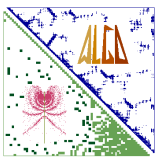
Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique

Design, Implementation, and Analysis of Maximum Transversal Algorithms

I. S. Duff, K. Kaya and B. Uçar

Technical Report TR/PA/10/76



Publications of the Parallel Algorithms Team
http://www.cerfacs.fr/algor/publications/

Abstract

We report on careful implementations of seven algorithms for solving the problem of finding a maximum transversal of a sparse matrix. We analyse 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 paper or they used non-uniform 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.

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 corresponds to a matching of cardinality n in $G_{\mathbf{A}}$. 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 paper is to review those algorithms. We have implemented seven existing algorithms that are described in Table 1. We give the distinguishing features of each algorithm in the second column. These features will be more 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 methods based on augmenting paths for the maximum cardinality bipartite matching. 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

Table 1: 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.

Algorithm	Description	Complexity
`DFSB`	Based on DFS. Quite common in software libraries, for example `dmperm` in Matlab [9] and MC21 in HSL	$\mathcal{O}(n\tau)$
`BFSB`	Based on BFS. Also quite common, see the algorithm FF in [26]	$\mathcal{O}(n\tau)$
`MC21A`	DFS+Lookahead [10]	$\mathcal{O}(n\tau)$
`PF`	Disjoint DFSs [29]	$\mathcal{O}(n\tau)$
`HK`	Shortest disjoint augmenting paths [20]	$\mathcal{O}(\sqrt{n}\tau)$
`HKDW`	`HK`+Disjoint DFS [12]	$\mathcal{O}(\sqrt{n}\tau)$
`ABMP`	Combined DFS and BFS [1]	Min. of $\mathcal{O}(\sqrt{n}\tau)$ and $\mathcal{O}(n^{1.5}\sqrt{\tau/\log n})$
`PF+`	A simple modification of `PF` (this paper)	$\mathcal{O}(n\tau)$

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.

In this paper, 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. Besides, we also improve the implementation of `HKDW` and make it faster than the one implemented by Duff and Wiberg [12].

In addition to the exact algorithms listed above, there are several fast heuristics which 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, e.g., [10] (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 [26]. However,

there are more sophisticated and better heuristics such as the Karp-Sipser heuristic, KSM [21], the minimum-degree heuristic MDM [23, 25] and some others proposed by Langguth et al. [23]. 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 paper exist [8, 10, 12, 26, 23, 29, 31], we think that they are not conclusive because of the following two reasons: first, all previous studies, except [23], 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 paper 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 $|\mathcal{M}|$ is the cardinality of $\mathcal{M}$. 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 matching cardinality and $|\mathcal{M}|$. A good discussion on matching theory can be found in Lovasz and Plummer's book [24].

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 paper.

Theorem 2.1 (Berge [6]) *Let G be a graph (bipartite or not) and $\mathcal{M}$ 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 $\oplus$ is the symmetric difference of two sets and $E(\mathcal{P})$ is the edge set of a path $\mathcal{P}$. 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 are the 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 $G_{\mathbf{A}}$. Let $\mathcal{M}$ be a perfect matching in $G_{\mathbf{A}}$. Then one can find an $n \times n$ permutation matrix $\mathbf{Q}$ such that $\mathbf{Q}_{ji} = 1$ iff row i and column j are matched in $\mathcal{M}$ so that the matrix $\mathbf{AQ}$ 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 $\mathbf{PAQ}$ 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, for example, [11, 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:

- $rids[1, \ldots, \tau]$: stores the row index of each nonzero entry. The nonzeros in a column are stored consecutively.

- $cptrs[1, \ldots, n+1]$: stores the location of the first nonzero of each column in array $rids$. In particular the row indices of the nonzeros in column j are stored in $rids[cptrs[j], \ldots, cptrs[j+1]-1]$. Note that $cptrs[n+1] = \tau + 1$.

