On algorithms for permuting large entries to the diagonal of a sparse matrix[1]

Iain S. Duff[2] and Jacko Koster[3]

ABSTRACT

We consider bipartite matching algorithms for computing permutations of a sparse matrix so that the diagonal of the permuted matrix has entries of large absolute value. We discuss various strategies for this and consider their implementation as computer codes. We also consider scaling techniques to further increase the relative values of the diagonal entries. Numerical experiments show the effect of the reorderings and the scaling on the solution of sparse equations by a direct method and by an iterative technique. The effect on preconditioning for iterative methods is also discussed.

Keywords: sparse matrices, bipartite weighted matching, Dijkstra's algorithm, direct methods, iterative methods, preconditioning.

AMS(MOS) subject classifications: 65F05, 65F50.

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

April 19, 1999.

Contents

1   Introduction

We say that an $n \times n$ matrix A has a large diagonal if the absolute value of each diagonal entry is large relative to the absolute values of the off-diagonal entries in its row and column. Permuting large nonzero entries onto the diagonal of a sparse matrix can be useful in several ways. If we wish to solve the system

$$Ax = b, \tag{1.1}$$

where A is a nonsingular square matrix of order n and x and b are vectors of length n, then a preordering of this kind can be useful whether direct or iterative methods are used for solution (see Olschowka and Neumaier (1996) and Duff and Koster (1997)).

The work in this report is a continuation of the work reported by Duff and Koster (1997) who presented an algorithm that maximizes the smallest entry on the diagonal and relies on repeated applications of the depth first search algorithm MC21 (Duff 1981) in the Harwell Subroutine Library (HSL 1996). In this report, we will be concerned with other bipartite matching algorithms for permuting the rows and columns of the matrix so that the diagonal of the permuted matrix is large. The algorithm that is central to this report computes a matching that corresponds to a permutation of a sparse matrix such that the product (or sum) of the diagonal entries is maximized. This algorithm is already mentioned and used in Duff and Koster (1997), but is not fully described. In this report, we describe the algorithm in more detail. We also consider a modified version of this algorithm to compute a permutation of the matrix that maximizes the smallest diagonal entry. We compare the performance of this algorithm with that of Duff and Koster (1997). We also investigate the influence of scaling of the matrix. Scaling can be used before or after computation of the matching to make the diagonal entries even larger relative to the off-diagonals. In particular, we look at a sparse variant of a bipartite matching and scaling algorithm of Olschowka and Neumaier (1996) that first maximizes the product of the diagonal entries and then scales the matrix so that these entries are one and all other entries are no greater than one.

The rest of this report is organized as follows. In Section 2, we describe some concepts of bipartite matching that we need for the description of the algorithms. In Section 3, we review the basic properties of algorithm MC21. MC21 is a relatively simple algorithm that computes a matching that corresponds to a permutation of the matrix that puts as many entries as possible onto the diagonal without considering their numerical values. The algorithm that maximizes the product of the diagonal entries is described in Section 4. In Section 5, we consider the modified version of this algorithm that maximizes the smallest diagonal entry of the permuted matrix. In Section 6, we consider the scaling of the reordered matrix. Computational experience for the algorithms applied to some practical problems and the effect of the reorderings and scaling on direct and iterative methods of

1

solution are presented in Sections 7 to 7.2. The effect on preconditioning is also discussed. Finally, we consider some of the implications of this current work in Section 8.

2 Bipartite matching

Let $A = (a_{ij})$ be a general $n \times n$ sparse matrix. With matrix A, we associate a bipartite graph $G_A = (V_r, V_c, E)$ that consists of two disjoint node sets V_r and V_c and an edge set E, where $(u, v) \in E$ implies that $u \in V_r$, $v \in V_c$. The sets V_r and V_c have cardinality n and correspond to the rows and columns of A respectively. Edge $(i, j) \in E$ if and only if $a_{ij} \neq 0$. We define the sets $ROW(i) = \{j | (i, j) \in E\}$, for $i \in V_r$, and $COL(j) = \{i | (i, j) \in E\}$, for $j \in V_c$. These sets correspond to the positions of the entries in row i and column j of the sparse matrix respectively. We use $| \ldots |$ both to denote the absolute value and to signify the number of entries in a set, sequence, or matrix. The meaning should always be clear from the context.

A subset $M \subseteq E$ is called a matching (or assignment) if no two edges of M are incident to the same node. A matching containing the largest number of edges possible is called a maximum cardinality matching (or simply maximum matching). A maximum matching is a perfect matching if every node is incident to a matching edge. Obviously, not every bipartite graph allows a perfect matching. However, if the matrix A is nonsingular, then there exists a perfect matching for G_A. A perfect matching M has cardinality n and defines an $n \times n$ permutation matrix $P = (p_{ij})$ with

$$\begin{cases} p_{ji} = 1, & \text{for } (i, j) \in M, \\ p_{ji} = 0, & \text{otherwise}, \end{cases}$$

so that both PA and AP are matrices with the matching entries on the (zero-free) diagonal. Bipartite matching problems can be viewed as a special case of network flow problems (see, for example, Ford Jr. and Fulkerson (1962)).

The more efficient algorithms for finding maximum matchings in bipartite graphs make use of augmenting paths. Let M be a matching in G_A. A node v is matched if it is incident to an edge in M. A path P in G_A is defined as an ordered set of edges in which successive edges are incident to the same node. A path P is called an M-alternating path if the edges of P are alternately in M and not in M. An M-alternating path P is called an M-augmenting path if it connects an unmatched row node with an unmatched column node. In the bipartite graph in Figure 2.1, there exists an M-augmenting path from column node 8 to row node 8. The matching M (of cardinality 7) is represented by the thick edges. The black entries in the accompanying matrix correspond to the matching and the connected matrix entries to the M-augmenting path. If it is clear from the context which matching

M is associated with the M-alternating and M-augmenting paths, then we will simply refer to them as alternating and augmenting paths.

Let M and P be subsets of E. We define

$$M \oplus P := (M \setminus P) \cup (P \setminus M).$$

If M is a matching and P is an M-augmenting path, then $M \oplus P$ is again a matching, and $|M \oplus P| = |M| + 1$. If P is an M-alternating cyclic path, i.e., an alternating path whose first and last edge are incident to the same node, then $M \oplus P$ is also a matching and $|M \oplus P| = |M|$.

Figure 2.1: **Augmenting path**
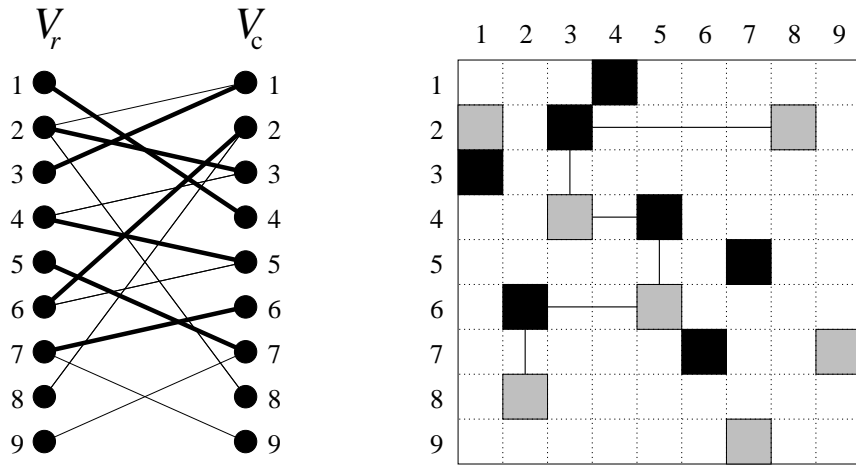
In the sequel, a matching M will often be represented by a pointer array $m : V_r \cup V_c \rightarrow V_r \cup V_c \cup \{\textbf{null}\}$ with

$$\begin{cases} m_i = j \text{ and } m_j = i, & \text{for } (i,j) \in M, \\ m_i = \textbf{null}, & \text{for } i \text{ unmatched.} \end{cases}$$

Augmenting paths in a bipartite graph G can be found by constructing alternating trees. An alternating tree $T = (T_r, T_c, E_T)$ is a subgraph of G rooted at a row or column node and each path in T is an M-alternating path. An alternating tree rooted at a column node j_0 can be grown in the following way. We start with the initial alternating tree $(\emptyset, \{j_0\}, \emptyset)$ and consider all the column nodes $j \in T_c$ in turn. Initially $j = j_0$. For each node j, we check the row nodes $i \in COL(j)$ for which an alternating path from i to j_0 does not yet exist. If node i is already matched, we add row node i, column node

m_i, and edges (i,j) and (i, m_i) to T. If i is not matched, we extend T by row node i and edge (i,j) (and the path in T from node i to the root forms an augmenting path). A key observation for the construction of a maximum or perfect matching is that a matching M is maximum if and only if there is no augmenting path relative to M.

