

Design, Implementation, and Analysis of Maximum Transversal Algorithms

IAIN S. DUFF, RAL and CERFACS
KAMER KAYA, CERFACS
BORA UÇAR, CNRS and ENS Lyon

We report on careful implementations of seven algorithms for solving the problem of finding a maximum transversal of a sparse matrix. We analyze the algorithms and discuss the design choices. To the best of our knowledge, this is the most comprehensive comparison of maximum transversal algorithms based on augmenting paths. Previous papers with the same objective either do not have all the algorithms discussed in this article or they used nonuniform implementations from different researchers. We use a common base to implement all of the algorithms and compare their relative performance on a wide range of graphs and matrices. We systematize, develop, and use several ideas for enhancing performance. One of these ideas improves the performance of one of the existing algorithms in most cases, sometimes significantly. So much so that we use this as the eighth algorithm in comparisons.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems; G.2.2 [Discrete Mathematics]: Graph Theory

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Bipartite graphs, matching, breadth first search, depth first search, graph theory, matrix transversals, assignment

ACM Reference Format:

Duff, I. S., Kaya, K., and Uçar, B. 2011. Design, implementation, and analysis of maximum transversal algorithms. *ACM Trans. Math. Softw.* 38, 2, Article 13 (December 2011), 31 pages.
DOI = 10.1145/2049673.2049677 <http://doi.acm.org/10.1145/2049673.2049677>

1. INTRODUCTION

We study algorithms for the permutation of a sparse square matrix \mathbf{A} of order n so that the diagonal of the permuted matrix is zero free. This has many applications; our main motivation arises in linear system solution in which such a zero-free diagonal is found to enable the subsequent identification of irreducible blocks. Finding such a permutation corresponds to the classical topic of finding a maximum cardinality matching in a bipartite graph. In a bipartite graph, a subset \mathcal{M} of edges is called a *matching* if any vertex is incident on at most one edge in \mathcal{M} . Consider a bipartite graph $G_{\mathbf{A}} = (V_R \cup V_C, E)$ corresponding to a given sparse matrix \mathbf{A} where the vertex sets V_R and V_C correspond to the rows and the columns of \mathbf{A} , respectively, so that for $i \in V_R$ and $j \in V_C$, the edge $(i, j) \in E$ exist iff $a_{ij} \neq 0$. A zero-free diagonal in an $n \times n$ sparse matrix

Author's addresses: I. S. Duff, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, UK OX11 0QX; email: iain.duff@stfc.ac.uk; K. Kaya, CERFACS, 42 Avenue Gaspard Coriolis, 31057 Toulouse Cedex 01 France; email: kamer@cerfacs.fr; B. Uçar, Laboratory of Parallel Computing, ENS Lyon, 46, Allée, d'Italie, F-69364, Lyon Cedex 07 France; email: bora.ucar@ens-lyon.fr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2011 ACM 0098-3500/2011/12-ART13 \$10.00

DOI 10.1145/2049673.2049677 <http://doi.acm.org/10.1145/2049673.2049677>

Table I. Eight Maximum Cardinality Bipartite Matching Algorithms Are Implemented in This Work. The first seven are from the literature with slight enhancements. DFS and BFS denote the well known depth and breadth first search techniques, respectively. The complexity of the algorithms is stated in terms of n and τ where n is the smaller of the cardinalities of the two vertex sets of the bipartite graph, and τ is the number of edges

Algorithm	Description	Complexity
DFSB	Based on DFS. Quite common in software libraries.	$\mathcal{O}(n\tau)$
BFSB	Based on BFS. Also quite common, see the algorithm FF in Mehlhorn and Näher [1999]	$\mathcal{O}(n\tau)$
MC21A	DFS+Lookahead [Duff 1981]. Also <code>dmperm</code> in Matlab [Davis 2006]	$\mathcal{O}(n\tau)$
PF	Disjoint DFSs [Pothen and Fan 1990]	$\mathcal{O}(n\tau)$
HK	Shortest disjoint augmenting paths [Hopcroft and Karp 1973]	$\mathcal{O}(\sqrt{n}\tau)$
HKDW	HK+Disjoint DFS [Duff and Wiberg 1988]	$\mathcal{O}(\sqrt{n}\tau)$
ABMP	Combined DFS and BFS [Alt et al. 1991]	Minimum of $\mathcal{O}(\sqrt{n}\tau)$ and $\mathcal{O}(n^{1.5}\sqrt{\tau/\log n})$
PF+	A simple modification of PF (this article)	$\mathcal{O}(n\tau)$

corresponds to a matching of cardinality n in G_A . Such matchings containing all the vertices of a bipartite graph are called *perfect*, and the corresponding diagonal in the associated matrix is called a *transversal*. Clearly, not all matrices have a transversal; in which case one looks for a permutation with the maximum number of nonzero diagonal entries.

There are a number of algorithms that can be used to find maximum matchings in bipartite graphs. The first aim of our article is to review those algorithms. We have implemented seven existing algorithms that are described in Table I. We give the distinguishing features of each algorithm in the second column. These features will be fully discussed in Section 3 where we consider each algorithm in detail. It suffices, for the moment, to note that all these algorithms start from a given, possibly empty, matching and increase the cardinality of the matching by finding and exploiting *augmenting paths* (see Section 2). The seven algorithms differ only in the way the augmenting paths are found. We note that these seven algorithms cover essentially all known methods based on augmenting paths for the maximum cardinality bipartite matching problem. We carefully implement all the algorithms with unified data structures and fine tune each one to improve its performance. Some of these algorithms are available from different sources; however, their implementation is far from uniform. They are implemented in different programming languages, have different data structures; some of them include techniques to speed up the computations while some others are discussed and implemented without those techniques. We try to incorporate all the known techniques into the seven algorithms. We believe that without this sort of uniformity, comparisons between different algorithms would not necessarily be fair, and the computational results would not necessarily be conclusive. We provide a large set of experiments on which we evaluate the algorithms. We conclude for example that the DFS-based implementations given in the table can have an unexpectedly long execution time compared to the others.

We also propose a simple modification of the PF algorithm. With this modification, the PF algorithm becomes much less sensitive to the order of the augmenting path searches. We refer to this alternative as PF+ throughout the text. When this modification helps, the execution time of PF reduces significantly; when it does not help, the execution time can increase, but only by a little. We also improve the implementation of HKDW and make it faster than the one implemented by Duff and Wiberg [1988].

In addition to the exact algorithms listed here, there are several fast heuristics that can be used to generate a large initial matching quickly prior to the execution of the

main algorithm. The simplest of these, SGM, is quite commonly used, see, for example, [Duff 1981] (included as MC21 in the HSL Mathematical Software Library, available at <http://www.hsl.rl.ac.uk/>) and the maximum cardinality bipartite matching algorithms in LEDA [Mehlhorn and Näher 1999]. However, there are more sophisticated and better heuristics such as the Karp-Sipser heuristic, KSM [Karp and Sipser 1981], the minimum-degree heuristic MDM [Langguth et al. 2010; Magun 1998] and some others proposed by Langguth et al. [2010]. We implement and use SGM, KSM and MDM as the jump-start routines in our experiments.

Although previous studies that compare the performance of some of the algorithms in this article exist [Cherkassky et al. 1998; Duff 1981; Duff and Wiberg 1988; Langguth et al. 2010; Mehlhorn and Näher 1999; Pothén and Fan 1990; Setubal 1996], we think that they are not conclusive because of the following two reasons. First, all previous studies, except Langguth et al. [2010], use either a simple greedy approach (much like SGM) or none as the jump-start routine. Second, as also stated by Langguth et al., the algorithms were implemented by different researchers in different programming languages, with different data structures, and hence their comparison is not necessarily fair.

The organization of the article is as follows. We present background material in the next section. The exact algorithms and the proposed modification of PF are described in Section 3, and the heuristics are discussed in Section 4. Section 5 gives the experimental results. We discuss the results further and make some concluding remarks in Section 6.

2. BACKGROUND AND NOTATION

In a bipartite graph $G = (V_1 \cup V_2, E)$, a subset \mathcal{M} of E is called a *matching* if a vertex in $V = V_1 \cup V_2$ is incident to at most one edge in \mathcal{M} . A matching \mathcal{M} is called *maximal*, if no other matching $\mathcal{M}' \supset \mathcal{M}$ exists. A maximal matching \mathcal{M} is called *maximum* if $|\mathcal{M}| \geq |\mathcal{M}'|$ for every matching \mathcal{M}' where $|\cdot|$ denotes the cardinality of a set. Furthermore, if $|\mathcal{M}| = |V_1| = |V_2|$, \mathcal{M} is called a *perfect* (complete) matching. Note that a perfect matching \mathcal{M} is maximum (hence maximal), and each vertex in V is incident to exactly one edge in \mathcal{M} . The *deficiency* of a matching \mathcal{M} is the difference between the maximum cardinality of a matching and $|\mathcal{M}|$. A good discussion on matching theory can be found in Lovasz and Plummer's book [1986].

Let \mathcal{M} be a matching in G . A vertex $v \in V$ is *matched* (by \mathcal{M}) if it is incident on an edge in \mathcal{M} ; otherwise, it is *unmatched*. A path in G is *\mathcal{M} -alternating* if its edges alternate between those in \mathcal{M} and those not in \mathcal{M} . An \mathcal{M} -alternating path \mathcal{P} is called *\mathcal{M} -augmenting* if the start and end vertices of \mathcal{P} are both unmatched. The following theorem is the underlying principle of the maximum cardinality matching algorithms discussed in this article.

THEOREM 2.1 [BERGE 1957]. *Let G be a graph (bipartite or not) and \mathcal{M} be a matching in G . Then \mathcal{M} is of maximum cardinality if and only if there is no \mathcal{M} -augmenting path in G .*

Following Theorem 2.1, several algorithms have been proposed for the maximum matching problem. Essentially, algorithms based on augmenting paths follow the same pattern: given a possibly empty matching \mathcal{M} , the algorithm searches for an \mathcal{M} -augmenting path \mathcal{P} . If none exists, then the algorithm stops since the matching is maximum. Otherwise, the alternating path \mathcal{P} is used to increase the cardinality of \mathcal{M} by setting $\mathcal{M} = \mathcal{M} \oplus E(\mathcal{P})$ where $E(\mathcal{P})$ is the edge set of a path \mathcal{P} , and $\mathcal{M} \oplus E(\mathcal{P}) = (\mathcal{M} \cup E(\mathcal{P})) \setminus (\mathcal{M} \cap E(\mathcal{P}))$ is the symmetric difference. How they find this augmenting path and other implementation details are what differentiates the algorithms based on augmenting paths, both in theory and in practice.

We use the following notation. The vertex sets V_R and V_C correspond to the sets of rows and columns, respectively, of a given $m \times n$ sparse matrix \mathbf{A} where without loss of generality $m \geq n$. Hence, we have $|V_R| = m$ and $|V_C| = n$. For a square matrix, we use n to denote its order. The number of nonzeros in the matrix, or the number of edges in the bipartite graph is denoted by τ .

There is a one-to-one correspondence between a zero-free diagonal in a (permuted) square sparse matrix \mathbf{A} and a perfect matching in the associated bipartite graph G_A . Let \mathcal{M} be a perfect matching in G_A . Then one can build an $n \times n$ permutation matrix \mathbf{Q} such that $Q_{ji} = 1$ iff row i and column j are matched in \mathcal{M} so that the matrix $\mathbf{A}\mathbf{Q}$ has a zero-free diagonal. If \mathbf{A} is not square or \mathcal{M} has a cardinality $\ell < n$, then one can find two permutation matrices \mathbf{P} and \mathbf{Q} which permute the matched rows and columns to the first ℓ positions. After these permutations, \mathcal{M} becomes a perfect matching for the $\ell \times \ell$ principal submatrix of $\mathbf{P}\mathbf{A}\mathbf{Q}$ and therefore can be used to obtain a zero-free diagonal for that submatrix.

3. MATCHING ALGORITHMS BASED ON AUGMENTING PATHS

We start by mentioning the common data structures for the implementation of the algorithms. The first two structures are for the pattern of the sparse matrix and the matching itself.

In all the algorithms that we review, the pattern of a sparse matrix is stored in either the *compressed column storage* (CCS) or *compressed row storage* (CRS) formats, or both. These are well known storage formats for sparse matrices (see, e.g., Duff et al. [1986, Section 2.7]). Consider an $m \times n$ sparse matrix \mathbf{A} with τ nonzeros. In CCS, the pattern of \mathbf{A} is stored in two arrays:

- rids*[1, . . . , τ] stores the row index of each nonzero entry. The nonzeros in a column are stored consecutively; and
- cptrs*[1, . . . , $n + 1$] stores the location of the first nonzero of each column in array *rids* where *cptrs*[$n + 1$] = $\tau + 1$. In particular, the row indices of the nonzeros in column j are stored in *rids*[*cptrs*[j], . . . , *cptrs*[$j + 1$] - 1].

The CRS of a matrix \mathbf{A} is the CCS of its transpose and vice-versa. In CRS, there are again two arrays *cids* and *rptrs*, with functions similar to the two arrays just described. Both formats may be necessary for an efficient implementation of some of the algorithms. We will specify which storage format is required for each of the eight algorithms that we implement.

The matching at any point during the course of the algorithms is stored in two arrays. The array *rmatch*[1, . . . , m] stores the index of the columns matched to the rows; *rmatch*[i] = j if the row i is matched to the column j , otherwise *rmatch*[i] = -1. The array *cmatch*[1, . . . , n] stores the dual information similarly. In what follows, all the matching related definitions are with respect to a matching \mathcal{M} represented by these two arrays.

As stated in Theorem 2.1, a matching \mathcal{M} is maximum if and only if there is no \mathcal{M} -augmenting path. Maximum cardinality matching algorithms based on augmenting paths exploit this fact and use graph search techniques to find an augmenting path if it exists. These algorithms mainly use depth first search (DFS), breadth first search (BFS) or a combination of these. We will organize the algorithms into these three classes and describe the implementation details.