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 those of the above. 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 is stored in two arrays. The array $rmatch[1, \ldots, m]$ stores the index of the columns matched to the rows; $rmatch[i] = j$ if row i is matched to column j, otherwise $rmatch[i] = -1$. The array $cmatch[1, \ldots, n]$ stores the dual information similarly.

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**), 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` more robust 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 (for example 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, we can work with the transposed matrix.

In a BFS based matching algorithm, `BFSB`, the unmatched column vertices are processed in a particular 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 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.

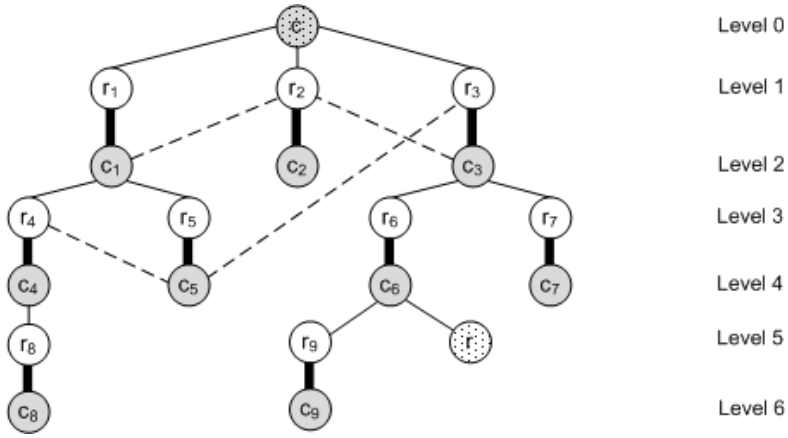


Figure 1: A BFS tree constructed from finding alternating paths from an unmatched column vertex c. The edges in the matching are shown as thick lines and those that do not belong to the tree are shown as dashed.

To implement `BFSB`, we need a *queue* of size n and another array of size m to mark the rows as *visited*. Note that, since we begin with an unmatched column c_i, we only put columns in the queue. 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 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, i.e., the corresponding entry of the *visited* array is set to i, and the column matched to r_k is enqueued. This process is called a `BFSB` *phase*. For each unmatched column in the matrix, this phase is repeated consecutively so the efficiency might depend on the ordering of the columns. Since the complexity of a phase is $\mathcal{O}(\tau)$, and since there are n columns, the total time complexity of the algorithm 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 [31][1]) which may reduce the execution time when the matrix is rank deficient, i.e., 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 BFS 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 subsequent BFSs.

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 $c' \in \{0, \ldots, n-1\}$ 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

[1] We do not know if the BFS algorithm used by Langguth et al. employs a pruning scheme since the algorithm is not public and not described in detail [23].

($parent(r) \bmod n) + 1$. 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 favour the runtime in our implementations and so do not incorporate these memory optimizations.

3.2 Algorithms based on depth-first search

DFS based algorithms are similar to the BFS based ones. As such, DFS based algorithms can be implemented to start either from the columns or the rows. We again choose to start from the columns; if there is a preference to start from the rows, the matrix can be transposed for the following discussion.

In a DFS based matching algorithm, the unmatched column vertices are processed in a particular order (normally from 1 to n). For each unmatched column, a DFS is initiated and when a column c is visited, the search path is extended with the next unvisited row of c whereas when a row r is visited, the search path is extended with the column matched with r. If such a column does not exist, i.e., if r is unmatched, then the search has found an augmenting path. 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, i.e., when all of the rows of the last column c in the *stack* have been visited by the current DFS, we remove c from the stack and continue with the previous entry on the stack. We repeat this process until an augmenting path is found or the stack is empty. This process is called 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, we start the search from position $(lastrow(c) + 1)$ in c's adjacency list. This allows us to avoid multiple scans of adjacency lists. We reset $lastrow(c)$ each time we put a column c into the stack. With this approach, each edge is scanned just 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