Alternating trees can be implemented using a pointer array $p : V_c \rightarrow V_c$ such that, given an edge $(i,j) \in E_T \setminus M$, node j is either the root node of the tree, or the edges (i,j), (m_j, j), and (m_j, p_j) are consecutive edges in an alternating path towards the root. Augmenting paths in an alternating tree (provided they exist) can thus easily be obtained from p and m.

Alternating trees are not unique. In general, one can construct several alternating trees starting from the same root node that have equal node sets, but different edge sets. Different alternating trees in general will contain different augmenting paths. The matching algorithms that we describe in the next sections impose different criteria on the order in which the paths in the alternating trees are grown in order to obtain augmenting paths and maximum matchings with special properties.

3 Matching

The asymptotically fastest currently known algorithm for finding a maximum matching is by Hopcroft and Karp (1973). It has a worst-case complexity of $\mathcal{O}(\sqrt{n}\tau)$, where $\tau = |E|$ is the number of entries in the sparse matrix. An efficient implementation of this algorithm can be found in Duff and Wiberg (1988). The algorithm MC21 implemented by Duff (1981) has a theoretically worst-case behaviour of $\mathcal{O}(n\tau)$, but in practice it behaves more like $\mathcal{O}(n + \tau)$. Because this latter algorithm is simpler, we concentrate on this in the following although we note that it is relatively straightforward to use the algorithm of Hopcroft and Karp (1973) in a similar way to how we will use MC21 in later sections.

MC21 is a depth-first search algorithm with look-ahead. It starts off with an empty matching M, and hence all column nodes are unmatched initially. See Figure 3.1. For each unmatched column node j_0 in turn, an alternating tree is grown until an augmenting path with respect to the current matching M is found (provided one exists). A set B is used to mark all the matched row nodes that have been visited so far. Initially, $B = \emptyset$. First, the row nodes in $COL(j_0)$ are searched (look-ahead) for an unmatched node i_0. If one is found, the singleton path $P = \{(i_0, j_0)\}$ is an M-augmenting path. If there is no such unmatched node, then an unmarked matched node $i_0 \in COL(j_0)$ is chosen, i_0 is marked, the nodes i_0 and j_1, $j_1 = m_{i_0}$, and the edges (i_0, j_0), (i_0, j_1) are added to the alternating tree (by setting $p_{j_1} = j_0$). The search then continues with column node j_1. For node j_1, the row nodes in $COL(j_1)$ are first checked for an unmatched node. If one exists, say i_1, then the path $P = \{(i_0, j_0), (i_0, j_1), (i_1, j_1)\}$ forms an augmenting

path. If there is no such unmatched node, a remaining unmarked node i_1 is picked from $COL(j_1)$, i_1 is marked, p_{j_2} is set to j_1, $j_2 = m_{i_1}$, and the search moves to node j_2. This continues in a similar (depth-first search) fashion until either an augmenting path $P = \{(i_0, j_0), (i_0, j_1), (i_1, j_1), \ldots, (i_k, j_k)\}$ is found (with nodes j_0 and i_k unmatched) or until for some $k > 0$, $COL(j_k)$ does not contain an unmarked node. In the latter case, MC21 backtracks by resuming the search at the previously visited column node j_{k-1} for some remaining unmarked node $i'_{k-1} \in COL(j_{k-1})$. Backtracking for $k = 0$ is not possible; if MC21 resumes the search at column node j_0 and $COL(j_0)$ does not contain an unmarked node, then an M-augmenting path starting at node j_0 does not exist. In this case, MC21 continues with the construction of a new alternating tree starting at the next unmatched column node. (The final maximum matching will have cardinality at most $n-1$ and hence will not be perfect.)

Figure 3.1: **Outline of MC21.**

```
for j₀ ∈ V_c do
    j := j₀;  p_j := null;  iap := null;
    B := ∅;
    repeat
        if there exists i ∈ COL(j) and i is unmatched then
            iap := i;
        else
            if there exists i ∈ COL(j) \ B then
                B := B + {i};
                p_{m_i} := j;
                j := m_i;
            else
                j := p_j;
            end if;
        end if;
    until iap ≠ null or j = null;
    if iap ≠ null then augment along path from node iap to node j₀;
end for
```

4   Weighted matching

In this section, we describe an algorithm that computes a matching for permuting a sparse matrix A such that the product of the diagonal entries of the permuted matrix is maximum in absolute value. That is, the algorithm determines a matching that corresponds to a permutation σ that maximizes

$$\prod_{i=1}^{n} |a_{i\sigma_i}|. \tag{4.1}$$

This maximization multiplicative problem can be translated into a minimization additive problem by defining matrix $C = (c_{ij})$ as

$$c_{ij} = \begin{cases} \log a_j - \log |a_{ij}|, & a_{ij} \neq 0, \\ \infty, & \text{otherwise,} \end{cases}$$

where $a_j = \max_i |a_{ij}|$ is the maximum absolute value in column j of matrix A. Maximizing (4.1) is equal to minimizing

$$\log \frac{\prod_{i=1}^{n} a_i}{\prod_{i=1}^{n} |a_{i\sigma_i}|} = \log \frac{\prod_{i=1}^{n} a_{\sigma_i}}{\prod_{i=1}^{n} |a_{i\sigma_i}|} = \sum_{i=1}^{n} \log a_{\sigma_i} - \sum_{i=1}^{n} \log |a_{i\sigma_i}| = \sum_{i=1}^{n} (\log a_{\sigma_i} - \log |a_{i\sigma_i}|) =$$

$$\sum_{i=1}^{n} c_{i\sigma_i}. \tag{4.2}$$

Minimizing (4.2) is equivalent to finding a minimum weight perfect matching in an edge weighted bipartite graph. This is known in literature as the bipartite weighted matching problem or (linear sum) assignment problem in linear programming and combinatorial optimization. Numerous algorithms have been proposed for computing minimum weight perfect matchings, see for example Burkard and Derigs (1980), Carpaneto and Toth (1980), Carraresi and Sodini (1986), Derigs and Metz (1986), Jonker and Volgenant (1987), and Kuhn (1955). A practical example of an assignment problem is the allotment of tasks to people; entry c_{ij} in the cost matrix C represents the cost or benefit of assigning person i to task j.

Let $C = (c_{ij})$ be a real-valued $n \times n$ matrix, $c_{ij} \geq 0$. Let $G_C = (V_r, V_c, E)$ be the corresponding bipartite graph each of whose edges $(i, j) \in E$ has weight c_{ij}. The weight of a matching M in G_C, denoted by $c(M)$, is defined by the sum of its edge weights, i.e.,

$$c(M) = \sum_{(i,j) \in M} c_{ij}.$$

A perfect matching M is said to be a minimum weight perfect matching if it has smallest possible weight, i.e., $c(M) \leq c(M')$, for all possible maximum matchings M'.

6

The key concept for finding a minimum weight perfect matching is the so-called shortest augmenting path. An M-augmenting path P starting at an unmatched column node j is called shortest if $c(M \oplus P) \leq c(M \oplus P')$, for all other possible M-augmenting paths P' starting at node j. We define

$$l(P) := c(M \oplus P) - c(M) = c(P \setminus M) - c(M \cap P)$$

as the length of alternating path P. A matching M is called extreme if and only if it does not allow any alternating cyclic path with negative length.