The way the DFS and BFS procedures are used is also common to all eight algorithms. Both of the search procedures are started from an unmatched row or an unmatched column (depending on the algorithm). Then, from a vertex, some of the edges are allowed to be traversed. For example, if a search is started from an unmatched column, then at any column vertex only the edges that are not in the matching are allowed,

whereas at any row vertex only a matching edge is allowed, if any. If we reach a row vertex with no matching edge, we have found an augmenting path. During a search with DFS or BFS, each vertex is visited only once, hence the search constructs a tree containing alternating paths and possibly, an augmenting path. Such a tree is called an *alternating search tree*. If a search is started from an unmatched column then, when the search finds a matched row, the search immediately continues from the column matched to that row.

For maximum transversal algorithms, a desired property is *robustness*. We call an algorithm *robust* if its performance is not highly sensitive to permutations of the input graph. A permutation only relabels the vertices of a graph, and therefore, it affects only the order in which the vertices are visited. In such cases, the running time of the algorithm should not drastically change. We provide the following table for the matrix *Hamrle3* from the University of Florida sparse matrix collection. We run the eight algorithms on the matrix itself and on random permutations of it. We investigate three types of random permutations: rows are permuted (**PA**), columns are permuted (**AQ**), and both rows and columns are permuted (**PAQ**). For a robust algorithm, the runtime should not change dramatically. However, as seen in the table below, quite amazingly almost all the algorithms are highly sensitive to the permutations, with PF+ and HKDW displaying more robust behavior than the others.

permutation	DFSB	BFSB	MC21A	PF	PF+	HK	HKDW	ABMP
A	267.3	117.0	5.4	0.1	2.6	0.8	0.3	47.8
AQ	291.6	20.5	238.8	101.9	11.1	74.0	44.3	241.7
PA	601.0	204.2	324.7	18.7	17.7	2.9	2.8	18.2
PAQ	296.2	78.4	129.3	51.7	48.9	106.0	67.3	76.2

The execution times given in the table are the averages for ten different random permutations (e.g., the times in row **AQ** of the table correspond to the average runtime for ten different column permutations). Further experiments of this sort can be found in Section 5. Of course, it is all very well for an algorithm to be robust but we are primarily concerned with its efficiency, that is having an algorithm with fast execution.

3.1. Algorithms Based on Breadth-First Search

BFS-based algorithms can be implemented to start either from the columns or the rows. We choose to start from the columns as we are assuming, without loss of generality, that there are less columns than rows. If this is not the case, one can work with the transposed matrix.

In a BFS-based matching algorithm, BFSB, the unmatched column vertices are processed in some order (normally from 1 to n). While processing an unmatched column vertex c , a BFS is started from c to find an unmatched row while building the corresponding alternating tree. If an unmatched row r is found (one with the shortest distance to the column vertex c), the matching is augmented using the unique alternating path between c and r in the alternating search tree, and the algorithm proceeds to the next unmatched column vertex. If the BFS terminates without finding an unmatched row, then the starting column vertex c remains unmatched, and there is no perfect matching. The algorithm then proceeds to the next unmatched column to start a BFS. Figure 1 shows a sample search tree where the root is an unmatched column c . For this example, the BFS stops when the unmatched row r is found with the corresponding augmenting path c, r_3, c_3, r_6, c_6, r . The levels of the vertices (their depth in the search tree) are also given in Figure 1.

To implement BFSB, we need a *queue* of size n and another array of size m to mark the rows as *visited*. Note that we only put columns in the queue, since we begin with an unmatched column. This works because of the nature of an alternating tree. That is, we start with a queue containing an unmatched column c_i and repeat the following

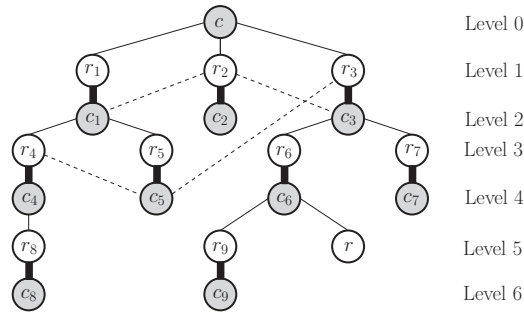


Fig. 1. A BFS tree constructed from finding alternating paths from an unmatched column vertex c . The edges in the matching are shown as thick lines, and those that do not belong to the tree are shown as dashed.

until the queue is empty or an augmenting path found: we dequeue a column c_j and traverse all of its adjacent rows. For each row r_k in this set, if r_k is unmatched we use the corresponding augmenting path to increase the size of the matching and stop the current search process. Otherwise, if r_k is matched and unvisited, it is marked as visited, that is, the corresponding entry of the *visited* array is set to i , and the column matched to r_k is enqueued. This process, a BFS search from an unmatched column, is called a phase of BFSB. This phase is carried out for each unmatched column in order. The efficiency of the algorithm might depend therefore on that order. Since the complexity of a phase is $\mathcal{O}(\tau)$, and since there are n columns, the time complexity of the overall algorithm BFSB is $\mathcal{O}(n\tau)$.

The *queue* and *visited* arrays are sufficient to perform a BFS from an unmatched column c . However, when an augmenting path is found, we need to find the path by going backwards in the tree. To be able to do this efficiently, we also use another array of size m to store the *parent* column for the visited rows in the search tree. Hence, in addition to the space used to store the matrix structure and matching information, an efficient implementation of the BFS-based algorithm requires $n + 2m$ more space. Note that for the BFS we only need to reach the adjacent rows of the columns. Hence, storing the matrix only in the CCS format is sufficient for this algorithm.

In our implementation, we incorporated a *pruning heuristic* (also implemented in practice by Setubal [1996]), which may reduce the execution time when the matrix is rank deficient, that is, when the corresponding graph does not contain a perfect matching. When a BFS is unsuccessful, we can prune all rows and columns visited during this BFS so that any future search will not visit them. This pruning is possible since the edges in the alternating search tree cannot be modified by any further augmentations—there is no augmenting path passing through these vertices. Note that the array *queue* is of size n . Hence, we have enough space to keep all of the visited columns during a search. We do not put rows into the *queue* since for a row, we can use only the matching edge. Therefore, after an unsuccessful BFS, we can prune these columns and their matched rows. To prune a row, we set its visited entry to $n + 1$ and do not visit such rows during the subsequent BFSB phases.

To guarantee that a vertex is visited only once during a BFS, we modify the corresponding entry in the *visited* array (similar techniques are quite common in sparse matrix folklore). That is, when the algorithm is searching for an augmenting path for an unmatched column c_i , if a row r is visited we set $visited(r) = i$ and visit a row r only if $visited(r) < i$. Hence, we initialize the *visited* array once before starting the BFSs so that $visited(r) = -1$ for all $r \in V_R$, and do not reset it for every BFS.

The additional space required by the matching algorithm can be reduced to $n + m$ by using the *parent* array to check if a row has been visited. A straightforward

implementation achieves that by resetting the *parent* array to -1 . On the other hand, a different approach avoids the reset operation after each augmentation and uses a distinct value for each BFS from the unmatched columns. A simple implementation of this approach assigns the parent column of a row r as $parent(r) = c' + n \times i$ where $c' \in \{0, \dots, n-1\}$ is equal to $c-1$ for the c th column and i is the number of augmentations done so far. Note that these calculations can result in overflows if not implemented carefully. Hence to check if a row r is visited or not, it is sufficient to check if $\lfloor parent(r)/n \rfloor = i$ or not, and the id of the parent column can be obtained by $(parent(r) \bmod n) + 1$. Note that such techniques can be used in practice to reduce the memory requirement of the algorithms described in this section. However, these techniques may slightly increase the execution times; we favor the runtime in our implementations and do not incorporate these memory optimizations.

3.2. Algorithms Based on Depth-First Search

There are three known DFS-based algorithms. The first one is DFSB that implements the same approach as in the previous section by replacing the BFSs with DFSs. Among the other two, MC21A uses a mechanism to improve DFSB, whereas the other one PF reorganizes DFS searches into disjoint sets. We will review these three algorithms in the next three subsections and propose a simple yet effective modification to PF resulting in the fourth algorithm PF+. We discuss the column-oriented version of these algorithms as we did in the previous section.

3.2.1. DFSB: The Pure Depth-First Search-Based Algorithm. In DFSB, the unmatched column vertices are processed in some order (normally from 1 to n). For each unmatched column c , a DFS is initiated, and when a column c_j is visited, the search path is extended with the next unvisited row of c_j whereas when a row r_i is visited, the search path is extended with the column matched to r_i . If such a column does not exist, that is, if r_i is unmatched, then the search has found an augmenting path between c and r_i . During the DFS, we store the columns of the current path in a *stack* of size n . When the path cannot be extended any longer, that is, when all of the rows of the last column c_j in the stack have been visited by the current DFS, we remove c_j from the stack and continue with the previous entry on the stack. We repeat this process until an augmenting path is found for the starting vertex c or the stack is empty. This process is called a *phase* of DFSB or a DFS phase. Figure 2 shows the search tree constructed by the vertices and edges used in a DFS initiated from the unmatched column c in the graph of Figure 1. Note that, in Figure 2, after c_1 is visited, before r_5 , all of the nodes under alternating subtrees rooted at r_2 and r_4 are visited. Similarly, after c_6 is visited, the augmenting path to r is not found before visiting r_9 and c_9 .

To implement a DFS-based algorithm, in addition to the stack, we need another array of size m to mark the *visited* rows. We also use an array *lastrow* of size n to keep a pointer to the last visited row of each column during a DFS. With this array, when looking for an unvisited row of the current column c_j , we start the search from position $(lastrow(c_j) + 1)$ in c_j 's adjacency list. This allows us to avoid multiple scans of adjacency lists. We reset *lastrow*(c_j) each time we put a column c_j into the stack. With this approach, each edge is scanned only once during a phase. Hence, each DFS has $\mathcal{O}(\tau)$ complexity, and the overall complexity is $\mathcal{O}(n\tau)$. Note that, when an unmatched row r is found in a DFS phase, the columns of the corresponding augmenting path from c (for which the initial DFS was started) to r are already in the stack in the reverse order. Therefore, unlike BFSB, we do not need to care about storing information to go backwards in the search tree for augmentation (popping up the stack suffices). Hence, our implementation requires $2n + m$ additional space (or $2n$ since one can use a technique similar to that in Section 3.1 and use *lastrow*(c_j) to check if the row r_i ,

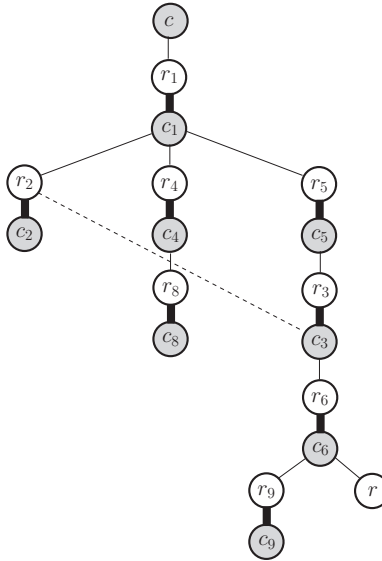


Fig. 2. A DFS tree constructed while searching for an augmenting path from an unmatched column node c in the graph shown in Figure 1. Matching edges are shown with thick lines, where the edges between columns and already visited rows are shown as dashed.

matched with column c_j , has been visited or not). Note that, for a DFS, we only need to reach adjacent rows of the columns. Hence, storing the matrix in the CCS format is sufficient for this algorithm.

Similar to the BFSB algorithm, we do not reset the *visited* array after each DFS. Also, the pruning heuristic of Section 3.1 is implemented by first putting each popped column to the end of the *stack* array and pruning the rows matched with these columns by modifying their *visited* entries. Here, we only use the unused space in the array holding the stack as temporary storage but not the stack data structure. We adapt the same ideas for avoiding resets and pruning rows for the other DFS-based algorithms in this subsection or the hybrid algorithms in the next subsection.

3.2.2. MC21A: Duff's Algorithm. MC21A is a DFS-based algorithm with an enhancement on the search process [Duff 1981]. Note that, in DFSB, the rows adjacent to a column are visited according to their order in the adjacency list, even if there is an unmatched row among them. In order to reach that unmatched row to find an augmenting path, a pure DFS-based algorithm may need to explore a large part of the graph and hence may be very costly. To alleviate this problem, a mechanism called *lookahead* is used [Duff 1981; Gustavson 1976; Pothen and Fan 1990]. With this mechanism, when a column c is visited, the algorithm first checks if c has an unmatched row r in its adjacency list, and if there is one, it uses the corresponding augmenting path. Otherwise, it continues with the usual DFS process.

For the *lookahead* mechanism, we use an array of size n to store the last checked row in the adjacency list of each column, initially pointing to the start of the adjacency list. When a column c is visited during the course of the overall algorithm (not a phase), *lookahead*(c) is incremented until an unmatched row is found or the end of the adjacency list is reached. The first case results in an augmenting path, and the second case tells the algorithm that *lookahead* for column c is no longer needed. Note that DFSs passing through a column will proceed as usual. The key observation is that a matched row will remain matched during the course of the algorithm. Lookahead

therefore adds at most $\mathcal{O}(n + \tau)$ time to the algorithm. Hence the complexity of MC21A is the same as that of DFSB which is $\mathcal{O}(n\tau)$, although *lookahead* improves the runtime significantly.

We also use the pruning heuristic described in Section 3.1, which is not a part of the original implementation of MC21A by Duff [1981].

In addition to the space used to store the matrix structure and matching information, MC21A requires $3n + m$ additional space for the arrays *stack* (size n), *visited* (size m), *lastrow* (size n), and *lookahead* (size n). For this algorithm, we only need to access the rows adjacent to a column. Hence, storing the matrix in the CCS format is sufficient.

3.2.3. PF: Pothen and Fan's Algorithm. PF is similar to MC21A. It initiates DFSs from unmatched columns, uses a lookahead mechanism in DFSs, and visits a vertex at most once in a phase. The only difference lies in the definition of a phase. An MC21A phase is simply a DFS from an unmatched column. On the other hand, a PF phase performs a maximal set of vertex disjoint DFSs each starting from a different unmatched column. Each DFS in a phase can visit only vertices that are not already visited in the phase. Hence, whereas the total number of DFSs in MC21A is n , each potentially taking $\mathcal{O}(n + \tau)$ time, the number of DFSs in a single phase of PF can be n , which still takes $\mathcal{O}(n + \tau)$ time. In our implementation, we maintain an array containing the *unmatched* columns and, if the DFS initiated from a column c is successful, we remove c from the *unmatched* array. Otherwise, c stays in the list, and a DFS is initiated from it in the next phase.