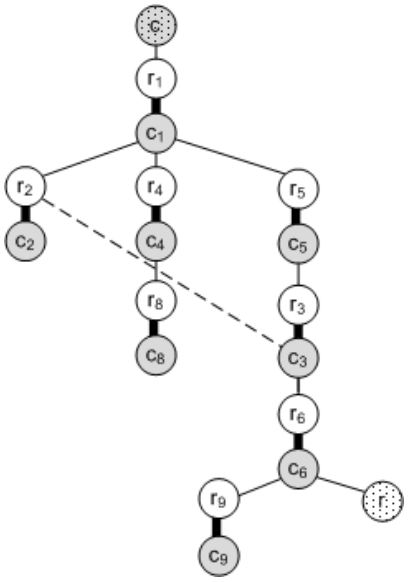


Figure 2: A DFS tree constructed while searching for an augmenting path from an unmatched column node c in the graph shown in Figure 1. Matching edges are shown with thick lines, where the edges between columns and already visited rows are shown as dashed.

corresponding augmenting path from c 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. 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 $lastrow(c)$ to check if the row r, matched with column c, 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 putting each popped column to the end of the *stack* and using this part of the stack to prune the rows matched with these columns by modifying their *visited* entries. We adapt the same ideas for avoiding resets and pruning rows for the other DFS based or hybrid algorithms in this section.

3.2.1 MC21A: Duff's algorithm

MC21A is a DFS based algorithm with an enhancement on the search process [10]. 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 [10, 19, 29]. 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 c 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.

Note that we also use the pruning heuristic described in Section 3.1 which

is not a part of the original implementation of `MC21A` by Duff [10].

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.

3.2.2 `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 each 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. 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 $\mathcal{O}(\tau)$ 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 node c does not prove the nonexistence 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 does not use this mechanism [29].

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.

3.2.3 `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 of visiting the rows in the adjacency lists of columns 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, i.e., the last row in the adjacency list is the first one to be investigated by the DFSs. The purpose of this modification is to treat each row in an adjacency list fairly in order to spread the search more evenly in the graph and to find, hopefully, an unmatched row faster than usual. 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.

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 which use DFS and BFS together and the level information of the vertices 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 also organizes the search for augmenting paths into phases [20]. 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. The algorithm then continues with the next phase until a maximum matching is found. Hopcroft and Karp 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 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, an *auxiliary graph* is constructed explicitly after the combined BFS [20]. This graph contains vertices visited by the combined BFS, and the edges (u, v) where u and v belong to 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, i.e., 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 stated above, 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 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) using a maximal set of vertex disjoint shortest augmenting paths of the shortest length strictly increases the shortest augmenting path length in the graph. These two results are used to demonstrate a time complexity of `HK` as follows [20]: 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 (i) above, 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 (ii) above, 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 [14] for this interpretation). Even and Tarjan 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 $\mathcal{O}(n \log n)$ [14, Theorem 5]. 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. The total length of augmenting paths through the course of the algorithm can be $\mathcal{O}(n^2)$.

As stated above, 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 processing 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 in [12] by Duff and Wiberg, 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 them 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 at the top level. 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 adds a third part to a phase of HK [12]. 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 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 to increase 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 which 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 in [12] and it will be denoted as HKDW in this paper.

3.3.3 ABMP: Alt et al.'s algorithm

This algorithm by Alt et al. incorporates some techniques used in solving maximum flow problems [17] into the Hopcroft and Karp's algorithm [1]. It runs in two consecutive stages. In the first stage, a set of augmentations are performed using a sophisticated search procedure which 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 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, each row and column is assigned to level 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 increasing during the course of 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 maintaining dynamic level information until a lower bound on the shortest augmenting path length is reached and then switches to HK to obtain the maximum matching.