The following two relations hold. First, a perfect matching has minimum weight if it is extreme. Second, if matching M is extreme and P is a shortest M-augmenting path, then $M \oplus P$ is extreme also. The proof for this goes roughly as follows. Suppose $M \oplus P$ is not extreme. Then there exists an alternating cyclic path Q such that $c((M \oplus P) \oplus Q) < c(M \oplus P)$. Since $(M \oplus P) \oplus Q = M \oplus (P \oplus Q)$ and M is extreme, there must exist a subset $P' \subseteq P \oplus Q$ that forms an M-augmenting path and is shorter than P. Hence, P is not a shortest M-augmenting path. This contradicts the supposition.

These two relations form the basis for many algorithms for solving the bipartite weighted matching problem: start from any (possibly empty) extreme matching M and successively augment M along shortest augmenting paths until M is maximum (or perfect).

In the literature, the problem of finding a minimum weight perfect matching is often stated as the following linear programming problem. Find matrix $X = (x_{ij}) \in R^{n \times n}$, minimizing

$$\sum_{(i,j) \in E} c_{ij} x_{ij}$$

subject to

$$\sum_{j \in V_c} x_{ij} = 1, \quad \text{for } i \in V_r,$$

$$\sum_{i \in V_r} x_{ij} = 1, \quad \text{for } j \in V_c,$$

$$x_{ij} \geq 0, \quad \text{for } (i,j) \in E,$$

$$x_{ij} = 0, \quad \text{for } (i,j) \notin E.$$

If there is a solution to this linear program, there is one for which $x_{ij} \in \{0, 1\}$ and there exists a permutation matrix X such that $M = \{(i,j) | x_{ij} = 1\}$ is a minimum weight perfect matching (Edmonds and Karp 1972, Kuhn 1955). Furthermore, M has minimum weight if and only if there exist $dual$ variables u_i and v_j with

$$\begin{cases} u_i + v_j \leq c_{ij}, & \text{for } (i,j) \in E, \\ u_i + v_j = c_{ij}, & \text{for } (i,j) \in M. \end{cases} \tag{4.3}$$

Using the reduced weight matrix $\overline{C} = (\overline{c}_{ij})$, with

$$\overline{c}_{ij} := c_{ij} - u_i - v_j \geq 0,$$

the reduced weight $\overline{c}(M)$ of matching M equals

$$\overline{c}(M) = 0,$$

the reduced length $\overline{l}(P)$ of any M-alternating path P equals

$$\overline{l}(P) = \sum_{(i,j) \in P \setminus M} \overline{c}_{ij} \geq 0,$$

and if $M \oplus P$ is a matching, the reduced weight of $M \oplus P$ equals

$$\overline{c}(M \oplus P) = \overline{l}(P).$$

Thus, finding a shortest augmenting path in graph G_C is equivalent to finding an augmenting path in graph $G_{\overline{C}}$, with minimum reduced length. Since $\overline{c}_{ij} = 0$ for every edge $(i,j) \in M$ and graph $G_{\overline{C}}$ contains no alternating paths P with negative length, $\overline{l}(P') \leq \overline{l}(P)$ for every principal leading subpath P' of P.

Shortest augmenting paths in a weighted bipartite graph $G = (V_r, V_c, E)$ can be obtained by means of a shortest alternating path tree. A shortest alternating path tree T is an alternating tree each of whose paths is a shortest path in G. For any node $i \in V_r \cup V_c$, we define d_i as the length of the shortest path in T from node i to the root node ($d_i = \infty$ if no such path exists). T is a shortest alternating path tree if and only if $d_i + \overline{c}_{ij} \geq d_j$, for every edge $(i,j) \in E$ and tree nodes i, j,

An outline of an algorithm for constructing a shortest alternating path tree rooted at column node j_0 is given in Figure 4.1. Because the reduced weights $\overline{c}_{ij}$ are non-negative, and graph $G_{\overline{C}}$ contains no alternating paths with negative length, we can use a sparse variant of Dijkstra's algorithm (Dijkstra 1959). The set of row nodes is partitioned into three sets B, Q, and W. B is the set of (marked) nodes whose shortest alternating paths and distances to node j_0 are known. Q is the set of nodes for which an alternating path to the root is known that is not necessarily the shortest possible. W is the set of nodes for which an alternating path does not exist or is not known yet. (Note that since W is defined implicitly as $V_r \setminus (B \cup Q)$, it is not actually used in Figure 4.1.) The algorithm starts with $(\emptyset, \{j_0\}, \emptyset)$ as initial shortest alternating tree and extends the tree until an augmenting path is found that is guaranteed to be a shortest augmenting path with respect to the current matching M. Initially, the length of the shortest augmenting path $lsap$ in the tree is set to infinity, and the length of the shortest alternating path lsp from the root to any node in Q is set to zero. On each pass through the main loop, another column node j is chosen that is closest to the root j_0. Initially $j = j_0$.

Each row node $i \in COL(j)$ whose shortest alternating path to the root is not known yet ($i \notin B$), is considered. If $P_{j_0 \to j \to i}$, the shortest alternating path from the root node j_0 to node j (with length lsp) extended by edge (i,j) from node j to node i (with length $\overline{c}_{ij}$), is longer than the tentative shortest augmenting path in the tree (with length $lsap$), then there is no need to modify the tree. If $P_{j_0 \to j \to i}$ has length smaller than $lsap$, and i is unmatched, then a new shorter augmenting path has been found and $lsap$ is updated. If i is matched and $P_{j_0 \to j \to i}$ is also shorter than the current shortest alternating path to i (with length d_i), then a shorter alternating path to node i has been found and the tree is updated, d_i is updated, and if node i has not been visited previously, i is moved to Q.

Next, if Q is not empty, a node $i \in Q$ is determined that is closest to the root. Since all weights $\overline{c}_{ij}$ in the bipartite graph are non-negative, there cannot be any other alternating path to node i that is shorter than the current one. Node i is marked (by adding it to B), and the search continues with column node $j' = m_i$. This continues until there are no more column nodes to be searched ($Q = \emptyset$), or until no new augmenting path can be found whose length is smaller than the current shortest one (line $lsap \leq lsp$).

The original Dijkstra algorithm (intended for dense graphs) has $\mathcal{O}(n^2)$ complexity. For sparse problems, the complexity can be reduced to $\mathcal{O}(\tau \log n)$ by implementing the set Q as a k-heap in which the nodes i are sorted by increasing distance d_i from the root (see for example Tarjan (1983) and Gallo and Pallottino (1988)). The running time of the algorithm is dominated by the operations on the heap Q of which there are $\mathcal{O}(n)$ delete operations, $\mathcal{O}(n)$ insert operations, and $\mathcal{O}(\tau)$ modification operations (these are necessary each time a distance d_i is updated). Each insert and modification operation runs in $\mathcal{O}(\log_k n)$ time, a delete operation runs in $\mathcal{O}(k \log_k n)$ time. Consequently, the algorithm for finding a shortest augmenting path in a sparse bipartite graph has run time $\mathcal{O}((\tau + kn) \log_k n)$ and the total run time for the sparse bipartite weighted algorithm is $\mathcal{O}(n(\tau + kn) \log_k n)$. If we choose $k = 2$, the algorithm uses binary heaps and we obtain a time bound of $\mathcal{O}(n(\tau + n) \log_2 n)$. If we choose $k = \lceil \tau/n \rceil$ (and $k \geq 2$), we obtain a bound of $\mathcal{O}(n\tau \log_{\tau/n} n)$.

The implementation of the heap Q is similar to the implementation proposed in Derigs and Metz (1986). Q is a pair (Q_1, Q_2) where Q_1 is an array that contains all the row nodes for which the distance to the root is shortest (lsp), and $Q_2 = Q \setminus Q_1$ is a 2-heap. By separating the nodes in Q that are closest to the root, we may reduce the number of operations on the heap, especially in those situations where the cost matrix C has only few different numerical values and many alternating paths have the same length. Deleting a node from Q for which d_i is smallest (see Figure 4.1), now consists of choosing an (arbitrary) element from Q_1. If Q_1 is empty, then we first move all the nodes in Q_2 that are closest to the root to Q_1.

