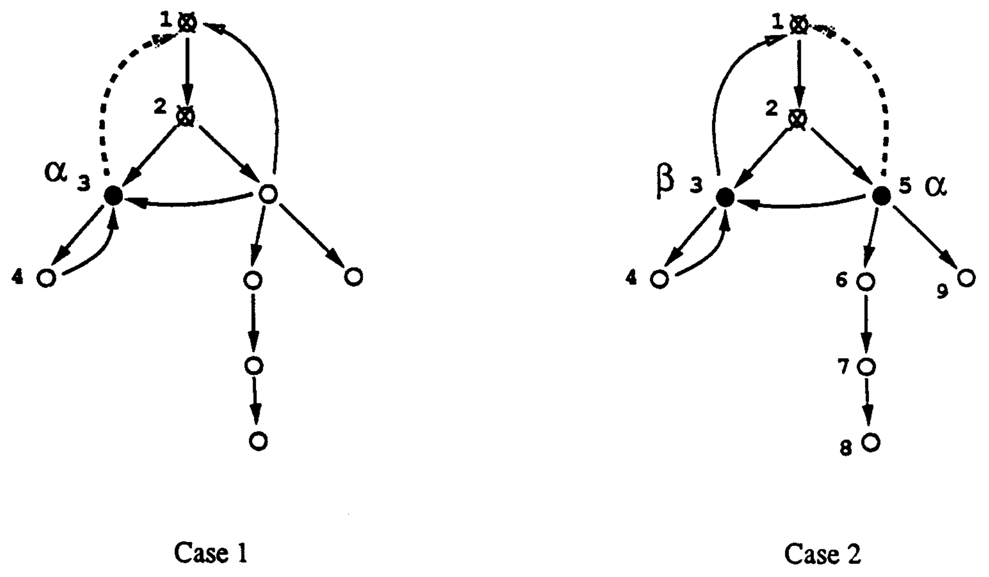


Figure 5: The 2 different cases when deleting nodes on the path from *low*(α) to α.

(*) Note that because the nodes on which backtracking occurred are all removed from the stack the number of nodes on top of *low*(α) equals the number of elements to be removed from the graph exactly.

The above described algorithm still does not give a satisfactory solution whenever the graph contains a large cycle, for instance. In this case the algorithm will push the cycle onto the stack and by descending the stack it will at some point (when the threshold is reached) put the remaining elements of the cycle in the border. This will affect the size of the original strongly connected component only marginally. For these cases there are two possible solutions. One solution is obtained by calling this algorithm recursively for the remaining graph (the graph with the border nodes deleted). In the case of the large cycle a second pass through the graph will cut the cycle at any given point as the first pass already cut the cycle once. The recursive calls, however, are very costly because the size of the remaining graph does not reduce significantly with each step.

The main reason for this algorithm to be not optimal is the fact that removing all the nodes in between α and *low*(α) from the graph is not really necessary as only these nodes have to be removed which are *pointed to* by nodes in the CSCcomponent. So, another solution is given by a different variant of this algorithm. For this variant the neighbors of a node which are lower in the stack than a particular threshold (Tpoint) are ignored. To be specific in the **then** clause of the first **then** clause in step **3** of **H1** *low*(a) is only updated if *low*(a) – *numb*(β) is smaller than Tpoint. After this the detection of the cut connected component is a little bit more tedious. All the nodes of the CSCcomponent have to be checked whether they have neighbors in the stack lower than *low*(α) in which case these neighbors need to be added to the border. This search has to be done only for these nodes of CSCcomponent which have not been updated once in step **3**. This can be easily achieved by marking these nodes whenever this occurs. So, the second **then** clause