The authors proved theoretically 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 $\mathcal{O}\left(\min(\sqrt{n}\tau, n^{1.5}\sqrt{\tau/\log n}\right)$. In our implementation, as suggested by Mehlhorn and Näher [26], the first stage continues until $L > 0.1\sqrt{n}$ or $50L$ is greater than 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 wrong level attributes, are always unsuccessful. However, these unsuccessful DFSs are necessary to update the level attributes. But 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 in [31] by Setubal, we periodically execute a global update procedure which 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 above, Alt et al. proposed to use `HK` to obtain a better overall complexity. Mehlhorn and Näher suggest using `BFSB` in the second stage [26]. In our experiments, in addition to the original version by Alt et al., we follow the decision in [26] and implement another version called `ABMP-BFS` which 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 before.

For example, in [4], Balinski and González proposed an $\mathcal{O}(n\tau)$-time algorithm based on a special structure known as a *strong spanning tree* (first defined by Balinski as a spanning tree rooted at a row vertex that has every column vertex of degree one as a child of the root [3]). González and Landaeta showed that the approach originating from the concept of strong spanning trees is related to the notion of augmenting paths[18]. By using this relation and the ideas in [20], they 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 [28]. 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 analyses of the maximum matching algorithms. Motwani analyses the `HK` algorithm on random graphs (on bipartite and also on standard undirected graphs) in order to characterize the average case behaviour of `HK` [27]. 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 with high probability. Later, Bast et al. improve this result where the average degree is a large enough constant bigger than or equal to 8.83 [5]. 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 $\mathcal{O}(\log n)$. In [7], Chebolu et al. take the output of the jump-start routine `KSM` (discussed in the next section), try to augment it by searching paths only between a subset of the unmatched vertices, and then use the algorithm of Bast et al. 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 by Feder and Motwani, where the runtime complexity of `HK` becomes $\mathcal{O}(\tau^* \sqrt{n})$-time where $\tau^* = \tau \log (n^2/\tau)/ \log n$ [15]. 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 [20] to obtain an $\mathcal{O}(p\tau)$ time algorithm that finds a matching with cardinality no less than $\frac{p}{p+1}\delta_{\max}$, where δ_{max} is the cardinality of a maximum matching. Let us make the runtime analysis of `HK` given in Section 3.3.1 in the reverse order. Let p be the current number of phases, δ be the size of the current matching Then the shortest

augmenting path length is at least $2p+1$, as each phase increases this length by at least 2; there are at most $\frac{\delta}{p}$ vertex disjoint augmenting paths, since we have δ matching edges and an alternating path should contain at least p of them. Since each augmenting path increases the cardinality of the current matching by one, δ_{max} can be at most $\delta + \delta/p$. One can rearrange the inequalities as $\delta \geq (1 - \frac{1}{p+1})\delta_{max}$ to see the approximation at the end of pth phase. This remark is also made by Gabow and Tarjan [16, 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 [29, 21, 23, 25, 31]. 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 [25] and Langguth et al. [23]. 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 [25] and [23].

4.1 Simple greedy matching (SGM)

The first heuristic, Algorithm 1, is used by Pothen and Fan [29] and Duff and Wiberg [12]. 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, e.g., LEDA [26], and `MC21A` in HSL. Also it is frequently used in previous papers that compare the performance of matching heuristics [30, 31]. A similar greedy matching heuristic, which iteratively chooses a random edge with two unmatched endpoints, is described by Dyer et al. [13], Korte and Hausmann [22], and Magun [25]. 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 [13] and [25].

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**
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\}$;
    $G \leftarrow G \setminus \{u, v\}$ ▶*remove vertices u, v and all edges incident on them*
  end
end
return $\mathcal{M}$;

Algorithm 1: Simple Greedy Matching (`SGM`)

4.2 Karp-Sipser greedy matching (KSM)

The second greedy heuristic is proposed by Karp and Sipser [21]. 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 the reductions above 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 analyse and proposed a simpler algorithm which is given below as Algorithm 2. This version, referred to as `KSM`, is called the Karp-Sipser heuristic in the literature, and its performance is analysed by Aronson et al. [2] and Chebolu et al. [7]. In our implementation of Algorithm 2, we maintain the degrees dynamically, i.e., when a matching (u, v) occurs we decrease the degrees of vertices adjacent to u or v by 1.