After the augmentation, the reduced weights $\overline{c}_{ij}$ have to be updated to ensure that alternating paths in the new $\overline{G}$ have non-negative length. This is done by modifying the

Figure 4.1: **Construction of a shortest augmenting path.**

$B := \emptyset; \quad Q := \emptyset;$
for $i \in V_r$ **do** $d_i := \infty;$
$lsp := 0; \quad lsap := \infty;$
$j := j_0; \quad p_j := \mathbf{null};$
while $true$ **do**
  for $i \in COL(j) \setminus B$ **do**
    $dnew := lsp + \overline{c}_{ij};$
    if $dnew < lsap$ **then**
      if i unmatched **then**
        $lsap := dnew; \quad isap := i;$
      else
        if $dnew < d_i$ **then**
          $d_i := dnew; \quad p_{m_i} := j;$
          if $i \notin Q$ **then** $Q := Q + \{i\};$
        end if;
      end if;
    end if;
  end for;
  if $Q = \emptyset$ **then** exit while-loop;
  choose $i \in Q$ with minimal d_i;
  $lsp := d_i;$
  if $lsap \le lsp$ **then** exit while-loop;
  $Q := Q - \{i\}; \quad B := B + \{i\};$
  $j := m_i;$
end while;
if $lsap \ne \infty$ **then** augment along path from node $isap$ to node j_0;

dual vectors u and v. If $T = (T_r, T_c, E_T)$ is the shortest alternating path tree that was constructed until the shortest augmenting path was found, then u_i and v_j are updated as follows:

$$\begin{cases} u_i := u_i + d_i - lsap, & \text{for } i \in T_r, \\ v_j := c_{ij} - u_i, & \text{for } j \in T_c. \end{cases}$$

The updated dual variables u and v satisfy (4.3) and the new reduced weights $\overline{c}_{ij}$ are non-negative.

The running time of the weighted matching algorithm can be decreased considerably by means of a cheap heuristic that determines a large initial extreme matching M. We use the strategy proposed by Carpaneto and Toth (1980). We calculate

$$u_i := \min_{j \in ROW(i)} c_{ij}, \quad \text{for } i \in V_r,$$

$$v_j := \min_{i \in COL(j)} (c_{ij} - u_i), \quad \text{for } j \in V_c.$$

Inspecting the sets $COL(j)$ for each column node j in turn, we determine a large initial matching M of edges for which $c_{ij} - u_i - v_j = 0$. Then, for each remaining unmatched column node j, every node $i \in COL(j)$ is considered for which $c_{ij} - u_i - v_j = 0$ and that is matched to a column node other than j, say j_1. So $(i, j_1) \in M$. If an unmatched row node $i_1 \in COL(j_1)$ can be found for which $c_{i_1 j_1} - u_{i_1} - v_{j_1} = 0$, then (i, j_1) in M is replaced by (i, j) and (i_1, j_1). After having repeated this for all unmatched columns, the search for shortest augmenting paths starts with respect to the current matching.

Finally, we note that the above weighted matching algorithm can also be used for maximizing the sum of the diagonal entries of matrix A (instead of maximizing the product of the diagonal entries). To do this, we again minimize (4.2), but we redefine matrix C as

$$c_{ij} = \begin{cases} a_j - |a_{ij}|, & a_{ij} \neq 0, \\ 0, & \text{otherwise.} \end{cases}$$

Maximizing the sum of the diagonal entries is equal to minimizing (4.2), since

$$\sum_{i=1}^{n} a_{\sigma_i} - \sum_{i=1}^{n} |a_{i\sigma_i}| = \sum_{i=1}^{n} (a_{\sigma_i} - |a_{i\sigma_i}|) = \sum_{i=1}^{n} c_{i\sigma_i}.$$

5   Bottleneck matching

We describe a modification of the weighted bipartite matching algorithm from the previous section for permuting rows and columns of a sparse matrix A such that the smallest ratio

between the absolute value of a diagonal entry and the maximum absolute value in its column is maximized. That is, the modification computes a permutation σ that maximizes

$$\min_{1 \leq i \leq n} \frac{|a_{i\sigma_i}|}{a_{\sigma_i}} \tag{5.1}$$

where a_j is the maximum absolute value in column j of the matrix A. Similarly to the previous section, we transform this into a minimization problem. We define the matrix $C = (c_{ij})$ as

$$c_{ij} = \begin{cases} 1 - \dfrac{|a_{ij}|}{a_j}, & a_{ij} \neq 0, \\ \infty, & \text{otherwise.} \end{cases}$$

Then maximizing (5.1) is equal to minimizing

$$\max_{1 \leq i \leq n} \frac{a_{\sigma_i} - |a_{i\sigma_i}|}{a_{\sigma_i}} = \max_{1 \leq i \leq n} c_{i\sigma_i}.$$

Given a matching M in the bipartite graph $G_C = (V_r, V_c, E)$, the bottleneck value of M is defined as

$$c(M) = \max_{(i,j) \in M} c_{ij}.$$

The problem is to find a perfect (or maximum) bottleneck matching M for which $c(M)$ is minimal, i.e. $c(M) \leq c(M')$, for all possible maximum matchings M'. A matching M is called extreme if it does not allow any alternating cyclic path P for which $c(M \oplus P) < c(M)$.

The bottleneck algorithm starts off with any extreme matching M. The initial bottleneck value b is set to $c(M)$. Each pass through the main loop, an alternating tree is constructed until an augmenting path P is found for which either $c(M \oplus P) = c(M)$ or $c(M \oplus P) - c(M) > 0$ is as small as possible. The initializations and the main loop for constructing such an augmenting path are those of Figure 4.1. Figure 5.1 shows the inner-loop of the weighted matching algorithm of Figure 4.1 modified to the case of the bottleneck objective function. The main differences are that the sum operation on the path lengths in Figure 4.1 is replaced by the "max" operation and, as soon as an augmenting path P is found whose length $lsap$ is less than or equal to the current bottleneck value b, the main loop is exited, P is used to augment M, and b is set to $\max(b, lsap)$. The bottleneck algorithm does not modify the edge weights c_{ij}.

Similarly to the implementation discussed in Section 4, the set Q is implemented as a pair (Q_1, Q_2), but now the array Q_1 contains all the nodes whose distance to the root is less than or equal to the tentative bottleneck value b. Q_2 contains the nodes whose distance to the root is larger than the bottleneck value but not infinity. Q_2 is again implemented as a heap.

12

Figure 5.1: **Modified inner loop of Figure 4.1 for the construction of a bottleneck augmenting path.**

```
for i ∈ COL(j) \ B do
    dnew := max(lsp, c_ij);
    if dnew < lsap then
        if i unmatched then
            lsap := dnew;  isap := i;
            if lsap ≤ b then exit while-loop;
        else
            if dnew < d_i then
                d_i := dnew;  p_{m_i} := j;
                if i ∉ Q then Q := Q + {i};
            end if;
        end if;
    end if;
end for;
```

A large initial extreme matching can be found in the following way. We define

$$r_i := \min_{j \in ROW(i)} c_{ij}, \quad \text{for } i \in V_r,$$

$$s_j := \min_{i \in COL(j)} c_{ij}, \quad \text{for } j \in V_c,$$

as the smallest entry in row i and column j, respectively. A lower bound b_0 for the bottleneck value is

$$b_0 := \max\{\max_i r_i, \max_j s_j\}.$$

An extreme matching M can be obtained from the edges (i, j) for which $c_{ij} \le b_0$; we scan all nodes $j \in V_c$ in turn and for each node $i \in COL(j)$ that is unmatched and for which $c_{ij} \le b_0$, edge (i, j) is added to M. Then, for each remaining unmatched column node j, every node $i \in COL(j)$ is considered for which $c_{ij} \le b$ and that is matched to a column node other than j, say j_1. So $(i, j_1) \in M$. If an unmatched row node $i_1 \in COL(j_1)$ can be found for which $c_{i_1 j_1} \le b$, then (i, j_1) in M is replaced by (i, j) and (i_1, j_1). After having done this for all unmatched columns, the search for shortest augmenting paths starts with respect to the current matching.