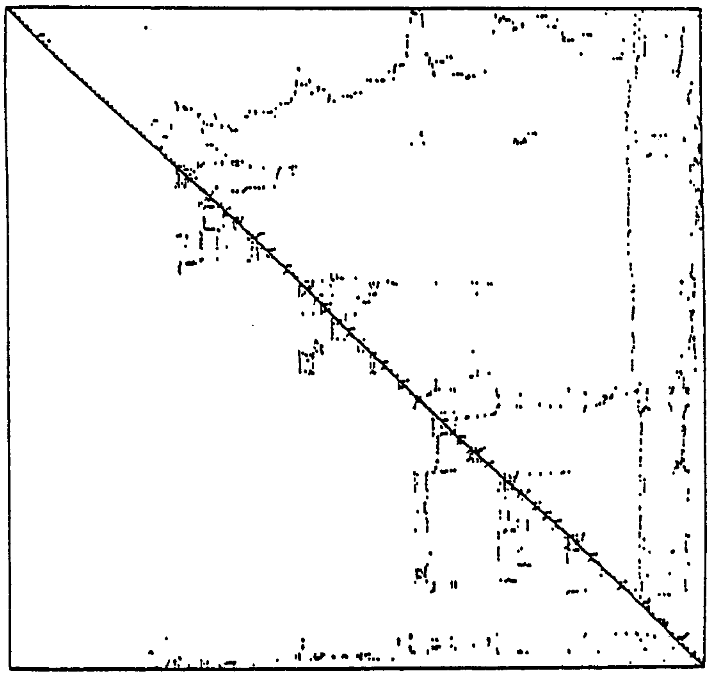


Figure 6: west2021 reordered by **H1**.

becomes:

```
then
  if $low(\alpha) - numb(\beta) <$ Tpoint;
    then
      $low(\alpha) = \min(low(\alpha), numb(\beta))$;
    else
      mark $\alpha$ as to be still checked yet;
  goto 3;
else ....
```

By adding the additional elements the border can grow too large in which case the construction is canceled and the algorithm proceeds with the next element in the stack. Note that with this variant large cycles (or cycles with arcs) will be decomposed. The worst case complexity of this variant is trivially bounded by $\Omega(|V|^2 + |E|)$, however the expected time for this algorithm is far less.

In figure 6 the result of this implementation of the ordering **H1**, with parameters Tsize $= N/10$, Tseparator $= 1/10$ and Tpoint $= 6$, is shown for the same matrix as in figure 3.

The algorithms as described above allows a simple change to control the absolute values of the entries in the border. By taking for each node as the first neighbor the one which belongs to the largest absolute entry in the matrix, the values in the border could be minimized. This will not always succeed, however. In the next sections other ways are described to minimize the absolute values of the entries in the border.

3 The H2 Ordering

The **H2** ordering is related to the Nested (One-way) Dissection algorithms. Nested dissection was introduced by A. George for finite element matrices [Geo73, GL81], and is now widely used for structurally symmetric matrices. This ordering is based on the construction of separator sets.

Definition 3.1 *Given a graph $G = (V, E)$ a* separator *set S of G is a subset of V such that there exists sets B and C with*
a) B, C and S disjoint,
b) $B \cup S \cup C = V$, and
c) there exist no edges $(x, y) \in E$ with $x \in B$ and $y \in C$ (or $y \in B$ and $x \in C$).

Whereas nested dissection works very well for regular grid type matrices in general this ordering will not always produce a nicely structured matrix. This is particularly the case when this ordering is used for structurally unsymmetric matrices. First structurally unsymmetric matrices are not likely to arise from regular grids. Secondly in order to use nested dissection the unsymmetric matrix A is expanded to $A + A^T$. In case that A is very unsymmetric the number of non-diagonal, non-zero entries will grow with approximately a factor of 2 making the interconnection structure of the associated digraph more interwoven. These considerations lead to the development of a second ordering **H2** which will exploit the unsymmetricity of the matrix.

The ordering **H2** is based on the construction of separator sets of the adjacency matrix of $A + A^T$. In fact the initial algorithm used for finding these separator sets does not appear to be very important. For the implementation of **H2** we used a straight-forward implementation of automatic nested dissection [GL81]. However, other initial orderings could have been used such as one-way dissection, more sophisticated implementations of automatic nested dissection [LRT79], or the graph bisection heuristics as proposed by [LL87]. Nested dissection will bring the matrix into arrowhead form. As the objective of the **H2** ordering is to bring the matrix into bordered upper triangular block form, nested dissection seems to be too rigorous. It is this fact which is exploited by the **H2** ordering. After each stage when a separator set S is constructed the ordering **H2** will reduce the number of nodes in S thereby allowing additional fill-in to be created in the upper triangular part of the matrix. This is established by moving nodes of the set S to either B or C depending on what case they are in. For each node α of the set S we can distinguish 3 cases:

1. there are no edges (α, β) with $\beta \in B$
2. there are no edges (β, α) with $\beta \in C$
3. α is neither in case 1 or 2.