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 $\mathcal{M}$;

Algorithm 2: Karp-Sipser Heuristic (`KSM`)

In the `KSM` heuristic, if the **if** statement is true the matching decision is optimal, i.e., there exists a maximum cardinality matching in the current graph G which 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, 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 analysed by Magun [25]. 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 [25]. We also direct the user to [23] for a more recent and comprehensive analysis of the heuristics and their effects on some of the algorithms described in this paper.

4.3 Minimum degree matching (MDM)

The minimum degree matching heuristic first chooses an unmatched vertex u with minimum degree [25]. The one-sided version chooses a random unmatched vertex v in the adjacency list of u, whereas the two-sided version chooses an unmatched row 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 [23], 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. This two-sided minimum degree matching heuristic shown below in Algorithm 3 uses a priority queue and has a runtime complexity of $\mathcal{O}(n+\tau)$.

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 the vertex u with the minimum positive degree;
  choose the 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 $\mathcal{M}$;

Algorithm 3: Minimum Degree Matching Heuristic (MDM)

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 show 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 paper. 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.

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 is used by Magun [25] and behaves similarly to 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 ith 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. [23], Cherkassky et al. [8] and Setubal [31].

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 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 6 matrices satisfying these properties in a group, we choose those with the largest number of nonzeros. There are 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 2.

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 paper [12, 23, 29, 31]. The `KSM` heuristic was experimented and analysed extensively on its own and used as a part of some randomized algorithms [21, 7]. The `MDM` heuristic is one of the best, but its effect on the matching algorithms have not been thoroughly investigated.

Table 3 shows the performance of the heuristics within `BFSB` on the 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 [23, 25], we give for each

Table 2: Test matrices from UFL Sparse Matrix Collection used in the experiments

Group	Name	Group	Name	Group	Name
AMD	G2_circuit	JGD_GL7d	GL7d15	Ronis	xenon2
Andrianov	ins2		GL7d23	Rothberg	cfd2
Andrianov	lp1	JGD_Homology	ch7-9-b4	Sandia	ASIC_320k
ATandT	pre2		ch7-9-b5		ASIC_320ks
	twotone		ch8-8-b4		ASIC_680k
Barabasi	NotreDame_actors		ch8-8-b5	Sch._IBMNA	c-73
	NotreDame_www		n4c6-b8		c-big
Botonakis	FEM_3D_thermal2		n4c6-b9		c-73b
	thermomech_TC	JGD_Margulies	kneser_10_4_1	Sch._IBMSDS	matrix_9
	thermomech_dM	Kamvar	Stanford		matrix-new_3
	thermomech_dK	Lin	Lin	Sch._ISEI	barrier2-4
CEMW	t2em	McRae	ecology1		barrier2-9
	tmt_unsym		ecology2		ohne2
	tmt_sym	Mittelmann	sgpf5y6		para-4
DNVS	ship_003		stormG2_1000	SNAP	web-Google
	shipsec1		watson_1		amazon0312
	shipsec5		watson_2		amazon0505
	shipsec8		neos		amazon0601
FreeFieldTech	mono_500Hz		neos3		roadNet-PA
Freescale	transient	Norris	lung2		roadNet-TX
GHS_indef	d_pretok		stomach	TKK	engine
	darcy003		torso2	Tromble	language
	helm2d03		torso3	TSOPF	FS_b39_c30
	mario002	Oberwolfach	filter3D	Um	2cubes_sphere
	turon_m		boneS01		offshore
	boyd2	Pajek	IMDB	UTEP	Dubcova3
GHS_psdef	ford2	PARSEC	Ga10As10H30	vanHeukelum	cage12
	apache2	QLi	crashbasis	Wissgott	parabolic_fem
Hamm	hcircuit		largebasis	Watson	Baumann
	scircuit	Rajat	Raj1	Yoshiyasu	image_interp
Hamrle	Hamrle3		rajat21		
HVDC	hvdc2		rajat23		
IBM_EDA	dc1		rajat24		
	trans4		rajat29		
JGD_Forest	TF19		rajat30		