Other initialization procedures can be found in the literature. For example, a slightly more complicated initialization strategy is used by Finke and Smith (1978) in the context

13

of solving transportation problems. For every $i \in V_r$, $j \in V_c$, they use

$$\begin{aligned}
g_i &:= |\{c_{ik}|k \in ROW(i) \text{ and } c_{ik} \le b_0\}|, \\
h_j &:= |\{c_{kj}|k \in COL(j) \text{ and } c_{kj} \le b_0\}|,
\end{aligned}$$

as the number of admissible edges incident to row node i and column node j respectively. The idea behind using g_i and h_j is that once an admissible edge (i,j) is added to M, all the other admissible edges that are incident to nodes i and j are no longer candidates to be added to M. Therefore, the method tries to pick admissible edges such that the number of admissible edges that become unusable is minimal. First, a row node i with minimal g_i is determined. From the set $ROW(i)$ an admissible entry (i,j) (provided one exists) is chosen for which h_j is minimal and (i,j) is added to M. After deleting the edges (i,k), $k \in ROW(i)$, and the edges (k,j), $k \in COL(j)$, the method repeats the same for another row node i' with minimal $g_{i'}$. This continues until all admissible edges are deleted from the graph.

Finally, we note that instead of maximizing (5.1) we also could have maximized the smallest absolute value on the diagonal. That is, we maximize

$$\min_{1 \le i \le n} |a_{i\sigma_i}|,$$

and define the matrix C as

$$c_{ij} = \begin{cases} a_j - |a_{ij}|, & a_{ij} \ne 0, \\ \infty, & \text{otherwise.} \end{cases}$$

Note that this problem is rather sensitive to the scaling of the matrix A. Suppose for example that the matrix A has a column containing only one nonzero entry whose absolute value v is the smallest absolute value present in A. Then, after applying the bottleneck algorithm, the bottleneck value b will be equal to this small value. The smallest entry on the diagonal of the permuted matrix is maximized, but the algorithm did not have any influence on the values of the other diagonal values. Scaling the matrix prior to applying the bottleneck algorithm avoids this.

In Duff and Koster (1997), a different approach is taken to obtain a bottleneck matching. Let A_ϵ denote the matrix that is obtained by setting to zero in A all entries a_{ij} for which $|a_{ij}| < \epsilon$ (thus $A_0 = A$) and M_ϵ denote the matching obtained by removing from matching M all the entries (i,j) for which $|a_{ij}| < \epsilon$ (thus $M_0 = M$). Throughout the algorithm, ϵmax and ϵmin are such that a maximum matching of size $|M|$ does not exist for $A_{\epsilon max}$ but does exist for $A_{\epsilon min}$. At each step, ϵ is chosen in the interval $(\epsilon min, \epsilon max)$, and a maximum matching for the matrix A_ϵ is computed using a variant of MC21. If this matching has size $|M|$, then ϵmin is set to ϵ, otherwise ϵmax is set to ϵ. Hence,

the size of the interval decreases at each step and ϵ will converge to the bottleneck value. After termination of the algorithm, M' is the computed bottleneck matching and ϵ the corresponding bottleneck value.

6 Scaling

Olschowka and Neumaier (1996) use the dual solution produced by the weighted matching algorithm to scale the matrix. Let u and v be such that they satisfy relation (4.3). If we define

$$\begin{aligned} D_1 &= diag(d_1^1, d_2^1, \ldots, d_n^1), \quad d_i^1 = \exp(u_i), \\ D_2 &= diag(d_1^2, d_2^2, \ldots, d_n^2), \quad d_j^2 = \exp(v_j)/a_j, \end{aligned} \tag{6.1}$$

then we have

$$d_i^1 |a_{ij}| d_j^2 =$$

$$\exp(u_i + \log(|a_{ij}|) + v_j - \log(a_j)) =$$

$$\exp(u_i + v_j - (\log(a_j) - \log(|a_{ij}|))) =$$

$$\exp(u_i + v_j - c_{ij}) \le 1$$

Equality holds when $u_i + v_j = c_{ij}$, that is $(i,j) \in M$. In words, $D_1 A D_2$ is a matrix whose diagonal entries are one in absolute value and whose off-diagonal entries are all less than or equal to one. Olschowka and Neumaier (1996) call such a matrix an I-matrix and use this in the context of dense Gaussian elimination to reduce the amount of pivoting that is needed for numerical stability. The more dominant the diagonal of a matrix, the higher the chance that diagonal entries are stable enough to serve as pivots for elimination.

For iterative methods, the transformation of a matrix to an I-matrix is also of interest. For example, from Gershgorin's theorem we know that the union of all discs

$$K_i = \left\{ \mu \in C \mid |\mu - a_{ii}| \le \sum_{k \ne i} |a_{ik}| \right\}$$

contains all eigenvalues of the $n \times n$ matrix A. Disc K_i has center at a_{ii} and radius that is equal to the sum of the absolute off-diagonal values in row i. Since the diagonal entries of an I-matrix are all one, all the n disks have center at 1. The estimate of the eigenvalues will be sharper as A deviates less from a diagonal matrix. That is, the smaller the radii of the discs, the better we know where the eigenvalues are situated. If we are able to reduce the radii of the discs of an I-matrix, i.e. reduce the off-diagonal values, then we tend to cluster the eigenvalues more around one. In the ideal case, all the discs of an I-matrix have a radius smaller than one, in which case the matrix is strictly row-wise diagonally

15

dominant. This guarantees that many types of iterative methods will converge (in exact arithmetic), even simple ones like the Jacobi and Gauss-Seidel method. However, if at least one disc remains with radius larger than or close to one, zero eigenvalues or small eigenvalues are possible.

A straightforward (but expensive) attempt to decrease large off-diagonal entries of a matrix is by row and column equalization (Olschowka and Neumaier 1996). Let A be an I-matrix. We define matrix $C = (c_{ij})$ as $c_{ij} = \log |a_{ij}|$. (For simplicity we assume that A contains no zero entries.) Equalization consists of repeatedly equalizing the largest absolute value in row i and the largest absolute values in column i:

$$p := 0;$$
$$\textbf{for } k := 1, 2, \ldots \textbf{ do}$$
$$\qquad \textbf{for } j := 1 \textbf{ to } n \textbf{ do}$$
$$\qquad\qquad y_1 := \max\{c_{jr} + p_j - p_r | r \neq j, c_{jr} \neq 0\};$$
$$\qquad\qquad y_2 := \max\{c_{rj} + p_r - p_j | r \neq j, c_{rj} \neq 0\};$$
$$\qquad\qquad p_j := p_j + (y_2 - y_1)/2;$$
$$\qquad \textbf{end};$$
$$\textbf{end};$$

For $k = \infty$, this algorithm minimizes

$$\max\{c_{ij} + p_i - p_j | i \neq j, c_{ij} \neq 0\}$$

and thus, if we define $d_i^1 := \exp(p_i)$ and $d_j^2 := 1/\exp(p_j)$, the algorithm minimizes the largest off-diagonal absolute value in matrix $D_1 A D_2$. The diagonal entries do not change.

Note that the above scaling strategy does not guarantee that all off-diagonal entries of an I-matrix will be smaller than one in absolute value, for example if the I-matrix A contains two off-diagonal entries a_{kl} and a_{lk}, $k \neq l$, whose absolute values are both one.

7 Experimental results

In this section, we discuss several cases where the reorderings algorithms from the previous section can be useful. These include the solution of sparse equations by a direct method and by an iterative technique. We also consider its use in generating a preconditioner for an iterative method.

The set of matrices that we used for our experiments are unsymmetric matrices taken from the Harwell-Boeing Sparse Matrix Test Collection (Duff, Grimes and Lewis 1992) and from the sparse matrix collection at the University of Florida (Davis 1997).

All matrices are initially row and column scaled. By this we mean that the matrix is scaled so that the maximum entry in each row and in each column is one.

The computer used for the experiments is a SUN UltraSparc with 256 Mbytes of main memory. The algorithms are implemented in Fortran 77.