To find the matching with maximum cardinality, PF executes several phases with a reduced set of unmatched columns until no augmenting path is found in a phase. In the worst case, there will be one augmentation in each phase and hence n phases in total. Since a vertex is visited just once in a phase, the complexity of each phase is $\mathcal{O}(\tau)$, and the overall complexity is $\mathcal{O}(n\tau)$. Similar to MC21A, the *lookahead* entries initially point to the start of the adjacency list for each column, and they do not need to be reset again. Hence, the total complexity of the lookahead is again $\mathcal{O}(n + \tau)$, which does not change the overall complexity.

Note that in BFS- or DFS-based algorithms with the traditional phase definition as in MC21A, when a search for an unmatched column c is unsuccessful, it implies that c will remain unmatched in the maximum matching at the end. However, in PF, an unsuccessful search from a vertex c does not prove the lack of an augmenting path for c since, in a phase, the search is restricted to use only the vertices that are not visited by the previous searches performed in this phase. Hence, the algorithm stops when all the DFSs in a phase are unsuccessful. With a similar reasoning, when a DFS is unsuccessful, we cannot prune the rows visited by the current DFS unless this is the first DFS of a phase or all the previous DFSs in the current phase were unsuccessful. If this is the case, we can employ the pruning process as in MC21A and also remove a column c from the *unmatched* array when the DFS from c is unsuccessful. We incorporated this pruning mechanism in our implementation of PF as it entails only a little overhead. The original implementation by Pothen and Fan [1990] does not use this mechanism.

Similar to MC21A, PF can be implemented by using $3n + m$ more space for the arrays *stack* (size n), *visited* (size m), *lastrow* (size n), and *lookahead* (size n). In addition to these, we use an array *unmatched* (size n) to keep track of the remaining unmatched columns after each PF phase. Note that an implementation that uses the stack memory to store unmatched columns is straightforward. However, this may increase the execution time slightly, and we again choose performance over memory and use this additional array. Hence, the total memory requirement of our implementation is $4n + m$.

PF only needs to access the adjacent rows of the columns. Hence, storing the matrix in the CCS format is sufficient.

3.2.4. PF+: A modification of PF. We have found PF to be efficient in practice, except that it is sensitive to row and column permutations. To make it less sensitive to permutations, we propose a simple modification. This modification usually improves the performance of PF and increases its robustness; in some cases, it results in remarkable speedups, and in all cases the overhead is negligible.

The modification we propose is to change the order in which the rows in the adjacency lists of columns are visited and to apply an alternating scheme. To implement PF+, we count the number of PF+ phases and during the DFSs in an odd-numbered phase, we traverse the adjacency list of a column from left to right whereas, during an even numbered phase, we scan the adjacency lists in the reverse order, that is, the last row in the adjacency list is the first one to be investigated by the DFSs. The purpose of this modification is to avoid always preferring one row to another because of their positions in the original ordering. The hope is that this will spread the search more evenly in the graph and will find an unmatched row more quickly. Note that this modification does not increase or reduce the complexity of PF. Also the other components of the algorithm, that is the pruning scheme, the lookahead scheme (no need to implement two lookaheads for the two visit orders), and the memory requirements are exactly the same in both algorithms.

The proposed modification can be used for any DFS-based algorithm such as DFSB and MC21A. We choose to use it for PF, since in PF, multiple DFS searches are initiated from the same unmatched vertex during the execution, whereas in the other two, a DFS is started only once from a vertex. With this modification, it is therefore possible to alternate the vertex visit order between two DFS searches that start from the same vertex (when the one executed in the preceding phase was unsuccessful). In the experiments, we have found PF to be faster than DFSB and MC21A, and we wanted to see if we could further improve its relative performance.

3.3. Algorithms Based on Both Breadth- and Depth-First Search

All of the algorithms described so far are based on either BFS or DFS and have a runtime complexity of $\mathcal{O}(n\tau)$. Here, we will describe other algorithms with better worst-case complexities that use DFS and BFS together, and the level information of the vertices that is obtained by the BFSs. Note that the level of a vertex, which is its depth in a BFS tree started from an unmatched column c , is the shortest alternating path length starting from c and ending at that vertex.

3.3.1. HK: Hopcroft and Karp's Algorithm. HK organizes the search for augmenting paths into phases [Hopcroft and Karp 1973]. In each phase, the algorithm starts a BFS search from all unmatched columns to find a set of shortest-length augmenting paths in the graph. Among those paths, a maximal set of disjoint augmenting paths are found using a restricted DFS, and this set of augmentations are applied to the current matching altogether. The algorithm then continues with the next phase until a maximum matching is found. Hopcroft and Karp [1973] proved that a maximum matching is obtained after at most \sqrt{n} phases and hence HK has a theoretical complexity of $\mathcal{O}(\sqrt{n}\tau)$.

A HK phase has two parts. It first starts with a BFS from a fictitious vertex that has all the unmatched columns in its adjacency list. This can be seen as a *combined BFS* from all unmatched columns, where the queue is initialized with all the unmatched columns. Unlike the original BFS-based matching algorithm, the combined BFS in HK is not intended to increase the cardinality of the current matching; it is used to assign level numbers to the vertices. In this scheme, the unmatched columns are on level zero. For level ℓ , the next level $\ell + 1$ has the set of the vertices in the queue after the last

vertex from ℓ is processed. As an example, if the level zero contains the unmatched columns, the first level is the set of rows which are adjacent to at least one unmatched column, and the second level is the set of columns matched with a row in the first level. The search process continues until an unmatched row is found at level L and all columns at level $L - 1$ are processed. Note that L is the shortest augmenting path length in the graph. The rows at level L are stored in a *stack*.

The second part of a phase of HK uses the level information found by the combined BFS of the first part to find augmenting paths. In the original description of HK [Hopcroft and Karp 1973], an *auxiliary graph* is constructed explicitly after the combined BFS. This graph contains vertices visited by the combined BFS, and the edges (u, v) where u and v belong to two adjacent levels. After the auxiliary graph is constructed, a DFS is initiated from each unmatched row in level L such that each vertex in the auxiliary graph is visited once by these DFSs. The DFSs are in the reverse direction, that is, from level L unmatched rows to level 0 unmatched columns, and they will find a maximal set of length L augmenting paths in the auxiliary graph. During the search process, while exploring a row, an unvisited column is chosen from its adjacency list (if there is none, the DFS backtracks as usual). On the other hand, while visiting a column, the next visited vertex will be its matched row, if it exists. If not, the current column is unmatched (it is in level 0). Therefore, the DFS is terminated, and the corresponding augmenting path is used to increase the size of the matching. After the DFS terminates (either by finding an unmatched column for the current level L row, or by backtracking until a row at level L), another DFS (restricted to avoid visiting the nodes again) is initiated from another unmatched row at level L . Except for initiating the DFSs from the unmatched rows instead of the unmatched columns, the second part of a HK phase is similar to a PF phase where each vertex is visited only once.

As previously stated, if a BFS is initiated from an unmatched column c , it will find an unmatched row r where the corresponding augmenting path between c and r is a shortest augmenting path for c . Hence, L , the maximum assigned level in the first part, is the length of the shortest augmenting path with respect to the current matching. Hopcroft and Karp [1973] proved the following: (i) for any bipartite graph with maximum matching cardinality δ_{max} , if the cardinality of a given matching \mathcal{M} is $\delta_{max} - k$, there are k vertex disjoint \mathcal{M} -augmenting paths; (ii) augmenting the matching using a maximal set of vertex disjoint augmenting paths of the shortest length strictly increases the shortest augmenting path length in the graph. These two results are used to demonstrate the time complexity of HK as follows [Hopcroft and Karp 1973]. It is obvious that if the cardinality of the maximum matching is smaller than \sqrt{n} , there will be at most \sqrt{n} phases. Assume the contrary, and let \mathcal{M} be the current matching with cardinality $\delta = \delta_{max} - \sqrt{n}$. From the first result there are \sqrt{n} vertex disjoint \mathcal{M} -augmenting paths. Hence, the shortest \mathcal{M} -augmenting path length can be at most $2\sqrt{n} - 1$. From this and the second result, we know that up to now, we have performed less than \sqrt{n} phases. Since the current matching has $\delta_{max} - \sqrt{n}$ pairs, the algorithm will run for at most \sqrt{n} more phases. Therefore, the total number of phases is $\mathcal{O}(\sqrt{n})$ for HK. In each phase, the algorithm executes a combined BFS and some DFSs in which each edge is used a constant number of times. Therefore, the time spent in a phase is $\mathcal{O}(\tau)$, and the time complexity is $\mathcal{O}(\sqrt{n}\tau)$.

The HK algorithm is also interpreted as a variant of Dinic's algorithm proposed for maximum flow problems (see Even and Tarjan [1975] for this interpretation). Even and Tarjan [1975, Theorem 5] proved that if an algorithm uses shortest augmenting paths to increase the cardinality of the matching (as in HK), the total length of all augmenting paths is at most $\mathcal{O}(n \log n)$. Note that if an augmenting-path-based algorithm does not use the shortest augmenting paths, there is a possibility that an augmenting path can

contain all of the matching edges, in which case the total length of augmenting paths through the course of the algorithm can be $\mathcal{O}(n^2)$.

As previously stated, for the combined BFS, we put all the unmatched columns in a *queue* of size n at the beginning, start the BFS and stop after discovering all the vertices in the first level L containing an unmatched row. During the combined BFS, we put unmatched level L rows in a stack of size m . Note that the auxiliary graph requires $\mathcal{O}(\tau)$ memory and a significant amount of time for its construction. For this reason, as also proposed by Duff and Wiberg [1988], instead of an auxiliary graph, we store the level numbers of the columns in a *levels* array of size n with the maximum level L , and we use them to restrict the DFSs in the second part to make the searches stay in the auxiliary graph. Hence, in a DFS when a row is visited in the second part, the next column to be visited will be one of the unvisited columns at the previous level. For the second part, we also use a *nextcol* array of size m to scan each edge once during the DFSs. Hence, in total, we need $2n + 2m$ memory (*stack*(m), *queue*(n), *nextcol*(m), *levels*(n)) for an efficient implementation of an HK phase, with its two parts. To be consistent with the previous implementations and for efficiency, we also use a visited array of size $\max(m, n)$, to check if a row in the first part of a phase and a column in the second part of a phase are visited or not.

For HK, we access both the adjacent rows of the columns and the adjacent columns of the rows. Hence we need to store the matrix in both CCS and CRS formats. We now propose a modification that reduces the storage requirement of HK. Assuming that the matrix is stored only in CCS format, in the first part, the combined BFS is executed from the unmatched columns, and the level information is obtained. In this modification, we obtain the level information not for the columns but for the rows, including the unmatched rows at the highest level L . Since we have the level information and the CCS, a DFS can be executed starting from all unmatched columns (instead of rows) ending with the unmatched rows at the highest level. Since this modified second part also finds a maximal set of disjoint shortest augmenting paths, the time complexity of the algorithm remains the same. Although this modification removes the necessity of a CRS, it may also increase the runtime slightly. This is because we may now visit some vertices and use some edges in the graph through which we cannot reach an unmatched row. Therefore, if storage is the main issue, we can use this proposed modification to reduce the storage requirements of HK. Since the performance of the algorithms is our main concern, we do not report on the version of HK optimized (for memory) in this way.

3.3.2. HKDW: Duff-Wiberg Variant of HK. This variant of HK [Duff and Wiberg 1988] adds a third part to a phase of HK. After augmenting the current matching with a maximal set of shortest augmenting paths, a restricted DFS is run from all unmatched rows to find more augmenting paths.

Duff and Wiberg [1988] observed that, in HK, the time spent for the combined BFS in the first part is much more than the time spent for DFSs in the second part. To solve this problem, they proposed increasing the number of augmentations in each phase by using extra DFSs from the remaining unmatched rows. Similar to HK, the algorithm HKDW starts with a combined BFS up to the lowest level L containing an unmatched row. After that, it first initiates DFSs from unmatched level L rows that only use edges in the auxiliary graph constructed by the combined BFS. Then, in the additional third part, more DFSs are initiated from other unmatched rows. These additional DFSs are not restricted, and they can use all the edges in the original graph (not only those in the auxiliary graph). However, they obey the restriction that a vertex cannot be visited twice in a phase. To achieve this, we use the same *visited* array and continue to mark the visited columns during these later DFSs. Hence, the augmenting paths found by

additional DFSs are still disjoint. This corresponds to the modification C4 by Duff and Wiberg [1988], and it will be denoted as HKDW in this article.

3.3.3. ABMP: Alt et al.'s Algorithm. This algorithm by Alt et al. [1991] incorporates some techniques used in solving maximum flow problems [Goldberg and Tarjan 1988] into the Hopcroft and Karp's algorithm. It runs in two consecutive stages. In the first stage, a set of augmentations are performed using a sophisticated search procedure that combines BFS and DFS. In the second stage, the algorithm calls HK to perform the remaining augmentations. The key to this algorithm is the search procedure of the first stage. This procedure performs augmentations (which are found by searches from unmatched columns) while maintaining a lower bound on the length of an alternating path from an unmatched row to each vertex.

In the first stage, ABMP combines the BFS and DFS algorithms to increase the size of the matching and to assign an attribute, called level, to each vertex. The level of a vertex is slightly different from the level attribute used in HK. For each vertex v with level ℓ in ABMP, the length of the alternating paths from any unmatched row to v is larger than or equal to ℓ . Hence, the level of a vertex v in ABMP is a lower bound on the length of a shortest alternating path from an unmatched row to v .

At the beginning of this stage, the levels of all rows and columns are set 0 and 1, respectively. During the course of this stage, all rows have even level attributes; all columns have odd level attributes; and all the unmatched columns are in levels L and $L + 2$ where $L \geq 1$ is an integer increased during this stage. After initializing the levels, DFSs are initiated from unmatched columns in level L . These DFSs use the level information such that after a level ℓ vertex v , only the adjacent vertices in level $\ell - 1$ are allowed in the current search for an augmenting path. If all such adjacent vertices are visited and v comes out of the stack before an augmenting path is found, its level is updated and increased by 2. In theory, the first stage continues until the minimum level of an unmatched column is larger than $\sqrt{\tau} \log n/n$.