Table 3: Execution times and the number of augmentations performed by the algorithm BFSB when combined with SGM, KSM and MDM for the matrices generated by the RBG-U and RBG-B generators. For each parameter set, the first row shows the execution times of BSFB in seconds, whereas the second row shows the deficiency of each heuristic. Each entry is the average of 10 runs.

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

heuristic its deficiency, i.e., 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 random 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 to use than SGM. However, as Table 3 shows, the overheads of KSM and MDM are insignificant when compared to the reductions in the total execution time of the matching algorithm.

As the table shows, MDM is a better heuristic than KSM in terms 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. But, for these graphs, the execution time of BFSB using KSM is less, since KSM is also cheaper than MDM. This is also true when the 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 3, KSM and MDM are considerably better than SGM as a jump-start routine. This is in agreement with previously reported studies [23, 25]. Second, as also observed by Duff [10, p.111] and Mehlhorn and Näher [26], using SGM and not using it, in general, are not much different for 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 1 and discussed in Section 3. For random instances, 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 [26, 31]. Furthermore, previous research states that ABMP performs much better than BFS in most cases [8, 31]. 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 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 [26] and Setubal [31], we may argue that BFSB will find augmenting paths more quickly than DFSB.

Table 4 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 which reduces the runtime of the DFSB from almost 3 hours to 69.3 seconds for some of the cases. Note that 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, 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

Table 4: 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.

	SGM			KSM		
Matrix	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

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). But, 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 paper can be used in the second stage, following the decision by Mehlhorn and Näher [26, p.106], we also implemented the version ABMP-BFS which uses BFSB in the second stage of the algorithm. Table 5 shows the execution times of the algorithms ABMP, ABMP-BFS and BFSB.

As Table 5 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.

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 4 and 5 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 pre-

Table 5: Execution times of algorithms ABMP, ABMP-BFS and BFSB in seconds combined with KSM and MDM. Each entry in the table is the average of 10 runs.

	KSM			MDM		
Matrix	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
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

vious 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 all the experiments with the graphs constructed by RBG-U and RBG-B generators are given in Tables 6 and 7. 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.

As Table 6 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 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 6, 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 [12] 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 7 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 maximum matching. Hence this table does not

Table 6: Execution times of algorithms in seconds combined with the KSM heuristic for RBG-U and RBG-B graphs. Each entry in the table is the average of 10 runs.

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

Table 7: Execution times of algorithms in seconds combined with the MDM heuristic for RBG-U and RBG-B graphs. Each entry in the table is the average of 10 runs.

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

Table 8: 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

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 matching problem is an interesting research area—a comment also made by Cherkassky et al. [8]. 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 8.

Even when the heuristics cannot find a maximum cardinality matching, their deficiencies are usually very small. From the above 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. 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 9 shows this by an example with the matrices amazon0312 (the original matrix) and rajat21 (rowwise and columnwise permuted version).

Tables 10 and 11 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. Firstly, we execute the corresponding algorithm ten times on the original matrix. Secondly, we apply ten random column permutations to the original matrix and execute the algorithm for each column-permuted matrix. Thirdly, we apply ten random row permutations to the original matrix and execute the algorithm for each row-permuted matrix. Lastly, we apply ten random row and column permutations and ex-

Table 9: 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 runs.

Matrix	Heuristic	Deficiency	BFSB	MC21A	PF	PF+	HK	HKDW	ABMP-BFS
amazon0312	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
rajat21	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

ecute 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 any permutation; 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 maximum, even after permuting the matrix. Hence the execution times are small, and the performance of the algorithms 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 efficiently, i.e., 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 10 and 11 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 paper. The algorithms HK and ABMP seem to be robust but the relative performance of HK is worse than HKDW, ABMP and PF. Although PF+ seems to be the most efficient and robust

Table 10: 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 runs. 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 the algorithms for the matrices below and for all 100 matrices are given.