We use the following acronyms. MC21 is the matching algorithm from the Harwell Subroutine Library for computing a matching such that the corresponding permuted matrix has a zero free-diagonal (see Section 3). BT is the bottleneck bipartite matching algorithm from Section 5 for permuting a matrix such that the smallest ratio between the absolute value of a diagonal entry and the maximum absolute value in its column is maximized. BT' is the bottleneck bipartite matching algorithm from Duff and Koster (1997). MPD is the weighted matching algorithm from Section 4 and computes a permutation such that the product of the diagonal entries of the permuted matrix is maximum in absolute value. MPS is equal to the MPD algorithm, but after the permutation, the matrix is scaled to an I-matrix (see Section 6).

Table 7.1 shows for some large sparse matrices the order, number of entries, and the time for the algorithms to compute a matching. The times for MPS are not listed, because they are almost identical to those for MPD. In general, MC21 needs the least time to compute a matching, except for the ONETONE and TWOTONE matrices. For these matrices, the search heuristic that is used in MC21 (a depth-first search with look-ahead) does not perform well. This is probably caused by the ordering of the columns (variables) and the entries inside the columns of the matrix. A random permutation of the matrix prior to applying MC21 might lead to other results. There is not a clear winner between the bottleneck algorithms BT and BT', although we note that BT' requires the entries inside the columns to be sorted by value. This sorting can be expensive for relatively dense matrices. MPD is in general the most expensive algorithm. This can be explained by the more selective way in which this algorithm constructs augmenting paths.

7.1 Experiments with a direct solution method

For direct methods, putting large entries on the diagonal suggests that pivoting down the diagonal might be more stable. Indeed, stability can still not be guaranteed, but if we have a solution scheme like the multifrontal method of Duff and Reid (1983), where a symbolic phase chooses the initial pivotal sequence and the subsequent factorization phase then modifies this sequence for stability, it can mean that the modification required is less than if the permutation were not applied.

In the multifrontal approach of Duff and Reid (1983), later developed by Amestoy and Duff (1989), an analysis is performed on the structure of $A + A^T$ to obtain an ordering that reduces fill-in under the assumption that all diagonal entries will be numerically suitable for pivoting. The numerical factorization is guided by an assembly tree. At each node of the tree, some steps of Gaussian elimination are performed on a dense submatrix whose Schur complement is then passed to the parent node in the tree where it is assembled

Table 7.1: **Times (in seconds) for matching algorithms.** Order of matrix is n and number of entries τ.

Matrix	n	τ	MC21	BT'	BT	MPD
MAHINDAS	1258	7682	0.01	0.01	0.01	0.02
GEMAT11	4929	33185	0.01	0.03	0.01	0.04
ONETONE1	36057	341088	2.67	0.70	0.18	0.61
ONETONE2	36057	227628	2.63	0.53	0.14	0.42
TWOTONE	120750	1224224	60.10	6.95	2.82	2.17
GOODWIN	7320	324784	0.27	2.26	4.17	1.82
LHR01	1477	18592	0.02	0.04	0.10	0.10
LHR02	2954	37206	0.04	0.14	0.16	0.21
LHR07	7337	156508	0.04	0.58	0.24	0.82
LHR14C	14270	307858	0.28	1.13	1.12	3.32
LHR71C	70304	1528092	1.86	9.00	11.96	37.73
AV41092	41092	1683902	35.72	10.81	37.82	65.13

(or summed) with Schur complements from the other children and original entries of the matrix. If, however, numerical considerations prevent us from choosing a pivot then the algorithm can proceed, but now the Schur complement that is passed to the parent is larger and usually more work and storage will be needed to effect the factorization.

The logic of first permuting the matrix so that there are large entries on the diagonal, before computing the ordering to reduce fill-in, is to try and reduce the number of pivots that are delayed in this way thereby reducing storage and work for the factorization. We show the effect of this in Table 7.2 where we can see that even using MC21 can be very beneficial although the other algorithms can show significant further gains.

In Table 7.3, we show the effect of this on the number of entries in the factors. Clearly this mirrors the results in Table 7.2.

In addition to being able to select the pivots chosen by the analysis phase, the multifrontal code MA41 will do better on matrices whose structure is symmetric or nearly so. Here, we define the structural symmetry for a matrix A as the number of entries a_{ij} for which a_{ji} is also an entry, divided by the total number of entries. The structural symmetry after the permutations is shown in Table 7.4. The matching orderings in some cases increase the symmetry of the resulting reordered matrix, which is particularly apparent when we have a very sparse system with many zeros on the diagonal. In that case, the reduction in number of off-diagonal entries in the reordered matrix has an influence on the symmetry. Notice that, in this respect, the more sophisticated matching algorithms may actually cause problems since they could reorder a symmetrically structured matrix with a zero-free diagonal, whereas MC21 will leave it unchanged.

Table 7.2: **Number of delayed pivots in factorization from MA41.** An "-" indicates that MA41 needed more than 200 MBytes of memory.

Matrix	Matching algorithm used				
	None	MC21	BT	MPD	MPS
GEMAT11	-	76	0	0	0
ONETONE1	-	16261	298	100	0
ONETONE2	40916	8310	411	100	0
GOODWIN	536	1622	427	53	41
LHR01	1378	171	42	18	0
LHR02	3432	388	143	56	0
LHR14C	-	7608	1042	169	274
LHR71C	-	35354	7424	2643	3190
AV41092	-	10151	2141	1730	1722

Table 7.3: **Number of entries (10^3) in the factors from MA41.**

Matrix	Matching algorithm used				
	None	MC21	BT	MPD	MPS
GEMAT11	-	128	79	78	78
ONETONE1	-	10,359	7,329	4,715	4,713
ONETONE2	14,083	2,876	2,298	2,170	2,168
GOODWIN	1,263	2,673	2,058	1,282	1,281
LHR01	997	137	210	113	111
LHR02	2,299	333	374	235	230
LHR14C	-	3,111	2,676	2,164	2,165
LHR71C	-	18,787	17,528	11,600	11,630
AV41092	-	16,226	14,968	14,110	14,111

Table 7.4: **Structural symmetry after permutation.** (1.00 = symmetric)

Matrix	Matching algorithm used			
	None	MC21	BT	MPD/MPS
GEMAT11	0.002	0.530	0.947	0.957
ONETONE1	0.990	0.368	0.427	0.434
ONETONE2	0.148	0.461	0.564	0.574
GOODWIN	0.642	0.288	0.365	0.583
LHR01	0.009	0.302	0.133	0.168
LHR02	0.009	0.302	0.141	0.168
LHR14C	0.007	0.336	0.125	0.150
LHR71C	0.002	0.384	0.182	0.207
AV41092	0.001	0.101	0.082	0.082

Finally, Table 7.5 shows the effect on the solution times of MA41. We sometimes observe a dramatic reduction in time for the solution when preceded by a permutation.

Table 7.5: **Solution time required by MA41.**

Matrix	Matching algorithm used				
	None	MC21	BT	MPD	MPS
GEMAT11	-	0.28	0.20	0.20	0.20
ONETONE1	-	225.71	95.33	44.22	42.97
ONETONE2	81.45	17.05	11.70	11.54	11.13
GOODWIN	3.64	14.63	7.98	3.56	3.56
LHR01	10.10	0.41	0.72	0.28	0.28
LHR02	24.85	1.07	1.10	0.58	0.55
LHR14C	-	12.66	10.48	5.88	5.83
LHR71C	-	148.07	127.92	43.33	42.90
AV41092	-	226.20	180.39	155.70	154.44

Our implementations of the algorithms described in this paper have been used successfully by Li and Demmel (1998) to stabilize sparse Gaussian elimination in a distributed-memory environment without the need for dynamic pivoting. Their method decomposes the matrix into an $N \times N$ block matrix $A[1 : N, 1 : N]$ by using the notion of unsymmetric supernodes (Demmel, Eisenstat, Gilbert, Li and Liu 1995). The blocks are mapped cyclically (in both row and column dimensions) onto the nodes (processors) of a two-dimensional rectangular processor grid. The mapping is such that at step k of the numerical factorization, a column of processors factorizes the block column $A[k : N, k]$, a row of processes participates in the triangular solves to obtain the block row $U[k, k+1 : N]$, and all processors participate in the corresponding multiple-rank update of the remaining matrix $A[k + 1 : N, k + 1 : N]$.