The second stage of the ABMP performs HK as described in Section 3.3.1. In other words, ABMP performs augmentations with a DFS while maintaining a dynamic level information until a lower bound on the shortest augmenting path length is reached. It then switches to HK to obtain a maximum matching.

Alt et al. [1991] proved that maintaining the level information dynamically up to level $L = \sqrt{\tau} \log n/n$ is cheaper than the BFS plus DFS approach of HK in terms of time complexity. With this bound on L , for dense graphs, the time complexity of ABMP becomes $\mathcal{O}(\min(\sqrt{n\tau}, n^{1.5}\sqrt{\tau/\log n}))$. In our implementation, as suggested by Mehlhorn and Näher [1999], the first stage continues until $L > 0.1\sqrt{n}$ or $50L$ is greater than the number of unmatched columns.

Note that the initial levels in the first stage are exact if the algorithm starts with an empty matching. That is, assuming there is no isolated vertex, the length of the shortest alternating path from an unmatched row to each row and column is 0 and 1, respectively. However, after a good jump-start routine, using 0 and 1 makes the algorithm slow since, for the DFSs from an unmatched column c , the search will be unsuccessful until the level attribute of c is equal to the shortest augmenting path length for c . Note that during the course of the first stage, the level attributes are not exact, that is, they are only lower bounds. Hence, the DFSs at the beginning, which use inaccurate level attributes, are always unsuccessful. However, these unsuccessful DFSs are necessary to update the level attributes. Such an update scheme may be time consuming when the difference between the lower bounds and exact values are large. To avoid this problem, as suggested by Setubal [1996], we periodically execute a global update procedure that makes the lower bounds exact. This global update procedure

is similar to the combined BFS part of HK; however, it does not stop after the first level containing an unmatched column but continues until the level L described in the previous paragraph. The global update procedure is run at the beginning once and then rerun when the total number of level updates is n since the last run.

Note that, in practice, any algorithm described in this report can be used in the second stage of ABMP to obtain the maximum cardinality matching. As mentioned previously, Alt et al. [1991] proposed to use HK to obtain a better overall complexity. Mehlhorn and Näher [1999] suggest using BFSB in the second stage. In our experiments, in addition to the original version by Alt et al. [1991], we follow the decision in Mehlhorn and Näher [1999] and implement another version called ABMP-BFS that uses BFSB in the second stage.

In addition to the space used to store the matrix structure and matching information, the first stage of ABMP with the global update scheme can be implemented by using $2n + 2m$ additional space for the arrays *queue* (size m), *visited* (size m), *lastrow* (size n), and *levels* (size n). For ABMP, we need to access the adjacent rows of the columns and the adjacent columns of the rows. Hence, we need to store the matrix in both CCS and CRS formats.

3.4. Other Approaches

There are other approaches for the maximum matching problem for bipartite graphs. We do not implement these algorithms but briefly discuss them for completeness. The main reason for not implementing them is that, for practical cases that we are interested in, the time complexity is not improved theoretically (with respect to $\mathcal{O}(\sqrt{n}\tau)$ of HK), and the algorithms are not practical to compete with the implementation of HK discussed previously.

For example, Balinski and González [1991] proposed an $\mathcal{O}(n\tau)$ -time algorithm based on a special structure known as a *strong spanning tree* (first defined by Balinski [1986] as a spanning tree rooted at a row vertex that has every column vertex of degree one as a child of the root). González and Landaeta [1995] showed that the approach originating from the concept of strong spanning trees is related to the notion of augmenting paths. By using this relation and the ideas in Hopcroft and Karp [1973], González and Landaeta [1995] proposed an $\mathcal{O}(\sqrt{n}\tau)$ algorithm for the maximum matching problem.

A randomized matching algorithm with $\mathcal{O}(n^w)$ time complexity, where w is the exponent of the best matrix multiplication algorithm is proposed by Mucha and Sankowski [2004]. This matching algorithm generates a random adjacency matrix $A(G)$ from G with random nonzero entries. After that, the inverse of the matrix $A^{-1}(G)$ is computed. Then, by using Gaussian elimination, a matching is obtained, and the matching is shown to be a maximum one with a high probability. Note that $w > 2$, and this approach is not suitable for large sparse graphs on the grounds of memory requirements and numerical difficulties that would arise.

There are randomized algorithms and probabilistic analyzes of the maximum matching algorithms. Motwani [1994] analyzes the HK algorithm on random graphs (on bipartite and also on standard undirected graphs) in order to characterize the average case behavior of HK. It is proved that, with high probability, there exists an augmenting path of length $\mathcal{O}(\log n)$ when the average degree is $\ln(n)$. Hence, the HK algorithm works, on the average, in $\mathcal{O}(\tau \log n)$ time. Later, Bast et al. [2006] improve this result where the average degree is a large enough constant bigger than or equal to 8.83. The main algorithmic tool is the construction of two alternating trees from the end vertices of an augmenting path; one from a row vertex and another from a column vertex. It has been shown that these trees will meet if their lengths are of size $\mathcal{O}(\log n)$. Chebolu et al. [2010] take the output of the jump-start routine KSM (discussed in the next section), try

ALGORITHM 1: Simple Greedy Matching (SGM)**Data:** A bipartite graph $G = (V_R \cup V_C, E)$ **Result:** A matching \mathcal{M} $\mathcal{M} \leftarrow \emptyset;$ **for** each unmatched $u \in V_C$ **do** **if** u is adjacent to an unmatched vertex $v \in V_R$ **then** $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\};$ $G \leftarrow G \setminus \{u, v\}$ \blacktriangleright remove vertices u, v , and all edges incident on them **end****end**return $\mathcal{M};$

to augment it by searching paths only between a subset of the unmatched vertices, and then use the algorithm of Bast et al. [2006] if these searches have not found a maximum matching. It has been shown that the overall algorithm runs in linear expected time for sparse random graphs.

Another technique is to transform the input graph such that the algorithms would run fast on it. This is exemplified in Feder and Motwani [1995], where the runtime complexity of HK becomes $\mathcal{O}(\tau^* \sqrt{n})$ -time where $\tau^* = \tau \log(n^2/\tau)/\log n$. Notice that the runtime is $\mathcal{O}(\tau \sqrt{n})$ for practical purposes (when the graphs are sparse), and hence there is no improvement over the standard HK.

A remarkable property of algorithm HK is that it leads to an approximation algorithm for the maximum matching problem. One can use the lemmas and proofs by Hopcroft and Karp [1973] to obtain an $\mathcal{O}(p\tau)$ -time algorithm that finds a matching with cardinality at least $p/(p+1)\delta_{max}$, where δ_{max} is the cardinality of a maximum matching. Let δ be the size of the matching after the p th phase of HK. Since the shortest path length increases by at least 2 in each phase, the shortest augmenting path length after the p th phase is at least $2p+1$. Then, each augmenting path should contain at least p matching edges. Thus, there are at most δ/p vertex disjoint augmenting paths. Since each augmenting path increases the cardinality of the current matching by one, δ_{max} can be at most $\delta + \delta/p$. That is, $(p+1)/p\delta \geq \delta_{max}$. One can rearrange the inequalities as $\delta \geq (1 - (1/(p+1)))\delta_{max}$ to see the approximation at the end of p th phase. This remark is also made by Gabow and Tarjan [1988, p. 415].

4. SOME HEURISTICS FOR THE MAXIMUM MATCHING PROBLEM

Several fast heuristics to jump-start maximum cardinality matching algorithms have been described in the literature [Karp and Sipser 1981; Langguth et al. 2010; Magun 1998; Pothen and Fan 1990; Setubal 1996]. Here we discuss three of these heuristics. The first one, although it is the worst one, is probably the most frequently used heuristic in practice. The second one is probably the most theoretically studied heuristic for the maximum cardinality matching problem. Surprisingly, according to our experiments, the last one is the best. These heuristics are experimentally investigated by Magun [1998] and Langguth et al. [2010]. We also briefly investigate these heuristics to see their effects on the runtime of the algorithms. Modified and extended versions of these heuristics and experimental analysis can be found in Magun [1998] and Langguth et al. [2010].

4.1. Simple Greedy Matching (SGM)

The first heuristic, Algorithm 1, is used by Pothen and Fan [1990] and Duff and Wiberg [1988]. In this simple heuristic, each unmatched column is examined in turn and matched with the first unmatched row in its adjacency list, if there is any.

Clearly, the time complexity of the algorithm is $\mathcal{O}(n + \tau)$. This type of fast heuristic is very popular and is used by many software packages, for example, LEDA [Mehlhorn and Näher 1999], and MC21A in HSL. Also it is frequently used in previous papers that compare the performance of matching heuristics [Setubal 1992; 1996]. A similar greedy matching heuristic, which iteratively chooses a random edge with two unmatched endpoints, is described by Dyer et al. [1993], Korte and Hausmann [1978], and Magun [1998]. Both heuristics obtain matchings with cardinality at least $\delta_{max}/2$ where δ_{max} is the cardinality of a maximum matching. Detailed mathematical and experimental analyses of the latter heuristic for general undirected graphs can be found in Dyer et al. [1993] and Magun [1998].

4.2. Karp-Sipser Greedy Matching (KSM)

The second greedy heuristic is proposed by Karp and Sipser [1981]. Their key observation is that when a graph $G = (V, E)$ has vertices of degree 1 and 2, the maximum cardinality matching problem on G can be reduced to the maximum matching problem on a smaller graph constructed using the following two rules.

Degree-1 reduction. If a vertex u has only one adjacent vertex v , then there is a maximum matching \mathcal{M} containing (u, v) . In fact, let $G' = G \setminus \{u, v\}$, be the reduced graph with the vertices u and v and all adjacent edges removed. Now, if \mathcal{M}' is a maximum matching for G' , then $\mathcal{M} = \mathcal{M}' \cup \{(u, v)\}$ is a maximum matching for G .

Degree-2 reduction. Let u be a vertex with degree 2, and v and w be the adjacent vertices. Let $G' = (G \setminus \{u\}) * \{v, w\}$ where $*\{v, w\}$ denotes the *merging* of the vertices v and w . That is, these vertices are removed from the graph, and a new vertex z is inserted with the edges (z, y) such that $y \neq v, w$ and either (v, y) or (w, y) , or both were in the graph $G \setminus \{u\}$. Let z be the merged vertex and \mathcal{M}' be a maximum matching for G' . A maximum matching \mathcal{M} for G can be obtained for the following cases:

- z is unmatched in \mathcal{M}' : $\mathcal{M} = \mathcal{M}' \cup \{(u, v)\}$;
- $(z, y) \in \mathcal{M}'$ where $(v, y) \in E$: $\mathcal{M} = \mathcal{M}' \setminus \{(z, y)\} \cup \{(u, w), (v, y)\}$;
- $(z, y) \in \mathcal{M}'$ where $(v, y) \notin E$: $\mathcal{M} = \mathcal{M}' \setminus \{(z, y)\} \cup \{(u, v), (w, y)\}$.

By using these two reductions, Karp and Sipser proposed a cheap heuristic for the maximum matching cardinality problem. When one of these reductions is possible, the graph can be reduced immediately. Otherwise, a vertex u with the maximum degree is chosen randomly, and it is matched to a random adjacent unmatched vertex v , and the graph is again reduced as $G' \leftarrow G \setminus \{u, v\}$. Clearly, the two reductions apply to bipartite graphs as well.

Karp and Sipser stated that this algorithm is hard to analyze and proposed a simpler algorithm given below as Algorithm 2. This algorithm does the degree-1 reductions and then resorts to random choices when there are no degree-1 vertices remaining. This version, referred to as KSM, is called the Karp-Sipser heuristic in the literature, and its performance is analysed by Aronson et al. [1998] and Chebolu et al. [2010]. In our implementation of Algorithm 2, we maintain the degrees dynamically, that is, when a matching (u, v) occurs, we decrease the degrees of vertices adjacent to u or v by 1.

In the KSM heuristic, if the **if** statement is always true the matching decision is optimal, that is, there exists a maximum cardinality matching in the current graph G that contains (u, v) . Hence, if all of the matching edges are chosen inside the **if** statement, KSM finds the maximum cardinality matching. Furthermore, if the matching is perfect and unique, KSM finds it, and all of the decisions are made inside the **if** statement. On the other hand, because of edges put into the \mathcal{M} inside the **else** part, the algorithm may not be able to compute a maximum cardinality matching. However,

ALGORITHM 2: Karp-Sipser Heuristic (KSM)**Data:** A bipartite graph $G = (V_R \cup V_C, E)$ **Result:** A matching \mathcal{M} $\mathcal{M} \leftarrow \emptyset;$ **while** E is not empty **do** **if** a degree-1 vertex u adjacent to vertex v exists **then** | choose the edge $(u, v);$ **else** | get a random edge $(u, v) \in E;$ **end** $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\};$ $G \leftarrow G \setminus \{u, v\}$ ▶ remove vertices u, v , and all edges incident on them**end**return $\mathcal{M};$

Karp and Sipser proved that a random graph with no vertices of degree 0 or 1 tends to have near-perfect matchings, and Algorithm 2 tends to find one of these matchings.

The performance of KSM-type greedy heuristics (with both degree 1 and 2 reductions) are analyzed by Magun [1998]. The experimental results show that the deficiencies of the heuristics are very small. For example, in random graphs where each possible vertex pair is included as an edge with equal probability, when the number of vertices is 10000 and the average degree is 3, the average deficiency of KSM is less than one. More detailed results can be found in Magun [1998]. We also direct the reader to Langguth et al. [2010] for a more recent and comprehensive analysis of the heuristics and their effects on some of the algorithms described in this article.

4.3. Minimum Degree Matching (MDM)

The minimum degree matching heuristic first chooses an unmatched vertex u with the minimum degree [Magun 1998]. The one-sided version chooses a random unmatched vertex v in the adjacency list of u , whereas the two-sided version chooses an unmatched vertex v in the adjacency list of u with the minimum degree. Then, the vertices u and v and all of the edges incident to them are removed from the graph. The process continues until E is empty.