If α is in case 1 then α is moved to C and if α is in case 2 then α is moved to B. If α is in case 3 it has to remain in S. In figures 7, 8, and 9 the three different cases are shown together with the corresponding actions (node α is moved to α' transforming edges e, f, and g to e', f', and g'). Notice that these transfer will create only edges from the set B into C. These edges will account for the fill-in in the upper triangular part of the resulting matrix. The algorithm updates each time, when it moves a node out of the set S, the cases of its neighbors in S as these can be affected by this transfer. Let's for the argument sake assume that the probabilities of an edge pointing to the "left" or to the right "right" are almost equal, which is not unreasonable as the separator sets are constructed on $A + A^T$. Then it can be seen that the higher the degree of a node in S is the more likely it will be that this node cannot be moved to either B or C. This explains

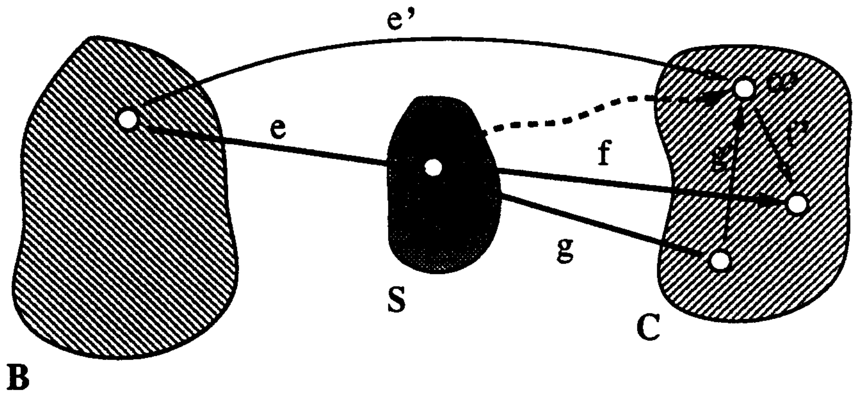


Figure 7: Case 1: α is moved to set C.

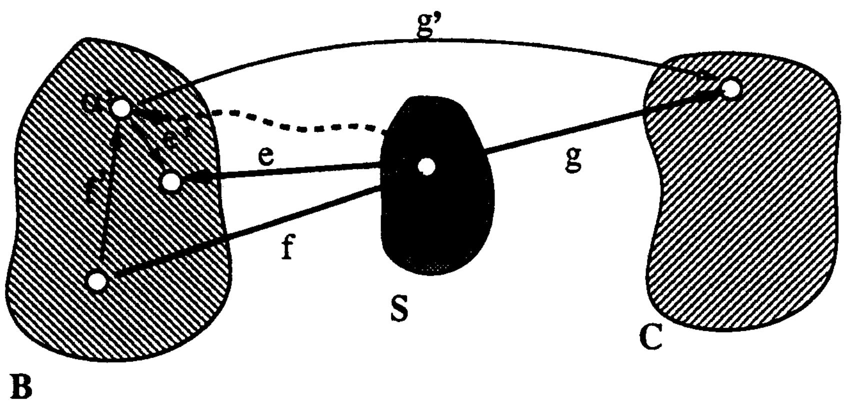


Figure 8: Case 2: α is moved to set B.

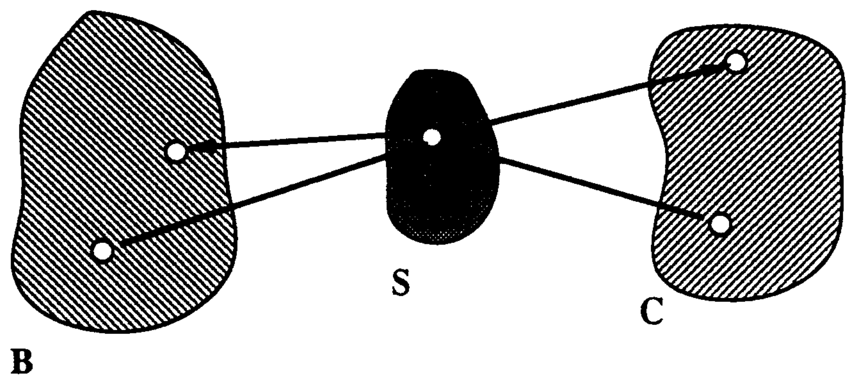


Figure 9: Case 3: α cannot be moved.