The numerical factorization phase in this method does not use (dynamic) partial pivoting on the block columns. This allows for the a priori computation of the nonzero structure of the factors, the distributed data structures, the communication pattern, and a good load balancing scheme, which makes the factorization more scalable on distributed-memory machines than factorizations in which the computational and communication tasks only become apparent during the elimination process. To ensure a solution that is numerically stable, the matrix is permuted and scaled before the factorization to make the diagonal entries large compared to the off-diagonal entries, any tiny pivots encountered during the factorization are perturbed, and a few steps of iterative refinement are performed during the triangular solution phase if the solution is not accurate enough. Numerical experiments demonstrate that the method (using the implementation of the MPS algorithm) is as stable as partial pivoting for a wide range of problems.

7.2 Experiments with iterative solution methods

For iterative methods, simple techniques like Jacobi or Gauss-Seidel converge more quickly if the diagonal entry is large relative to the off-diagonals in its row or column, and techniques like block iterative methods can benefit if the entries in the diagonal blocks are large. Additionally, for preconditioning techniques, for example for diagonal preconditioning or incomplete LU preconditioning, it is intuitively evident that large diagonals should be beneficial.

7.2.1 Preconditioning by incomplete factorizations

In incomplete factorization preconditioners, pivots are often taken from the diagonal and fill-in is discarded if it falls outside a prescribed sparsity pattern. (See Saad (1996) for an overview.) Incomplete factorizations are used so that the resulting factors are more economical to store, to compute, and to solve with.

One of the reasons why incomplete factorizations can behave poorly is that pivots can be arbitrarily small (Benzi, Szyld and van Duin 1997, Chow and Saad 1997). Pivots may even be zero in which case the incomplete factorization fails. Small pivots allow the numerical values of the entries in the incomplete factors to become very large, which leads to unstable and therefore inaccurate factorizations. In such cases, the norm of the residual matrix $R = A - \hat{L}\hat{U}$ will be large. (Here, $\hat{L}$ and $\hat{U}$ denote the computed incomplete factors.)

A way to improve the stability of the incomplete factorization, is to preorder the matrix to put large entries onto the diagonal. Obviously, a successful factorization still cannot be guaranteed, because nonzero diagonal entries may become very small (or even zero) *during* the factorization, but the reordering may mean that zero or small pivots are less likely to occur. Table 7.6 shows some results for the reorderings applied prior to incomplete factorizations of the form ILU(0), ILU(1), and ILUT and the iterative methods GMRES(20), BiCGSTAB, and QMR. In some cases, the method will only converge after the permutation, in others it greatly improves the convergence.

However, we emphasize that permuting large entries to the diagonal of matrix will not always improve the accuracy and stability of incomplete factorization. An inaccurate factorization can also occur in the absence of small pivots, when many (especially large) fill-ins are dropped from the incomplete factors. In this respect, it may be beneficial to apply a symmetric permutation after the matching reordering to reduce fill-in. Another kind of instability in incomplete factorizations, which can occur with and without small pivots, is severe ill-conditioning of the triangular factors. (In this situation, $||R||_F$ need not be very large, but $||I - A(\hat{L}\hat{U})^{-1}||_F$ will be.) This is also a common situation when the coefficient matrix is far from diagonally dominant.

21

Table 7.6: Number of iterations required by some preconditioned iterative methods after permutation.

Matrix and method		Matching algorithm			
		MC21	BT	MPD	MPS
IMPCOL E					
ILU(0)	GMRES(20)	-	17	15	14
	BiCGSTAB	123	25	11	10
	QMR	101	25	17	16
ILU(1)	GMRES(20)	59	15	11	11
	BiCGSTAB	98	19	8	7
	QMR	72	21	12	12
ILUT	GMRES(20)	8	7	8	7
	BiCGSTAB	9	4	5	4
	QMR	10	7	8	8
MAHINDAS					
ILU(0)	GMRES(20)	-	-	179	116
	BiCGSTAB	-	-	39	38
	QMR	-	-	55	55
ILU(1)	GMRES(20)	-	-	69	58
	BiCGSTAB	-	-	26	21
	QMR	-	-	34	34
ILUT	GMRES(20)	-	-	15	13
	BiCGSTAB	-	151	11	8
	QMR	-	-	17	14
WEST0497					
ILU(0)	GMRES(20)	-	40	19	19
	BiCGSTAB	-	71	22	16
	QMR	-	48	23	21
ILU(1)	GMRES(20)	-	19	15	15
	BiCGSTAB	-	26	15	11
	QMR	-	30	18	16
ILUT	GMRES(20)	-	14	10	7
	BiCGSTAB	-	11	7	4
	QMR	-	15	12	7

22

We also performed a set of experiments in which we first permuted the columns of the matrix A by using a reordering computed by one of the matching algorithms, followed by a symmetric permutation of A generated by the reverse Cuthill-McKee ordering (Cuthill and McKee 1969) applied to $A + A^T$. The motivation behind this is that the number of entries that is dropped from the factors can be reduced by applying a reordering of the matrix that reduces fill-in. In the experimental results, we noticed that the additional permutation sometimes has a positive as well as a negative effect on the performance of the iterative solvers. Table 7.7 shows some results for the three iterative methods from Table 7.6 preconditioned by ILUT on the WEST matrices from the Harwell-Boeing collection.

Table 7.7: Number of iterations required by some ILUT-preconditioned iterative methods after the matching reordering with and without reverse Cuthill-McKee.

Matrix and method		Matching algorithm without RCM				Matching algorithm with RCM			
		MC21	BT	MPD	MPS	MC21	BT	MPD	MPS
WEST0497	GMRES(20)	-	14	10	7	14	15	10	5
	BiCGSTAB	-	11	7	4	60	23	10	3
	QMR	-	15	12	7	-	19	10	6
WEST0655	GMRES(20)	-	-	58	17	-	-	-	-
	BiCGSTAB	-	-	42	14	-	-	-	71
	QMR	-	-	38	18	-	-	-	76
WEST0989	GMRES(20)	-	-	-	-	-	37	13	8
	BiCGSTAB	-	-	280	-	262	35	9	5
	QMR	-	-	-	-	-	36	15	8
WEST1505	GMRES(20)	-	-	-	-	-	-	117	17
	BiCGSTAB	-	-	809	-	-	-	42	15
	QMR	-	-	-	-	-	-	60	20
WEST2021	GMRES(20)	-	-	-	-	-	-	36	11
	BiCGSTAB	-	-	-	-	-	-	26	7
	QMR	-	-	-	-	-	-	30	12

7.2.2 Experiments with a block iterative solution method

The Jacobi method is not a particularly current or powerful method so we focussed our experiments on the block Cimmino implementation of Arioli, Duff, Noailles and Ruiz (1992), which is equivalent to using a block Jacobi algorithm on the normal equations. In this implementation, the subproblems corresponding to blocks of rows from the matrix

are solved by the sparse direct method MA27 (HSL 1996).

We show the effect of this in Table 7.8 on the problem MAHINDAS from Table 7.6. The matching algorithm was followed by a reverse Cuthill-McKee algorithm to obtain a block tridiagonal form. The matrix was partitioned into 2, 4, 8, and 16 blocks of rows and the accelerations used were block CG algorithms with block sizes 1, 4, and 8. The block rows are chosen of equal (or nearly equal) size.

Table 7.8: Number of iterations of block Cimmino algorithm for the matrix MAHINDAS.