In Langguth et al. [2010], it is stated that the two-sided version is better than the one-sided one. Taking this result into consideration, we implemented the two-sided version. The two-sided minimum degree matching heuristic shown in Algorithm 3 uses a priority queue implemented with buckets and has a runtime complexity of $\mathcal{O}(n + \tau)$. In the implementation, for each degree $i = \{1, 2, \dots, n\}$, we use a bucket to store the vertices with degree i . During the course of the heuristic, when the degree of a vertex is reduced, we remove it from its current bucket (corresponding to its current degree), and we put it into the previous bucket (degree minus one). To obtain $\mathcal{O}(1)$ time for each update, we keep an array that stores the location of each vertex in the combined list of buckets. Since there are τ edges, there will be at most τ updates and the complexity follows.

4.4. Comments on the Heuristics

In Algorithm 2, we need to detect the existence of both degree-1 rows and columns and need to find the unmatched vertices in their adjacency lists. In Algorithm 3, the vertex with the minimum degree can be either a row or a column. Hence, an efficient implementation of these two heuristics requires the storage of the matrix and its transpose. Hence, if memory is the main issue, SGM is preferable. However, as will be

ALGORITHM 3: Minimum Degree Matching Heuristic (MDM)**Data:** A bipartite graph $G = (V_R \cup V_C, E)$ **Result:** A matching \mathcal{M} $\mathcal{M} \leftarrow \emptyset;$ **while** E is not empty **do** choose a vertex u with the minimum positive degree; choose a vertex v with the minimum degree among those adjacent to u ; $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\};$ $G \leftarrow G \setminus \{u, v\}$ ▶ remove vertices u, v , and all edges incident on them**end**return \mathcal{M} ;

shown in Section 5, using KSM or MDM is much better than using SGM as a jump-start routine in terms of the execution times of the matching algorithms. Hence, for our experiments, it is no longer important whether the matching algorithms require one of CCS or CRS, or both. However, this may not be the case where the available memory is only sufficient for one of them.

Note that the degree-1 treatment of KSM is automatic in MDM, and hence we can argue that MDM is an extended version of KSM. According to our experience, MDM's performance is better than KSM in terms of reducing the number of unmatched vertices. There are some rare cases where reducing the deficiency by the jump-start routine does not lead to any reduction in the overall running time.

5. EXPERIMENTS

Here we give experimental results comparing the performance of the algorithms described in this article. We implemented the algorithms in C programming language and compiled them with gcc version 3.4.5. We test their performance with random square matrices and matrices from real life problems. All of the experiments were conducted on a 2.4-Ghz Dual Core computer, equipped with 4-GB RAM and Windows Vista.

5.1. Data Set

We adapt two random bipartite graph generators to create a set of square random matrices.

- (1) RBG-U(n', d). Let $n = n' \times 10^5$ be the size of the matrix, and d be the desired number of nonzeros in a row/column on average. We uniformly choose $n \times d$ nonzeros. Therefore, each n^2 potential nonzero is present in a random matrix with the same probability. This generator, which is based on the random graph model of Erdős and Rényi [1959], is used by Magun [1998] and behaves like the sprand command of Matlab.
- (2) RBG-B(n', k, d). Let $n = n' \times 10^5$ be the size of the matrix, and d be the desired number of nonzeros in a row/column on average. Assume that the set of rows and columns are divided into k groups of equal size. To create the nonzeros in a column in the i th group, we first randomly choose the number of nonzeros in column c , denoted by $deg(c)$, from a binomial distribution with mean d . We then uniformly choose $deg(c)$ random rows from the $(i - 1)$ th through $(i + 1)$ th groups of rows and add the corresponding nonzeros. This generator is also used by Langguth et al. [2010], Cherkassky et al. [1998] and Setubal [1996].

After creating the random graphs, we permute their vertices and store the corresponding matrix in CSR and/or CSC formats. For convenience, the row indices of

Table II. Test Matrices from UFL Sparse Matrix Collection Used in the Experiments.

Group	Name	Group	Name	Group	Name
AMD	<i>G2_circuit</i>	JGD.GL7d	<i>GL7d15</i>	Ronis	<i>xenon2</i>
Andrianov	<i>ins2</i>		<i>GL7d23</i>	Rothberg	<i>cf2</i>
Andrianov	<i>lp1</i>	JGD.Homology	<i>ch7-9-b4</i>	Sandia	<i>ASIC_320k</i>
ATandT	<i>pre2</i>		<i>ch7-9-b5</i>		<i>ASIC_320ks</i>
	<i>twotone</i>		<i>ch8-8-b4</i>		<i>ASIC_680k</i>
Barabasi	<i>NotreDame_actors</i>		<i>ch8-8-b5</i>	Sch..IBMNA	<i>c-73</i>
	<i>NotreDame_www</i>		<i>n4c6-b8</i>		<i>c-big</i>
Botonakis	<i>FEM_3D_thermal2</i>		<i>n4c6-b9</i>		<i>c-73b</i>
	<i>thermomech_TC</i>	JGD.Margulies	<i>kneser_10_4_1</i>	Sch..IBMSDS	<i>matrix-new_3</i>
	<i>thermomech_dM</i>	Kamvar	<i>Stanford</i>		<i>barrier2-4</i>
	<i>thermomech_dK</i>	Lin	<i>Lin</i>	Sch..ISEI	<i>barrier2-9</i>
CEMW	<i>t2em</i>	McRae	<i>ecology1</i>		<i>ohne2</i>
	<i>tmt_unsym</i>		<i>ecology2</i>		<i>para-4</i>
	<i>tmt_sym</i>	Mittelmann	<i>sgp75y6</i>		<i>web-Google</i>
DNVS	<i>ship_003</i>		<i>stormG2_1000</i>	SNAP	<i>amazon0312</i>
	<i>shipsec1</i>		<i>watson_1</i>		<i>amazon0505</i>
	<i>shipsec5</i>		<i>watson_2</i>		<i>amazon0601</i>
	<i>shipsec8</i>		<i>neos</i>		<i>roadNet-PA</i>
FreeFieldTech	<i>mono_500Hz</i>		<i>neos3</i>		<i>roadNet-TX</i>
Freescale	<i>transient</i>	Norris	<i>lung2</i>	TKK	<i>engine</i>
GHS_indef	<i>d_pretok</i>		<i>stomach</i>	Tromble	<i>language</i>
	<i>darcy003</i>		<i>torso2</i>	TSOPF	<i>FS_b39_c30</i>
	<i>helm2d03</i>		<i>torso3</i>	Um	<i>2cubes_sphere</i>
	<i>mario002</i>	Oberwolfach	<i>filter3D</i>		<i>offshore</i>
	<i>turon_m</i>		<i>boneS01</i>	UTEP	<i>Dubcova3</i>
	<i>boyd2</i>	Pajek	<i>IMDB</i>	vanHeukelum	<i>cage12</i>
GHS_psdef	<i>ford2</i>	PARSEC	<i>Ga10As10H30</i>	Wissgott	<i>parabolic_fem</i>
	<i>apache2</i>	QLi	<i>crashbasis</i>	Watson	<i>Baumann</i>
Hamm	<i>hcircuit</i>		<i>largebasis</i>	Yoshiyasu	<i>image_interp</i>
	<i>scircuit</i>	Rajat	<i>Raj1</i>		
Hamrle	<i>Hamrle3</i>		<i>rajat21</i>		
HVDC	<i>hvdc2</i>		<i>rajat23</i>		
IBM.EDA	<i>dc1</i>		<i>rajat24</i>		
	<i>trans4</i>		<i>rajat29</i>		
JGD.Forest	<i>TF19</i>		<i>rajat30</i>		

nonzeros in a column are stored in increasing order in *rids*, and the same is done for the column indices stored in *cids*.

We use some real-life matrices from the University of Florida Sparse Matrix Collection (<http://www.cise.ufl.edu/research/sparse/matrices/>). The matrices satisfy the following properties: $10^5 \leq \min(m, n) \leq 15 \times 10^5$ and furthermore $2.5 \times \max(m, n) \leq \tau \leq 7 \times 10^6$. In order to avoid bias or skew in the data set, we choose at most six matrices from each matrix group (there were a total of 160 groups in the collection at the time of writing). When there are more than six matrices satisfying these properties in a group, we choose those with the largest number of nonzeros. There were a total of 107 matrices satisfying these assertions at the time of writing, where 19 of them are rectangular which are transposed, if necessary, to have $m > n$. Some of them have the same pattern, and we remove the duplicates from our test set. The names of the matrices and their groups are given in Table II.

5.2. Performances of the Greedy Heuristics

We implemented the heuristics given in Algorithms 1, 2, and 3. The first heuristic, SGM, was previously used with some of the matching algorithms described in this article [Duff and Wiberg 1988; Langguth et al. 2010; Pothen and Fan 1990; Setubal 1996]. The KSM heuristic was experimented and analyzed extensively on its own and used as a part of some randomized algorithms [Chebolu et al. 2010; Karp and Sipser 1981]. The

Table III. Execution Times and the Number of Augmentations Performed by the Algorithm BFSB when Combined with SGM, KSM, and MDM for Matrices Generated by the RBG-U and RBG-B Generators. For each parameter set, the first row shows the execution times of BFSB in seconds, whereas the second row shows the deficiency of each heuristic. Each entry in the table is an average for 10 random matrices

Matrix	SGM	KSM	MDM	Matrix	SGM	KSM	MDM
RBG-U(1, 3)	0.2 15019	0.1 1	0.1 0	RBG-U(7, 3)	26.1 105244	1.1 9	1.4 10
RBG-U(1, 5)	0.3 13027	0.1 4	0.1 2	RBG-U(7, 5)	15.8 91180	1.1 23	1.2 8
RBG-U(1, 10)	0.1 6906	0.1 4	0.2 0	RBG-U(7, 10)	3.7 48547	1.5 21	1.9 1
RBG-U(1, 15)	0.1 4594	0.1 4	0.2 0	RBG-U(7, 15)	2.0 32327	2.1 25	2.5 0
RBG-U(3, 3)	4.4 45055	0.3 3	0.4 4	RBG-U(9, 3)	36.1 135279	1.4 14	1.8 17
RBG-U(3, 5)	3.5 39229	0.4 5	0.5 2	RBG-U(9, 5)	16.3 117488	1.3 28	1.6 10
RBG-U(3, 10)	0.8 20806	0.5 10	0.7 1	RBG-U(9, 10)	4.2 62328	1.9 39	2.3 1
RBG-U(3, 15)	0.5 13853	0.6 10	1.0 1	RBG-U(9, 15)	2.5 41521	2.5 43	3.0 0
RBG-B(5, 1, 3)	11.8 75129	0.5 3	0.6 1	RBG-B(5, 100, 3)	6.7 75022	1.3 75	1.4 76
RBG-B(5, 1, 5)	11.0 65128	0.6 2	0.9 1	RBG-B(5, 100, 5)	7.9 65245	2.7 104	1.0 4
RBG-B(5, 1, 10)	2.1 34655	1.0 5	1.4 0	RBG-B(5, 100, 10)	2.8 34631	2.5 97	1.1 0
RBG-B(5, 1, 15)	1.1 23081	1.1 7	1.8 0	RBG-B(5, 100, 15)	3.2 23121	3.4 101	1.4 0
RBG-B(5, 40, 3)	7.1 75018	0.5 18	0.7 17	RBG-B(5, 200, 3)	6.5 74972	2.8 204	2.9 192
RBG-B(5, 40, 5)	5.8 65219	0.8 20	1.0 2	RBG-B(5, 200, 5)	5.2 65273	2.7 182	1.0 6
RBG-B(5, 40, 10)	3.5 34730	2.3 57	1.2 0	RBG-B(5, 200, 10)	3.2 34666	2.8 181	1.0 0
RBG-B(5, 40, 15)	2.2 23053	2.3 44	1.7 0	RBG-B(5, 200, 15)	3.1 23085	3.4 184	1.3 0

MDM heuristic is one of the best, but its effect on the matching algorithms have not been thoroughly investigated.

Table III shows the performance of the heuristics within BFSB on RBG-U and RBG-B graphs. We conducted the same tests with other algorithms and found that there is little difference in the relative performance of the heuristics. Therefore, we did not put them into the table. In addition to the execution times, similar to previous papers [Langguth et al. 2010; Magun 1998] we give for each heuristic its deficiency, that is, the difference between the cardinalities of a maximum matching and the matching obtained by the heuristic. Hence, the smaller the deficiency, the better the heuristic. Since the SGM heuristic uses more arbitrary decisions, it is expected that KSM and MDM will perform better than SGM. As the experiments show, the difference between the performance of SGM and the other heuristics is huge.

Although KSM or MDM have good performance and runtime complexity, they are costlier than SGM. However, as Table III shows, the overheads of KSM and MDM are insignificant when compared to the reductions in the total execution time of the matching algorithm. For RBG-U graphs in the table, the heuristics SGM, KSM, and MDM take 18%, 80%, and 90% of the total execution time, respectively. For RBG-B graphs, these numbers are 3%, 47%, and 81%.

As the table shows, MDM is a better heuristic than KSM in terms of deficiency. This is expected, since MDM is an enhanced version of KSM. For graphs containing only one group, such as those generated by the RBG-U generator and those generated by the RBG-B generator with $k = 1$, KSM performs slightly worse than MDM in terms of deficiency. However, for these graphs, the execution time of BFSB using KSM is less, since KSM is also cheaper than MDM. This is also true when RBG-B graphs are created with the parameter $d = 3$. Hence, we can argue that KSM performs well when there is only one group or the average degree is low, and therefore KSM may be preferable for these cases. On the other hand, MDM performs much better than KSM for RBG-B graphs with $k > 1$.

Although MDM would be the method of choice for jump-starting the matching algorithms, we shall use KSM and MDM to assess the relative performance of the matching algorithms. This is for two reasons. First, as seen in Table III, KSM and MDM are considerably better than SGM as a jump-start routine. This is in agreement with previously reported studies [Langguth et al. 2010; Magun 1998]. Second, as also observed by Duff [1981, p.111] and Mehlhorn and Näher [1999], using SGM and not using it, in general, are not much different for the shortest augmenting path based algorithms; in fact, the first phase of HK where one finds augmenting paths of length one corresponds to SGM. On the other hand, there is no comprehensive study investigating the effects of KSM and MDM.