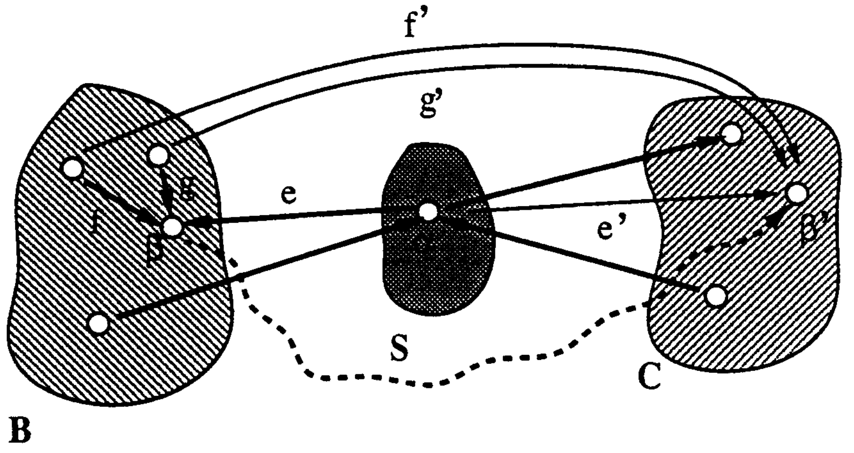


Figure 10: A neighbor β of α is moved from set B to set C.

why the initial algorithm used for finding the separator sets does not have to be as optimal. This is caused by the fact that good (small) separator sets are likely to contain nodes with high degree.

The reduction of the separator sets can be optimized. For each node α of S which is in case 3, its neighbors could be transferred from set B to C and visa versa, such that α will get into case 1 or 2 and can be moved out of the separator set. This can only be done for these neighbors of α in B (C) which have only incoming (outgoing) edges. In figure 10, a neighbor β of α in set B, which has only incoming edges (e, f, and g), is moved to set C creating edges e', f', and g'. By moving this neighbor the state of α is changed from 3 to 1. The state of α is only changed, when all the nodes in set B which are pointed to by α can be moved to C, or, when all the nodes in set C which are pointing to α can be moved to set B. Note that, because of the condition that only nodes in B (C), of which all the edges are incoming (outgoing), are moved, the edges which are introduced between B and C will all point from B to C. So, the resulting algorithm is:

```
1 Find an initial separator set $S$ for $A + A^T$
2 Reduce $S$ as much as possible by moving nodes
  which are in case 1 or 2 to $B$ or $C$
3 Move each node $\beta$ of $B$ which satisfies:
    i) there is a node $\alpha$ in $S$ which points to $\beta$, and
    ii) there are no nodes in $B$ which are pointed to by $\beta$
  to $C$ provided that $|B|$ is greater than some threshold
  repeat 2
4 repeat the algorithm on $B$ respectively $C$ until $B$ and $C$
  have the desired size
```

Instead of moving nodes from B to C nodes can also be moved from C to B in a similar way. The complexity of this algorithm is $O(|E| \log |E|)$ as each edges is only examined a constant number of times when reducing each separator set.

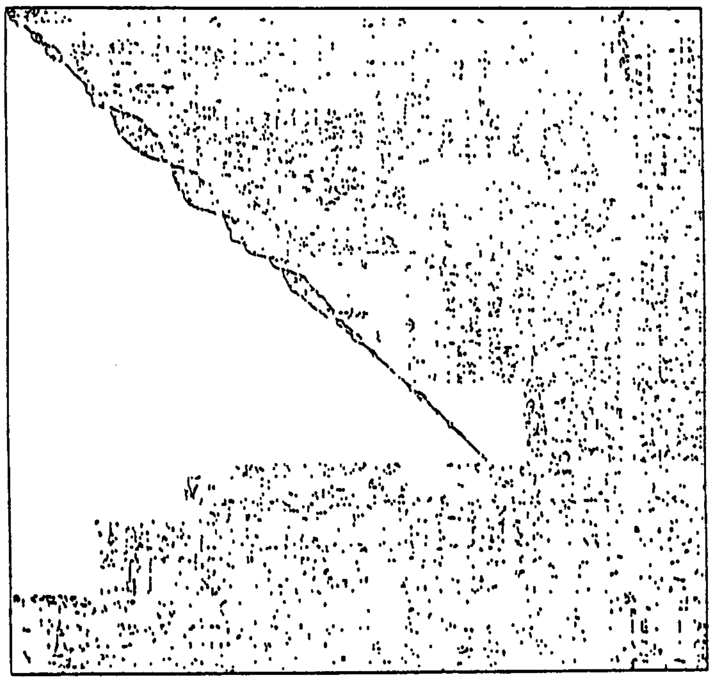


Figure 11: west2021 reordered by **H2**.