Acceleration + # block rows	Matching algorithm				
	None	MC21	BT	MPD	MPS
CG(1)					
2	324	267	298	295	105
4	489	383	438	438	141
8	622	485	532	524	167
16	660	572	574	574	175
CG(4)					
2	148	112	130	133	68
4	212	190	199	194	92
8	261	235	232	233	111
16	281	245	253	253	112
CG(8)					
2	80	62	72	75	54
4	117	105	109	108	71
8	140	133	127	130	84
16	151	142	137	136	90

In general, we noticed in our experiments that the block Cimmino method often was more sensitive to the scaling (in MPS) and less to the reorderings. The convergence properties of the block Cimmino method are independent of row scaling. However, the sparse direct solver MA27 (HSL 1996) used for solving the augmented systems, performs numerical pivoting during the factorizations of the augmented matrices. Row scaling might well change the choice of the pivot order and affect the fill-in in the factors and the accuracy of the solution. Column scaling should affect convergence of the method since it can be considered as a diagonal preconditioner. For more details see (Ruiz 1992).

8   Conclusions and future work

We have considered, in Sections 3-4, techniques for permuting a sparse matrix so that the diagonal of the permuted matrix has entries of large absolute value. We discussed various criteria for this and considered their implementation as computer codes. We also considered in Section 6 possible scaling strategies to further improve the weight of the diagonal with respect to the off-diagonal values.

In Section 7, we then indicated several cases where such a permutation (and scaling) can be useful. These include the solution of sparse equations by a direct method and by an iterative technique. We also considered its use in generating a preconditioner for an iterative method. The numerical experiments show that for a multifrontal solver and preconditioned iterative methods, the effect of these reorderings can be dramatic. The effect on the block Cimmino iterative method seems to be less dramatic. For this method, the discussed scaling tends to have a more important effect.

While it is clear that reordering matrices so that the permuted matrix has a large diagonal can have a very significant effect on solving sparse systems by a wide range of techniques, it is somewhat less clear that there is a universal strategy that is best in all cases. One reason for this is that increasing the size of the diagonal only is not always sufficient to improve the performance of the method. For example, for the incomplete preconditioners that we used for the numerical experiments in Section 7, it is not only the size of the diagonal but also the amount and size of the discarded fill-in plays an important role. We have thus started experimenting with combining the strategies mentioned in Sections 3-4 and, particularly for generating a preconditioner and the block Cimmino approach, with combining our unsymmetric ordering with symmetric orderings.

Another interesting extension to the discussed reorderings is a block approach to increase the size of diagonal blocks instead of only the diagonal entries and use for example a block Jacobi preconditioner on the permuted matrix. This is of particular interest for the block Cimmino method. One could also build other criteria into the weighting for obtaining a bipartite matching, for example, to incorporate a Markowitz cost so that sparsity would also be preserved by the choice of the resulting diagonal as a pivot. Such combination would make the resulting ordering suitable for a wider class of sparse direct solvers.

Finally, we notice in our experiments with MA41 that one effect of the matching algorithm was to increase the structural symmetry of unsymmetric matrices. We are exploring further the use of ordering techniques that more directly attempt to increase structural symmetry.

25

References

Amestoy, P. R. and Duff, I. S. (1989), 'Vectorization of a multiprocessor multifrontal code', *Int. J. of Supercomputer Applics.* **3**, 41–59.

Arioli, M., Duff, I. S., Noailles, J. and Ruiz, D. (1992), 'A block projection method for sparse matrices', *SIAM J. Sci. Stat. Comput.* pp. 47–70.

Benzi, M., Szyld, D. B. and van Duin, A. (1997), Orderings for incomplete factorization preconditioning of nonsymmetric problems, Technical Report LA-UR-97-3525, Los Alamos National Laboratory, Los Alamos, NM.

Burkard, R. E. and Derigs, U. (1980), *Assignment and Matching Problems: Solution Methods with FORTRAN-Programs*, Springer, Berlin-Heidelberg-New York. Lecture Notes in Economics and Mathematical Systems 184.

Carpaneto, G. and Toth, P. (1980), 'Solution of the assignment problem (Algorithm 548)', *ACM Trans. Math. Software* pp. 104–111.

Carraresi, P. and Sodini, C. (1986), 'An efficient algorithm for the bipartite matching problem', *European Journal of Operational Research* **23**, 86–93.

Chow, E. and Saad, Y. (1997), Experimental study of ILU preconditioners for indefinite matrices, Technical Report TR 97/95, Department of Computer Science, and Minnesota Supercomputer Institute, University of Minnesota, Minneapolis.

Cuthill, E. and McKee, J. (1969), Reducing the bandwidth of sparse symmetric matrices, *in* 'Proceedings 24th National Conference of the Association for Computing Machinery', Brandon Press, New Jersey, pp. 157–172.

Davis, T. A. (1997), University of Florida sparse matrix collection, Available at http://www.cise.ufl.edu/~davis and ftp://ftp.cise.ufl.edu/pub/faculty/davis.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S. and Liu, J. W. H. (1995), A supernodal approach to sparse partial pivoting, Technical Report UCB//CSD-95-883, Computer Science Division, University of California at Berkeley, CA. To appear in *SIAM J. Matrix Anal. Appl.*

Derigs, U. and Metz, A. (1986), 'An efficient labeling technique for solving sparse assignment problems', *Computing* **36**, 301–311.

Dijkstra, E. W. (1959), 'A note on two problems in connection with graphs', *Numerische Mathematik* **1**, 269–271.

Duff, I. S. (1981), 'Algorithm 575. Permutations for a zero-free diagonal', *ACM Trans. Math. Software* **7**(3), 387–390.

Duff, I. S. and Koster, J. (1997), The design and use of algorithms for permuting large entries to the diagonal of sparse matrices, Technical Report RAL-TR-97-059, Rutherford Appleton Laboratory, Oxfordshire, England. Accepted for publication in SIAM J. Matrix Anal. Appl.

Duff, I. S. and Reid, J. K. (1983), 'The multifrontal solution of indefinite sparse symmetric linear systems', *ACM Trans. Math. Software* **9**, 302–325.

Duff, I. S. and Wiberg, T. (1988), 'Remarks on implementations of $O(n^{1/2}\tau)$ assignment algorithms', *ACM Trans. Math. Software* **14**(3), 267–287.

Duff, I. S., Grimes, R. G. and Lewis, J. G. (1992), Users' guide for the Harwell-Boeing sparse matrix collection (Release 1), Technical Report RAL-92-086, Rutherford Appleton Laboratory, Oxfordshire, England.

Edmonds, J. and Karp, R. M. (1972), 'Theoretical improvements in algorithmic efficiency for network problems', *Journal of the ACM* **19**, 248–264.

Finke, G. and Smith, P. (1978), 'Primal equivalents for the threshold algorithm', *Op. Res. Verf.* **31**, 185–198.

Ford Jr., L. R. and Fulkerson, D. R. (1962), *Flows in Networks*, Princeton Univ. Press, Princeton, NJ.

Gallo, G. and Pallottino, S. (1988), 'Shortest path algorithms', *Annals of Operations Research* **13**, 3–79.

Hopcroft, J. E. and Karp, R. M. (1973), 'An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs', *SIAM J. Comput.* **2**, 225–231.

HSL (1996), *Harwell Subroutine Library, A Catalogue of Subroutines (Release 12)*, AEA Technology, Harwell Laboratory, Oxfordshire, England.

Jonker, R. and Volgenant, A. (1987), 'A shortest augmenting path algorithm for dense and sparse linear assignment problems', *Computing* **38**, 325–340.

Kuhn, H. W. (1955), 'The Hungarian method for the assignment problem', *Naval Research Logistics Quarterly* **2**, 83–97.

Li, X. S. and Demmel, J. W. (1998), Making sparse Gaussian elimination scalable by static pivoting, *in* 'Proceedings of Supercomputing', Orlando, Florida.

Olschowka, M. and Neumaier, A. (1996), 'A new pivoting strategy for Gaussian elimination', *Lin. Alg. and Its Appl.* **240**, 131–151.

Ruiz, D. (1992), Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment, PhD thesis, CERFACS, Toulouse, France.

Saad, Y. (1996), *Iterative methods for sparse linear systems*, PWS Publishing Company, Boston.

Tarjan, R. E. (1983), *Data structures and network algorithms*, SIAM, Philadelphia, USA. CBMS-NSF Regional conference series in applied mathematics 44.