5.3. Performance of the Algorithms

We compare the eight algorithms listed in Table I and discussed in Section 3. For random instances with different parameters, we run each algorithm ten times and store the average execution time of each algorithm for each parameter set. For real-life matrices, in order to reduce the measurement errors (in some cases, the algorithms run in less than a second), we repeat the same experiment ten times and store the average execution time. The relative performance of each algorithm on a particular problem is computed by dividing the average execution time of the algorithm by the best average time for the same problem. Then, we display these average relative values in the tables.

5.3.1. Change of the Jump-Start Routine. Previous comparisons, which use SGM as the jump-start routine, suggest using BFS instead of DFS while searching for augmenting paths [Mehlhorn and Näher 1999; Setubal 1996]. Furthermore, previous research states that ABMP performs much better than BFSB in most cases [Cherkassky et al. 1998; Setubal 1996]. In this section, we will show that changing the jump-start routine affects the relative performance of the algorithms. In addition, we will see the improvement of MC21A over DFSB thanks to the use of the lookahead technique.

DFS vs. BFS. As stated in Section 3.1, a search in BFSB always finds one of the shortest augmenting paths in the graph for the corresponding unmatched column. On the other hand, DFSB may visit a large portion of the graph even when there is an unmatched row in the adjacency list of the unmatched column from which the DFS has been started. Employing a lookahead scheme, MC21A alleviates this problem for all trivial paths. However, it may not solve the cases when the shortest paths are longer. Hence, as also stated by Mehlhorn and Näher [1999] and Setubal [1996], we may argue that BFSB will find augmenting paths more quickly than DFSB.

Table IV shows the results of the experiments conducted to compare the algorithms DFSB, BFSB, and MC21A for a class of random graphs generated by the RBG-U generator. When the algorithms are used with SGM, BFSB is clearly the best algorithm. It is 4 to 35 times better than MC21A even with the significant effect of the lookahead scheme that reduces the runtime of the DFSB from almost 3 hours to 69.3 seconds for some of the cases. As seen, the impact of the lookahead increases when the number of edges in the graph increases. This is expected since, when a column is visited during the search,

Table IV. Comparison of DFS- and BFS-Based Algorithms, DFSB, BFSB, and MC21A using SGM and KSM as the jump-start routine. The execution times of the algorithms are given in seconds. Each entry is an average for 10 random matrices

Matrix	SGM			KSM		
	DFSB	BFSB	MC21A	DFSB	BFSB	MC21A
RBG-U(1, 3)	6.9	0.2	2.8	0.1	0.1	0.1
RBG-U(1, 5)	23.6	0.3	3.0	0.1	0.1	0.1
RBG-U(1, 10)	41.9	0.1	0.9	0.1	0.1	0.1
RBG-U(1, 15)	46.4	0.1	0.4	0.1	0.1	0.1
RBG-U(3, 3)	187.4	4.4	64.0	0.3	0.3	0.3
RBG-U(3, 5)	464.6	3.5	49.2	0.4	0.4	0.4
RBG-U(3, 10)	721.7	0.8	14.2	0.7	0.5	0.5
RBG-U(3, 15)	736.2	0.5	6.0	0.9	0.6	0.6
RBG-U(5, 3)	585.3	11.3	193.0	0.6	0.5	0.6
RBG-U(5, 5)	1442.5	8.7	148.5	1.1	0.7	0.9
RBG-U(5, 10)	2378.8	2.0	43.3	2.3	1.0	1.2
RBG-U(5, 15)	2486.2	1.2	19.7	3.2	1.2	1.4
RBG-U(7, 3)	1167.1	26.1	397.7	1.4	1.1	1.4
RBG-U(7, 5)	2956.2	15.8	301.1	2.5	1.1	1.3
RBG-U(7, 10)	5135.3	3.7	91.5	5.4	1.5	1.9
RBG-U(7, 15)	5396.0	2.0	41.7	9.1	2.1	2.7
RBG-U(9, 3)	1971.8	36.1	652.3	2.3	1.4	1.8
RBG-U(9, 5)	5019.9	16.3	485.0	3.6	1.3	1.6
RBG-U(9, 10)	8972.7	4.2	152.8	11.4	1.9	2.0
RBG-U(9, 15)	9667.1	2.5	69.3	17.3	2.5	2.8
Avg. Relative Perf.	885.4	1.0	87.0	2.3	1.0	1.1

the lookahead scheme has a higher chance to find an unmatched row if the degree of the visited column is higher.

The difference between the left half and the right half of the table is a change in the jump-start routine. For relatively smaller random graphs with $n \leq 3 \times 10^5$, we do not observe a difference in performance, since the KSM heuristic leaves only 3 to 10 vertices to be matched by the algorithms later. Note that this number is between 13000 and 45000 for the SGM heuristic. However, even for larger matrices, when KSM is used, the execution times of BFSB and MC21A are close because, in this case, the shortest path lengths increase and BFSB spends more time per BFS search, especially when the average degree is high (in which case, lookahead performs better). However, as the table shows, BFSB is still better than all simple DFS-based alternatives in terms of execution time for these sets of random graphs. Although we still keep MC21A, due to the bad performance of DFSB, we will not consider it as a candidate for the rest of the experiments.

Different Versions of ABMP. As described in Section 3.3.3, ABMP is a two-stage algorithm that uses the algorithm HK in the second stage. Although any algorithm described in this article can be used in the second stage, following the decision by Mehlhorn and Näher [1999, p. 106], we also implemented the version ABMP-BFS, which uses BFSB in the second stage of the algorithm. Table V shows the execution times of the algorithms ABMP, ABMP-BFS and BFSB.

As Table V shows, when MDM is used as the jump-start routine in the preprocessing step, the performance of the algorithms gets closer to each other since most of the work is done by the heuristic. For this case, ABMP-BFS is better than ABMP only for graphs containing many groups and having small d . On the other hand, when KSM is used, ABMP-BFS is better than the original version in general, especially when the graphs created by the RBG-B generator contains many groups. Hence, in the following experiments, we will use ABMP-BFS instead of ABMP.

Table V. Execution Times of Algorithms ABMP, ABMP-BFS, and BFSB in Seconds Combined with KSM and MDM. Each entry in the table is an average for 10 random matrices

Matrix	KSM			MDM		
	ABMP	ABMP-BFS	BFSB	ABMP	ABMP-BFS	BFSB
RBG-U(10, 3)	1.4	1.3	1.4	1.7	1.7	1.7
RBG-U(10, 5)	1.7	1.7	2.3	2.2	2.2	2.3
RBG-U(10, 10)	3.0	2.7	2.3	3.4	3.0	2.8
RBG-U(15, 3)	2.6	2.5	2.5	3.4	3.4	4.1
RBG-U(15, 5)	3.4	3.1	3.0	3.8	3.7	3.2
RBG-U(15, 10)	4.7	3.8	4.6	4.5	4.5	4.7
RBG-U(20, 3)	3.3	3.4	7.3	3.8	3.8	8.1
RBG-U(20, 5)	4.2	4.3	4.6	5.1	5.0	4.6
RBG-U(20, 10)	6.8	5.5	6.6	6.1	6.1	6.2
Avg. Relative Perf.	1.1	1.0	1.2	1.1	1.0	1.2
RBG-B(10, 1, 3)	1.2	1.2	1.1	1.6	1.6	1.4
RBG-B(10, 1, 5)	1.7	1.7	1.6	2.1	2.1	2.0
RBG-B(10, 1, 10)	2.2	2.0	2.5	3.1	3.0	3.3
RBG-B(10, 1, 15)	3.0	2.7	2.5	4.6	4.3	4.0
RBG-B(10, 40, 3)	1.4	1.2	1.4	1.6	1.6	1.8
RBG-B(10, 40, 5)	2.2	2.2	1.9	2.3	2.3	2.0
RBG-B(10, 40, 10)	4.4	3.7	4.6	2.9	2.9	2.9
RBG-B(10, 40, 15)	5.0	6.2	6.8	4.0	3.9	4.2
RBG-B(10, 100, 3)	3.4	2.6	2.9	3.5	2.6	3.0
RBG-B(10, 100, 5)	5.8	5.5	7.8	2.5	2.3	2.1
RBG-B(10, 100, 10)	7.8	6.7	7.6	2.6	2.6	2.6
RBG-B(10, 100, 15)	10.8	8.5	9.7	3.5	3.5	3.5
RBG-B(10, 200, 3)	7.1	4.8	6.6	7.2	5.1	5.7
RBG-B(10, 200, 5)	9.9	8.1	7.7	3.1	2.5	2.1
RBG-B(10, 200, 10)	11.1	7.2	7.7	2.4	2.4	2.4
RBG-B(10, 200, 15)	11.6	8.2	9.1	3.1	3.0	3.1
Avg. Relative Perf.	1.2	1.0	1.2	1.1	1.0	1.0

When KSM is used, ABMP-BFS is generally better than BFSB, which shows that the first stage of the algorithm is useful in practice. On the other hand, when MDM is used, the performances are almost equal for the graphs created by the RBG-B generator. For the graphs created by the RBG-U generator, the results are more interesting. While ABMP-BFS is much better than BFSB for $d = 3$, it is worse for $d = 5$, whereas, their performances are the same for $d = 10$. We observed that for the case $d = 5$, the first stage of ABMP-BFS takes only a small percentage of the execution time, and hence it is not the main reason for worse relative performance. Although the second stage contains fewer BFS searches than BFSB, it takes more time than BFSB due to the longer augmenting paths remaining after the first stage. This may be expected since a BFS search from a column c takes more time when the shortest augmenting path for c is longer.

As Tables IV and V show, using a good jump-start routine in the preprocessing step makes the performance of the algorithms very close to each other since most of the work is done by the heuristic. Hence, most of the previous studies in the literature on performance comparisons becomes obsolete since they use the SGM heuristic.

5.3.2. Experiments with random matrices. The results of the experiments with the graphs constructed by RBG-U and RBG-B generators are given in Tables VI and VII. In these tables, the average relative performance of each algorithm is given for each generator. These values are computed as follows. For each row in the table, we find the algorithm with the minimum execution time t . We set the relative performance of this algorithm for this row to be 1.0 and compute the relative performance of the other algorithms by dividing their execution times by t . The values in the last row are the averages of these relative performances.

Table VI. Execution Times of Algorithms in Seconds Combined with the KSM Heuristic for RBG-U and RBG-B Graphs. Each entry in the table is an average for 10 random matrices

Matrix	BFSB	MC21A	PF	PF+	HK	HKDW	ABMP-BFS
RBG-U(10, 3)	1.4	1.7	1.4	1.4	1.8	1.5	1.3
RBG-U(10, 5)	2.3	2.9	2.3	2.2	2.5	1.9	1.7
RBG-U(10, 10)	2.3	2.9	2.3	2.3	2.7	2.6	2.7
RBG-U(15, 3)	2.5	4.2	2.6	2.4	3.8	3.2	2.5
RBG-U(15, 5)	3.0	4.7	3.2	3.1	3.4	3.0	3.1
RBG-U(15, 10)	4.6	6.6	4.5	4.2	5.3	4.1	3.8
RBG-U(20, 3)	7.3	12.9	6.0	5.5	8.0	4.0	3.4
RBG-U(20, 5)	4.6	8.5	4.6	4.2	4.9	4.5	4.3
RBG-U(20, 10)	6.6	10.7	6.5	6.4	7.2	5.8	5.5
Avg. Relative Perf.	1.2	1.9	1.2	1.1	1.4	1.1	1.0
RBG-B(7, 1, 3)	1.0	0.9	0.9	0.8	0.9	0.8	0.8
RBG-B(7, 1, 5)	1.1	1.3	1.3	1.3	1.1	1.2	1.1
RBG-B(7, 1, 10)	1.4	1.5	1.7	1.7	1.6	1.7	1.5
RBG-B(7, 1, 15)	1.7	1.8	1.8	1.8	1.9	1.9	1.7
RBG-B(7, 40, 3)	0.9	1.0	0.9	1.0	1.5	1.0	0.9
RBG-B(7, 40, 5)	1.3	1.4	1.3	1.3	1.4	1.1	1.2
RBG-B(7, 40, 10)	2.5	1.8	1.8	1.8	2.8	1.6	2.6
RBG-B(7, 40, 15)	3.4	1.7	1.6	1.6	3.3	1.7	4.2
RBG-B(7, 100, 3)	2.9	2.5	1.8	1.4	4.2	2.6	2.3
RBG-B(7, 100, 5)	3.5	1.8	1.4	1.5	3.6	1.2	2.2
RBG-B(7, 100, 10)	5.0	2.1	1.7	1.7	4.4	1.7	4.1
RBG-B(7, 100, 15)	5.7	2.0	1.7	1.7	6.5	1.7	5.3
RBG-B(7, 200, 3)	3.5	4.1	2.7	2.4	7.0	4.6	3.1
RBG-B(7, 200, 5)	6.0	2.8	1.5	1.4	7.0	1.4	4.8
RBG-B(7, 200, 10)	4.4	2.2	1.7	1.7	6.3	1.7	4.4
RBG-B(7, 200, 15)	5.7	2.3	1.9	1.9	6.8	1.7	4.7
RBG-B(10, 1, 3)	1.1	1.1	1.1	1.2	1.2	1.3	1.2
RBG-B(10, 1, 5)	1.6	1.8	1.9	1.9	1.8	1.7	1.7
RBG-B(10, 1, 10)	2.5	2.9	3.4	3.4	2.8	2.5	2.0
RBG-B(10, 1, 15)	2.5	2.4	2.3	2.4	2.8	2.8	2.7
RBG-B(10, 40, 3)	1.4	1.8	1.6	1.5	2.2	1.5	1.2
RBG-B(10, 40, 5)	1.9	1.8	1.8	1.8	2.3	1.7	2.2
RBG-B(10, 40, 10)	4.6	3.2	3.2	3.2	4.3	2.5	3.7
RBG-B(10, 40, 15)	6.8	3.2	3.1	3.1	5.5	2.7	6.2
RBG-B(10, 100, 3)	2.9	2.6	1.7	1.7	5.0	2.9	2.6
RBG-B(10, 100, 5)	7.8	3.0	1.9	1.9	5.4	1.8	5.5
RBG-B(10, 100, 10)	7.6	3.3	2.9	2.9	8.9	2.5	6.7
RBG-B(10, 100, 15)	9.7	3.3	2.8	2.8	8.5	2.6	8.5
RBG-B(10, 200, 3)	6.6	7.8	4.2	3.5	11.9	6.0	4.8
RBG-B(10, 200, 5)	7.7	3.6	2.1	2.0	9.4	2.2	8.1
RBG-B(10, 200, 10)	7.7	3.6	2.5	2.5	9.4	2.4	7.2
RBG-B(10, 200, 15)	9.1	3.3	2.5	2.5	13.3	2.5	8.2
Avg. Relative Perf.	2.2	1.4	1.1	1.1	2.5	1.2	1.9

As Table VI shows, when the KSM heuristic is used, for RBG-U graphs, BFSB is better than the DFS based alternative MC21A. However, for RBG-B graphs with high group numbers, MC21A works much better than BFSB; see RBG-B(10, 100, 15) and RBG-B(10, 200, 15) in the table. This is expected because the higher the number of groups, the longer the augmenting paths and therefore the slower the algorithm BFSB. Due to the bad relative performance of MC21A for RBG-U graphs and BFSB for RBG-B graphs, we do not suggest using them for the maximum cardinality matching problem.