In figure 11 the effect of this **H2** ordering is shown on the same matrix as for figure 3. The threshold used for reducing the B sets was $|B|/(\text{initial size of } A)$ is greater than 0.04 and the desired size of B and C was chosen to be $N/10$. As can be seen from this figure the implied border structure for this **H2** ordering can be reasonably large. In the hybrid ordering as described in the next section this effect is reduced. Note that the edges from each S into B or C will represent non-zero elements in the border of the resulting matrix. So, the size of these non-zero entries can be reduced by doing step **2** only for nodes which have "large" edges pointing into B or C first and step **3** for those nodes which are neighbors of these nodes. Secondly step **2** and **3** are done for the remaining nodes.

Because of the recursive nature of this ordering (the algorithm is repeated on each set B and C repeatedly), this ordering is very suited to reduce the diagonal block sizes to their desired size. One could think of running **H1** first on each set B and C before applying this algorithm. However, once **H1** has been run on the matrix, subsequent runs only marginally improve the reduction of the sizes of the diagonal blocks.

4 The Hybrid Ordering

Both the **H1** and the **H2** ordering as described in the previous sections are symmetric orderings (heuristics). This allows us to precede these orderings by an initial ordering **H0** which is unsymmetric and tries to push the largest possible elements of the matrix onto the main diagonal. Note that these elements will stay on the main diagonal during either **H1** or **H2**. This **H0** ordering is obtained by extending the depth-first search algorithm for finding a transversal of a sparse matrix [DER86] to an algorithm which takes into account the numerical values of the entries of the matrix. The transversal algorithm chooses at each step the first row which will produce a non-zero on the main diagonal. If it does not succeed in finding such a row it backtracks on a previous choice of a row. The **H0** differs from this algorithm in the sense that it will consider only rows which will produce a non-zero entry a_{ij} on the main diagonal with

$$|a_{ij}| \geq \text{Tdisp}.\max_k |a_{ik}|.$$

Depending on the choice of Tdisp this algorithm will succeed in finding a transversal or not. As the optimal choice of Tdisp can differ a lot for different matrices, the algorithm is implemented for different choices of Tdisp. First **H0** tries to find a transversal for Tdisp $= 10^{-2}$. If it does not succeed in finding a transversal Tdisp will be taken to be 10^{-4}. After this Tdisp $= 10^{-8}$, and if **H0** does not succeed in this case then the original transversal algorithm is run. Note that the worst case complexity for **H0** is four times the worst case complexity of the original algorithm. However, as the expected time for the original algorithm is much less than the worst case time, **H0** can take considerably more time than the original algorithm. The choice of the parameter Tdisp can be done by the user, and depending on his knowledge of the structure of the matrix the time for searching a transversal can be reduced significantly.

The hybrid ordering (heuristic) **H*** is defined to be:

1. Apply **H0** to A yielding A^1.
2. Apply Tarjan's algorithm to A^2 yielding A^2.
3. Apply **H1** to each diagonal block of A^2 which is larger than some threshold Tblock. This results in a matrix A^3.
4. Apply **H2** to each diagonal block of A^3 larger than Tblock yielding A^4.

The reason for applying Tarjan's algorithm at the second step of the hybrid ordering is that, if A^1 is reducible, then **H1** is invoked on smaller matrices reducing the size of the overall border. As can be concluded from the previous section the (dis)advantages of both **H1** and **H2** are used in this hybrid ordering. First, as **H1** allows a better manipulation of the size of the border, this ordering is invoked first. Second, the disadvantage of the **H2** ordering of introducing relative large borders is minimized by the fact that this ordering is only invoked for diagonal blocks which are already reduced in size. Further the **H2** is more flexible in reducing the size of the diagonal blocks, which can be exploited to bring the matrix in a reasonably balanced form. In figure 12 the effect of **H*** is shown for the same matrix as in figure 3. Tblock is chosen to be $N/10$. It should be mentioned that the implementation of this ordering does not require that the intermediate matrices A^i are explicitly constructed at each stage. In fact only the permutation array has to be constructed for each intermediate stage and the actual ordering can be performed after the **H2** ordering.