The performance of HK and ABMP-BFS are comparable with HKDW and PF for RBG-U graphs. Since ABMP-BFS is better than BFSB, we can argue that the two-stage approach works, and the first stage of ABMP-BFS that uses the shortest paths up to some level has a positive effect. In fact, ABMP-BFS is the best algorithm for RBG-U graphs. On the

Table VII. Execution Times of Algorithms in Seconds Combined with the MDM Heuristic for RBG-U and RBG-B Graphs. Each entry in the table is an average for 10 random matrices

Matrix	BFSB	MC21A	PF	PF+	HK	HKDW	ABMP-BFS
RBG-U(10, 3)	1.7	2.0	1.7	1.7	2.2	1.9	1.7
RBG-U(10, 5)	2.3	2.6	2.7	2.7	2.4	2.2	2.2
RBG-U(10, 10)	2.8	2.9	2.8	2.8	2.8	3.6	3.0
RBG-U(15, 3)	4.1	4.7	3.0	2.9	4.2	4.1	3.4
RBG-U(15, 5)	3.2	3.4	3.4	3.4	3.3	3.4	3.7
RBG-U(15, 10)	4.7	5.5	6.2	6.1	4.9	5.3	4.5
RBG-U(20, 3)	8.1	14.1	6.4	5.5	8.9	4.8	3.8
RBG-U(20, 5)	4.6	5.5	4.9	4.7	4.8	4.7	5.0
RBG-U(20, 10)	6.2	6.9	6.9	6.8	6.4	6.4	6.1
Avg. Relative Perf.	1.2	1.5	1.2	1.1	1.3	1.2	1.1
RBG-B(7, 1, 3)	1.1	1.2	1.2	1.2	1.2	1.1	1.0
RBG-B(7, 1, 5)	1.3	1.5	1.5	1.5	1.3	1.3	1.4
RBG-B(7, 1, 10)	2.1	2.2	2.4	2.4	2.1	2.5	2.1
RBG-B(7, 1, 15)	2.8	2.8	3.0	3.0	2.9	3.2	2.9
RBG-B(7, 40, 3)	1.1	1.3	1.2	1.2	1.7	1.2	1.1
RBG-B(7, 40, 5)	1.5	1.5	1.7	1.7	1.6	1.5	1.5
RBG-B(7, 40, 10)	1.9	2.0	2.0	2.0	1.9	2.1	1.9
RBG-B(7, 40, 15)	2.5	2.4	2.4	2.5	2.5	2.6	2.5
RBG-B(7, 100, 3)	2.6	2.4	1.7	1.5	3.8	2.6	2.5
RBG-B(7, 100, 5)	1.6	1.4	1.6	1.7	1.9	1.5	1.7
RBG-B(7, 100, 10)	1.7	1.8	1.8	1.8	1.7	1.8	1.7
RBG-B(7, 100, 15)	2.2	2.3	2.3	2.3	2.2	2.2	2.2
RBG-B(7, 200, 3)	3.6	4.3	3.0	2.5	7.4	5.2	3.3
RBG-B(7, 200, 5)	2.1	1.4	1.4	1.4	2.2	1.4	2.3
RBG-B(7, 200, 10)	1.6	1.8	1.8	1.8	1.6	1.6	1.6
RBG-B(7, 200, 15)	2.0	2.0	2.0	2.0	2.0	2.0	1.9
RBG-B(10, 1, 3)	1.4	1.5	1.5	1.5	1.5	1.5	1.6
RBG-B(10, 1, 5)	2.0	2.2	2.3	2.3	1.9	2.1	2.1
RBG-B(10, 1, 10)	3.3	3.6	4.1	4.0	3.4	3.7	3.0
RBG-B(10, 1, 15)	4.0	4.0	4.0	4.0	4.3	4.7	4.3
RBG-B(10, 40, 3)	1.8	2.1	2.0	1.9	2.5	1.9	1.6
RBG-B(10, 40, 5)	2.0	2.0	2.1	2.1	2.2	2.3	2.3
RBG-B(10, 40, 10)	2.9	3.1	3.2	3.2	2.9	3.0	2.9
RBG-B(10, 40, 15)	4.2	4.3	4.4	4.5	4.0	4.2	3.9
RBG-B(10, 100, 3)	3.0	2.9	2.1	2.0	4.5	3.1	2.6
RBG-B(10, 100, 5)	2.1	2.0	2.2	2.2	2.1	2.2	2.3
RBG-B(10, 100, 10)	2.6	2.8	2.7	2.8	2.6	2.6	2.6
RBG-B(10, 100, 15)	3.5	3.5	3.7	3.6	3.5	3.6	3.5
RBG-B(10, 200, 3)	5.7	7.8	4.5	3.7	12.2	6.7	5.1
RBG-B(10, 200, 5)	2.1	2.1	2.2	2.2	2.3	2.1	2.5
RBG-B(10, 200, 10)	2.4	2.5	2.5	2.6	2.4	2.4	2.4
RBG-B(10, 200, 15)	3.1	3.1	3.1	3.1	3.1	3.1	3.0
Avg. Relative Perf.	1.1	1.2	1.1	1.1	1.3	1.2	1.1

contrary, using shortest paths for every augmentation is not a good idea especially for RBG-B graphs. This is exemplified by HK: it uses shortest augmenting paths, and its relative performance is the worst.

As shown in Table VI, the modified versions of the algorithms work better. That is, HKDW is better than HK for both of the random matrix generators, and PF+ is slightly better than PF. For these experiments, the most important modification (HKDW vs. HK) comes from Duff and Wiberg [1988] since there is only a slight algorithmic difference between PF and PF+. Even though HK is one of the worst algorithms in terms of relative performance, HKDW, which combines HK with PF's multiple DFS search technique is one of the best algorithms for this set of experiments on random graphs.

As Table VII shows, when the MDM heuristic is used, the performance of the algorithms get closer especially for RBG-B graphs, for which the MDM heuristic obtains a nearly

Table VIII. The Number of Matrices (among 100) for which KSM and MDM Heuristics Find the Maximum Matching.

heuristic	A	AQ	PA	PAQ
KSM	16	16	16	16
MDM	56	53	53	53

Table IX. Execution Times of Algorithms in Seconds Combined with KSM and MDM for the Matrices *amazon0312* and *rajat21*.

Each entry in the table is the average of 10 random permutations

Matrix	Heuristic	Deficiency	BFSB	MC21A	PF	PF+	HK	HKDW	ABMP-BFS
<i>amazon0312</i>	KSM	11254	22.7	68.6	26.3	18.4	5.4	18.3	2.3
A	MDM	5852	25.1	61.0	26.2	19.1	5.8	16.6	2.2
<i>rajat21</i>	KSM	7421	2.0	14.8	6.9	6.4	4.6	2.1	2.5
PAQ	MDM	1302	2.4	10.2	6.3	5.8	5.2	2.4	3.3

maximum matching. Hence, this table does not say too much about the performance of the algorithms except that MC21A and HK are worse than the other algorithms.

5.3.3. Experiments with Matrices from Real-Life Problems. Since both of the heuristics work well, in order to differentiate between the algorithms and to compare their performance, we need to find graphs where KSM and MDM perform poorly. Constructing or characterising hard test cases for the bipartite cardinality matching problem is an interesting research area—a comment also made by Cherkassky et al. [1998]. We try to obtain such hard instances from the original and permuted versions of real-life matrices from which we believe we can build a fair benchmark set to compare the performance of the algorithms.

The KSM and MDM heuristics perform very well on the real-life matrices; the numbers of matrices (among 100) for which they can find the maximum matching are given in Table VIII. For real matrices, the average deficiency of the KSM heuristic is 3738 whereas it is 445 for MDM.

Even when the heuristics cannot find a maximum cardinality matching, their deficiencies are usually very small. From these numbers, we can argue that MDM performs better than KSM in terms of deficiency. Note that a smaller deficiency usually implies a shorter execution time of the ensuing exact algorithm. However, there are some matrices for which the algorithms using KSM are as fast as the ones using MDM even when MDM obtains a much larger initial matching. Table IX shows this by an example with the matrices *amazon0312* (the original matrix) and *rajat21* (row-wise and column-wise permuted version). For the two matrices in the table, the heuristics KSM and MDM take around 0.1 and 0.4 seconds on the average, respectively.

Tables X and XI show the execution times and average relative performance of the algorithms for some of the test matrices. For each matrix, we performed four sets of experiments. First, we execute the corresponding algorithm on the original matrix. Second, we apply ten random column permutations to the original matrix and execute the algorithm for each column-permuted matrix. Third, we apply ten random row permutations to the original matrix and execute the algorithm for each row-permuted matrix. Last, we apply ten random row and column permutations, and execute the algorithm for each totally permuted matrix. Averages of these ten values are given in the table for each set of experiments. Note that we build the CSR or CSC after applying the permutations. In this setting, the indices in *rids* or *cids* are stored in increasing order.

For some other matrices (not shown in the tables), the heuristics find a matching of the maximum cardinality or a matching whose cardinality is very close to the

Table X. Execution Times of Algorithms in Seconds Combined with the KSM Heuristic for Real-Life Matrices.

Each entry in the table is the average of 10 random permutations. The second column shows the permutation. Only the matrices for which the algorithms are highly perturbed by random permutations are given in the table. The relative average performance of algorithms for these matrices and for all 100 matrices are given

Matrix	Perm	Avg. Def.							ABMP-
			BFSB	MC21A	PF	PF+	HK	HKDW	BFS
<i>pre2</i>	A	11878	8.5	37.2	3.0	0.2	1.2	1.8	3.0
	AQ	11834	3.7	69.4	26.7	0.7	2.5	1.1	5.8
	PA	11830	20.0	4.8	0.7	0.4	2.8	10.8	4.2
	PAQ	11835	5.3	6.5	1.1	0.7	4.1	1.2	7.3
<i>t2em</i>	A	13182	4.8	135.6	4.6	0.1	6.7	3.2	0.6
	AQ	13239	1.6	83.2	53.3	0.8	12.7	2.1	2.0
	PA	13325	14.9	4.6	1.3	0.9	11.1	47.4	1.8
	PAQ	13320	2.4	3.1	1.6	1.3	16.0	1.8	2.9
<i>tmt_unsym</i>	A	13156	6.2	142.0	5.1	0.1	6.1	3.8	0.6
	AQ	13305	1.7	90.2	61.4	1.2	12.5	2.3	2.0
	PA	13336	18.8	4.9	1.1	0.8	11.4	51.6	1.9
	PAQ	13286	2.0	3.0	1.5	1.2	16.2	1.9	2.7
<i>tmt_sym</i>	A	13306	6.0	147.8	3.8	0.1	3.7	3.1	0.5
	AQ	13330	1.4	100.5	72.1	0.6	8.8	1.5	1.4
	PA	13326	16.5	3.3	0.7	0.6	7.6	56.6	1.4
	PAQ	13410	2.2	2.2	1.1	1.0	11.9	1.3	2.2
<i>apache2</i>	A	6762	5.0	114.3	17.7	1.9	3.5	16.5	0.9
	AQ	6744	1.9	195.4	110.6	0.3	7.6	0.6	1.9
	PA	6703	14.3	5.3	0.5	0.5	7.3	91.3	1.9
	PAQ	6741	2.7	4.1	0.8	0.5	9.9	0.8	3.0
<i>Hamrle3</i>	A	23057	10.4	135.4	26.5	4.6	29.8	0.9	11.7
	AQ	23127	28.7	233.6	100.6	11.0	54.1	37.9	36.5
	PA	23062	24.9	177.7	22.8	20.9	64.1	3.0	26.4
	PAQ	23115	42.9	100.9	47.4	46.4	85.4	61.8	45.4
<i>ecology1</i>	A	14461	5.7	216.9	15.2	1.1	6.9	13.8	0.7
	AQ	14471	1.7	195.6	119.6	0.8	14.3	2.4	2.4
	PA	14498	17.8	4.7	1.3	0.9	13.0	94.5	2.1
	PAQ	14494	3.0	3.1	1.5	1.3	18.5	2.0	3.0
<i>ecology2</i>	A	14578	5.8	220.2	15.6	1.1	8.0	13.6	0.6
	AQ	14438	1.9	193.1	119.3	0.8	15.0	2.4	2.3
	PA	14447	17.7	4.7	1.2	0.8	12.7	96.7	2.1
	PAQ	14415	2.4	3.3	1.8	1.3	18.4	1.9	2.7
<i>amazon0312</i>	A	11254	22.7	68.6	26.3	18.4	5.4	18.3	2.3
	AQ	11252	24.1	73.2	30.6	21.5	6.1	20.0	2.1
	PA	11235	26.1	93.2	38.9	27.4	6.1	22.7	3.6
	PAQ	11258	27.7	99.1	44.5	30.6	7.1	23.0	2.5
<i>language</i>	A	3707	4.6	16.9	2.5	0.9	2.2	1.0	5.7
	AQ	3726	8.2	21.5	4.8	1.8	3.1	2.5	6.7
	PA	3668	6.1	19.1	3.4	2.2	2.9	1.0	6.0
	PAQ	3719	10.1	23.6	6.0	3.9	4.2	3.1	7.4
Relative average performance	A		26.2	544.3	26.1	2.4	23.6	18.1	5.5
	AQ		4.1	165.1	97.0	1.9	11.6	3.1	3.5
	PA		21.0	15.5	3.1	2.4	11.7	65.5	4.2
	PAQ		3.8	7.9	3.0	2.2	9.8	2.2	3.2
Overall relative average performance	A		6.1	95.5	7.2	1.3	5.9	5.1	2.4
	AQ		2.9	47.4	25.3	1.3	5.4	1.7	2.6
	PA		7.3	8.7	1.6	1.3	5.3	14.9	3.0
	PAQ		3.4	4.1	1.6	1.3	5.7	1.8	3.2

maximum, even after permuting the matrix. Hence, the execution times of the algorithms are small, and the performance of them are very close to each other. On the other hand, for some of the matrices, the heuristics obtain a matching with high deficiency, usually after permutations, but still almost all of the algorithms perform quite

Table XI. Execution Times of Algorithms in Seconds Combined with the MDM Heuristic for Real-Life Matrices. Each entry in the table is the average of 10 random permutations. The second column shows the permutation. Only the matrices highly sensitive to the permutations are given in the table. The relative average performance of the algorithms for these matrices and for all 100 matrices are given

Matrix	Perm	Avg. Def.								ABMP-
			BFSB	MC21A	PF	PF+	HK	HKDW	BFS	
<i>pre2</i>	A	408	1.9	2.6	0.8	0.4	1.0	1.6	1.5	
	AQ	554	7.1	13.5	6.3	1.0	3.1	1.1	6.3	
	PA	341	3.6	1.6	0.8	0.8	2.2	3.2	2.5	
	PAQ	440	10.6	2.5	1.3	1.2	4.7	1.5	7.7	
<i>Hamrle3</i>	A	39	0.7	0.4	0.3	0.3	1.2	0.3	0.7	
	AQ	1162	179.1	92.7	70.2	11.9	65.9	34.1	198.9	
	PA	22544	75.9	165.9	22.3	19.9	93.4	4.2	66.7	
	PAQ	18288	144.1	87.7	47.1	46.2	144.7	68.4	149.2	
<i>rajat21</i>	A	1292	0.6	0.2	0.2	0.4	1.1	0.4	0.5	
	AQ	1298	1.3	0.7	0.6	0.9	2.7	1.1	2.2	
	PA	1158	1.8	5.2	2.2	2.0	3.0	1.3	1.8	
	PAQ	1302	2.4	10.2	6.3	5.8	5.2	2.4	3.3	
<i>rajat29</i>	A	596	0.3	0.2	0.3	0.3	0.8	0.4	0.9	
	AQ	549	1.1	0.7	0.8	0.7	2.3	1.2	3.9	
	PA	505	1.0	0.7	1.3	1.0	2.7	1.4	1.7	
	PAQ	533	1.7	1.4	1.6	1.5	3.0	1.8	2.7	
<i>amazon0312</i>	A	5802	25.1	61.0	26.2	19.1	5.8	16.6	2.2	
	AQ	5838	27.2	70.3	31.8	22.9	6.8	19.4	3.5	
	PA	5928	28.3	85.7	38.8	27.1	6.6	21.0	5.2	
	PAQ	5981	30.5	95.4	45.0	31.5	7.7	22.5	4.5	
<i>language</i>	A	2677	4.9	13.9	2.7	1.1	2.2	0.8	5.5	
	AQ	2874	10.0	19.4	5.1	2.1	3.5	2.8	6.4	
	PA	3253	6.6	18.1	3.7	2.5	3.1	1.2	6.3	
	PAQ	3467	12.4	23.9	6.6	4.4	4.6	3.6	7.6	
Relative average performance	A		4.9	9.2	3.5	2.6	3.6	2.9	3.5	
	AQ		6.4	8.8	4.3	2.0	3.3	2.4	6.1	
	PA		6.2	13.4	3.6	2.8	5.9	2.3	4.9	
	PAQ		4.1	6.2	3.0	2.3	2.4	1.8	2.7	
Overall relative average performance	A		1.6	2.2	1.4	1.2	1.6	1.2	1.3	
	AQ		1.7	2.3	1.6	1.2	1.6	1.2	1.5	
	PA		1.9	2.4	1.3	1.2	1.8	1.5	1.7	
	PAQ		1.5	1.9	1.3	1.2	1.5	1.2	1.3	

efficiently, that is, less than 3 seconds. To avoid having large tables, we only put the most interesting results where the heuristics, and hence the algorithms are evidently sensitive to the permutations. Note that the average relative performance of the algorithms are computed by taking all of the results into account, including those not given in the table.

The experiments with real-life matrices also show that the modified versions PF+ and HKDW perform better than PF and HK, and that MC21A performs relatively badly compared to other algorithms. Although the runtimes of the algorithms get closer when MDM is used instead of KSM, we can argue that PF+ and HKDW are better than the other algorithms with MDM.

As Tables X and XI show, PF+ is less sensitive to permutations of the matrices than the original algorithm PF. Furthermore, it appears that the fairness mechanism that we have incorporated into PF makes it the most robust algorithm among all the algorithms in this article. The algorithms HK and ABMP-BFS seem to be robust but the relative performance of HK is worse than HKDW, ABMP-BFS and PF. Although PF+ seems to be the most efficient and robust algorithm, there are some cases on which it performs much worse than ABMP-BFS (see *amazon0312*).

The difference between the performance of PF (or PF+) and MC21A is huge in favor of PF. This result is interesting since they both use nothing but DFS with a lookahead

technique. Furthermore, MC21A can prune all of the visited vertices after each unsuccessful DFS, whereas PF can do the same after only some of the DFSs. We observed that if one uses multiple DFSs like PF, it is highly probable that the algorithm finds a large maximal set of augmenting paths.

In agreement with the results of the experiments with random graphs given in Table IV, for most of the real-life matrices BFSB is better than MC21A. It is interesting to see that when KSM is used, BFSB is better than MC21A for all of the 100 unpermuted matrices except *FS_b39_c30*, *rajat24*, and *c-big*. However, especially for the row-permuted matrices, there are some cases where MC21A is 5 times faster than BFSB such as *pre2* and *tmt_sym* in Table X.

6. DISCUSSIONS AND CONCLUSIONS

We have reviewed and implemented eight algorithms for solving the problem of finding a maximum cardinality matching in bipartite graphs namely, DFSB, BFSB, MC21A, PF, PF+, HK, HKDW, and ABMP. We have also reviewed and implemented three heuristics that are used in general as a preprocessing step by these exact algorithms as a jump-start routine. We have described the details of the algorithms and systematized, developed, and used several ideas for enhancing performance. We have suggested a simple yet effective modification to PF that we call PF+. Although PF seems to be an efficient algorithm in practice, we have witnessed cases (especially with real-life matrices) in which it is badly affected by row and column permutations. The experiments show that PF+ withstands the permutations much better than any of the other algorithms while exhibiting the same practical performance as PF.

We have performed extensive tests with the algorithms and concluded the following.

- Instead of SGM or KSM one should use MDM.* Although we observed some cases where using MDM may increase the execution times of the algorithms, these cases are very rare, and MDM's positive effect is much higher than its negative effect especially on the matrices from the UFL collection.
- The pure search techniques such as DFSB and BFSB are not robust.* That is, the probability that they suffer due to a worst-case performance is much higher than other algorithms in this article. Although for the RBG-U random matrix class, BFSB is a good alternative, there are algorithms that are better in general. Hence, we do not suggest using either DFSB or BFSB.
- Instead of PF and HK, one should use PF+ and HKDW.* For PF+, there is almost no overhead for the fairness mechanism, and the final algorithm is more robust. The improvement due to this additional mechanism on the average execution times over PF is very impressive for some matrices. For HKDW, although there is the overhead of additional DFSs, the reduction in the execution time is high, and there is a good overall improvement over HK.
- ABMP and its variants are not the best performing algorithms as stated in the literature.* Previous results show that ABMP is much better than HK and other existing algorithms. Most of these studies use SGM as the jump-start routine and do not have all the algorithms in this article. As our experiments show, using MDM instead of SGM decreases the effect of the first stage, and the performance of ABMP-BFS gets more closer to the performance of HK. Additionally, the algorithm PF+ has a better performance on average (although there are some cases for which PF+ performs much worse than HK, HKDW, or ABMP).

In addition to these practical conclusions, our experience also leads to the following two more theoretically oriented results.

- Parallelization is not straightforward.* Although we did not discuss the parallelization of bipartite cardinality matching algorithms, we make the following comment. Our experiments with the matrices from the UFL collection in which we randomly permuted the rows and/or columns of the matrices show that even if these algorithms are parallelized, the partition of the data and/or the order in which the vertices/edges are processed would affect the run time, and hence there is no easy way to conduct speedup studies.
- A lower bound on the time complexity needs to be developed.* As stated in Section 3.3.1, the total length of the shortest augmenting paths is $\mathcal{O}(n \log n)$ for algorithms that augment matchings through the shortest augmenting paths. It seems therefore highly implausible that there would be an augmenting path-based algorithm that breaks this barrier. By $\mathcal{O}(n \log n)$, we mean the worst-case performance, and therefore we rule out the randomized algorithms and analysis.

REFERENCES

- ALT, H., BLUM, N., MEHLHORN, K., AND PAUL, M. 1991. Computing a maximum cardinality matching in bipartite graph in time $\mathcal{O}(n^{1.5} \sqrt{m/\log n})$. *Inf. Proc. Lett.* 37, 4, 237–240.
- ARONSON, J., FRIEZE, A. M., AND PITTEL, B. 1998. Maximum matchings in sparse random graphs: Karp-Sipser revisited. *Rand. Struct. Algor.* 12, 111–177.
- BALINSKI, M. L. 1986. A competitive (dual) simplex method for the assignment problem. *Math. Prog. (Series B)* 34, 2, 125–141.
- BALINSKI, M. L. AND GONZÁLEZ, J. 1991. Maximum matching in bipartite graphs via strong spanning trees. *Networks* 21, 2, 165–179.
- BAST, H., MEHLHORN, K., SCHAFER, G., AND TAMAKI, H. 2006. Matching algorithms are fast in sparse random graphs. *Theory of Comput. Syst.* 39, 1, 3–14.
- BERGE, C. 1957. Two theorems in graph theory. In *Proc. Nat. Acad. Sci. USA.* 43, 842–844.
- CHEBOLU, P., FRIEZE, A., AND MELSTED, P. 2010. Finding a maximum matching in a sparse random graph in $\mathcal{O}(n)$ expected time. *J. ACM* 57, 4, 1–27.
- CHEKASKASY, B. V., GOLDBERG, A. V., MARTIN, P., SETUBAL, J. C., AND STOLFI, J. 1998. Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. *J. Exper. Algor.* 3, 8.
- DAVIS, T. A. 2006. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms 2. SIAM, Philadelphia, PA, USA.
- DUFF, I. S. 1981. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Soft.* 7, 315–330.
- DUFF, I. S., ERISMAN, A. M., AND REID, J. K. 1986. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, UK.
- DUFF, I. S. AND WIBERG, T. 1988. Remarks on implementations of $\mathcal{O}(n^{1/2}\tau)$ assignment algorithms. *ACM Trans. Mathe. Softw.* 14, 267–287.
- DYER, M., FRIEZE, A., AND PITTEL, B. 1993. The average performance of the greedy matching algorithm. *Ann. Appl. Prob.* 3, 526–552.
- ERDÖS, P. AND RÉNYI, A. 1959. On random graphs. i. *Publ. Mathe.* 6, 290–297.
- EVEN, S. AND TARJAN, R. E. 1975. Network flow and testing graph connectivity. *SIAM J. Comput.* 4, 507–518.
- FEDER, T. AND MOTWANI, R. 1995. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. Syst. Sci.* 51, 2, 261–272.
- GABOW, H. N. AND TARJAN, R. E. 1988. Algorithms for two bottleneck optimization problems. *J. Algor.* 9, 3, 411–417.
- GOLDBERG, A. V. AND TARJAN, R. E. 1988. A new approach to the maximum flow problem. *J. ACM* 35, 921–940.
- GONZÁLEZ, J. AND LANDAETA, O. 1995. A competitive strong spanning tree algorithm for the maximum bipartite matching problem. *SIAM J. Discr. Math.* 8, 186–195.
- GUSTAVSON, F. G. 1976. Finding the block lower-triangular form of a sparse matrix. In *Sparse Matrix Computations*, J. R. Bunch and D. J. Rose, Eds., Academic Press, New York, 275–289.
- HOPCROFT, J. E. AND KARP, R. M. 1973. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.* 2, 4, 225–231.
- KARP, R. M. AND SPSER, M. 1981. Maximum matching in sparse random graphs. In *Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computer Science (FOCS 1981)*. IEEE Computer Society, Los Alamitos, CA, 364–375.

- KORTE, B. AND HAUSMANN, D. 1978. An analysis of the greedy algorithm for independence systems. *Ann. Discr. Mathe.* 2, 65–74.
- LANGGUTH, J., MANNE, F., AND SANDERS, P. 2010. Heuristic initialization for bipartite matching problems. *J. Exper. Algor.* 15, 1.1–1.22.
- LOVASZ, L. AND PLUMMER, M. D. 1986. *Matching Theory*. North-Holland Mathematics Studies. Elsevier Science Publishers, Amsterdam, Netherlands.
- MAGUN, J. 1998. Greeding matching algorithms, an experimental study. *J. Exper. Algor.* 3.
- MEHLHORN, K. AND NÄHER, S. 1999. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK.
- MOTWANI, R. 1994. Average-case analysis of algorithms for matchings and related problems. *J. ACM* 41, 1329–1356.
- MUCHA, M. AND SANKOWSKI, P. 2004. Maximum matchings via Gaussian elimination. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*. IEEE Computer Society.
- POTHEN, A. AND FAN, C.-J. 1990. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw.* 16, 303–324.
- SETUBAL, J. C. 1992. New experimental results for bipartite matching. Tech. rep. DCC-07/92, University of Campinas.
- SETUBAL, J. C. 1996. Sequential and parallel experimental results with bipartite matching algorithms. Tech. rep. IC-96-09, University of Campinas.

Received September 2010; revised March 2011; accepted May 2011