For studying the effectiveness of the ordering **H*** more rigorously we have taken a subset of the Harwell/Boeing collection of sparse matrices which are structurally unsymmetric. The ordering was ran on each of these matrices with different parameter settings. The parameter settings as presented in this paper for the different phases of the ordering have proven to be the most successful. A description of these experiments can be found in [GMW91]. The results of **H*** on this subset of Harwell/Boeing matrices is shown in table 1. The matrices bp_xxx, mahistlh, and shl_400 arise from linear programming, fs_680 is a facsimile convergence matrix, the matrix gre_1107 arises from Markov chain modeling, the matrix impcol_d is from a nitric acid plant model, and the westxxxx matrices are from chemical engineering. The diagonal block sizes of the resulting matrices were all

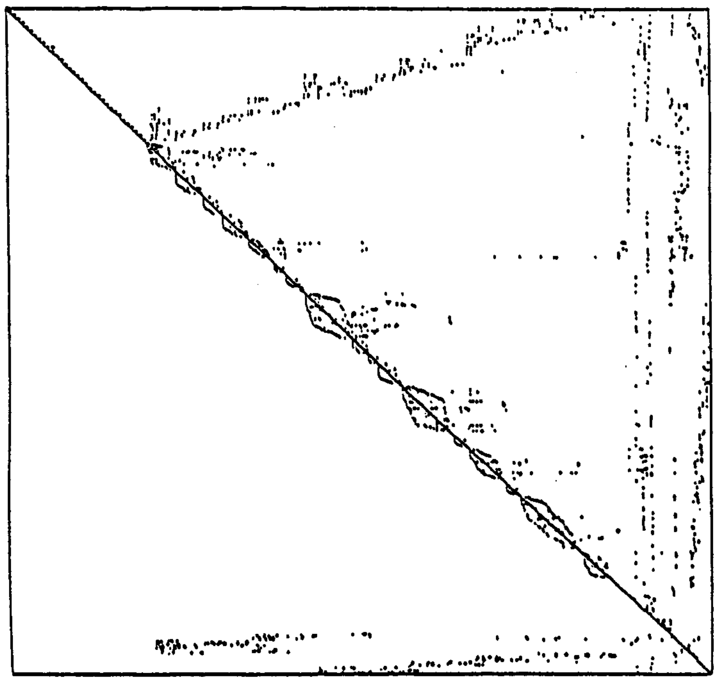


Figure 12: west2021 reordered by **H***.

	Dimension	Border Size
bp_1000	822	16
bp_1600	822	30
bp_800	822	19
fs_680	680	30
gre_1107	1107	327
impcol_d	425	25
mahistlh	1258	140
shl_400	663	0
west1505	1505	107
west2021	2021	118

Table 1: The effectiveness of **H***.

smaller than $N/10$, with N the dimension of the matrix. As can be seen from this table the resulting block sizes are all less than 10% of the dimension of the matrices except for gre_1107 where the border size is almost one third of the dimension. This is caused by the fact that this matrix has the property that the associated digraph consists of a multitude of cycles which are all intertwined with each other making it extremely hard to decompose the graph in separate components.

Concluding we can say that the orderings as presented in this paper are powerful tools to bring a sparse matrix into bordered upper triangular block form, as they are very different in nature but still allow a nice composition so that the various advantages of the components can be exploited. Further, the orderings allow simple manipulations, so that the growth of the entries in the border can be regulated. This is of crucial importance for this ordering to be suited for incorporation into a direct solver. This matter will be studied in full detail in a forth-coming paper [GMW91], together with some more results about the effectiveness of the orderings.

Acknowledgments I would like to thank K. Gallivan and B. Marsolf for their helpful discussions and implementation of the orderings, and Y. Saad for using the matrix display tool from *SparsKit.*

References

[BM76] J.A. Bondy and U.S.R. Murty. *Graph Theory with Applications.* North-Holland, 1976.

[DER86] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

[DGLP82] I.S. Duff, R.G. Grimes, J.G. Lewis, and W.G.Jr. Poole. Sparse matrix test problems. *SIGNUM Newsletter*, 17:22–23, 1982.

[EGL+87] A.M. Erisman, R.G. Grimes, J.G. Lewis, W.G.Jr. Poole, and H.D. Simon. Evaluation of orderings for unsymmetric sparse matrices. *SIAM J. Sci. Stat. Comput.*, 8:600–624, 1987.

[Geo73] A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

[GL81] A. George and J.W. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, 1981.

[GMW91] K. Gallivan, B. Marsolf, and H. Wijshoff. A Large-Grain Parallel System Solver. Technical Report in Preparation, Center for Supercomputing Research & Development, 1991.

[LL87] C.E. Leiserson and J.G. Lewis. Orderings for parallel sparse symmetric factorization. In *Proc. Third SIAM Conf. on Parallel Proc. for Scient. Comp.*, pages 27–31, Los Angeles, CA., 1987.

[LRT79] R.J. Lipton, D.J. Rose, and R.E. Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16:346–358, 1979.

[Tar72] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.