Matrix	Perm	Avg. Def.	BFSB	MC21A	PF	PF+	HK	HKDW	ABMP-BFS
pre2	**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
t2em	**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
tmt_unsym	**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
tmt_sym	**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
apache2	**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
Hamrle3	**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
ecology1	**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
ecology2	**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
amazon0312	**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
language	**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

Table 11: 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 runs. 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 the matrices below and for all 100 matrices are given.

Matrix	Perm	Avg. Def.	BFSB	MC21A	PF	PF+	HK	HKDW	ABMP-BFS
pre2	**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
Hamrle3	**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
rajat21	**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
rajat29	**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
amazon0312	**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
language	**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

algorithm, there are some cases on which it performs much worse than ABMP (see amazon0312).

The difference between the performance of PF (or PF+) and MC21A is huge in favour 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 4, 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 10.

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 which 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 its 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 paper. Although for the RBG-U random matrix class, BFSB is a good alternative, there are algorithms which 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 paper. 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 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.

We mean the worst-case performance, and therefore we rule out the randomized algorithms and analysis.

References

[1] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$. *Information Processing Letters*, 37(4):237–240, 1991.

[2] J. Aronson, A. M. Frieze, and B. Pittel. Maximum matchings in sparse random graphs: Karp-Sipser revisited. *Random Structures and Algorithms*, 12:111–177, 1998.

[3] M. L. Balinski. A competitive (dual) simplex method for the assignment problem. *Math. Program.*, 34(2):125–141, 1986.

[4] M. L. Balinski and J. González. Maximum matching in bipartite graphs via strong spanning trees. *Networks*, 21:165–179, 1991.

[5] H. Bast, K. Mehlhorn, G. Schafer, and H. Tamaki. Matching algorithms are fast in sparse random graphs. *Theory of Computing Systems*, 39(1):3–14, 2006.

[6] C. Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the USA*, 43:842–844, 1957.

[7] P. Chebolu, A. Frieze, and P. Melsted. Finding a maximum matching in a sparse random graph in $O(n)$ expected time. *Journal of the Association for Computing Machinery, Volume 57, Issue 4, April 2010*, 57:1–27, 2010.

[8] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi. Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. *Journal of Experimental Algorithmics*, 3:8, 1998.

[9] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. Number 2 in Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[10] I. S. Duff. On algorithms for obtaining a maximum transversal. *ACM Transactions of Mathematical Software*, 7:315–330, 1981.

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

[12] I. S. Duff and T. Wiberg. Remarks on implementations of $O(n^{1/2}\tau)$ assignment algorithms. *ACM Transactions of Mathematical Software*, 14:267–287, 1988.

[13] M. Dyer, A. Frieze, and B. Pittel. The average performance of the greedy matching algorithm. *Annals of Applied Probability*, 3:526–552, 1993.

[14] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal of Computing*, 4:507–518, 1975.

[15] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261–272, 1995.

[16] Harold N. Gabow and Robert E. Tarjan. Algorithms for two bottleneck optimization problems. *J. Algorithms*, 9(3):411–417, 1988.

[17] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[18] J. González and O. Landaeta. A competitive strong spanning tree algorithm for the maximum bipartite matching problem. *SIAM Journal of Discrete Math.*, 8:186–195, 1995.

[19] F. G. Gustavson. Finding the block lower-triangular form of a sparse matrix. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 275–289. Academic Press, New York and London, 1976.

[20] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, 1973.

[21] R. M. Karp and M. Sipser. Maximum matching in sparse random graphs. In *22nd Annual IEEE Symposium on Foundations of Computer Science (FOCS 1981)*, pages 364–375, Los Alamitos, CA, USA, 1981. IEEE Computer Society.

[22] B. Korte and D. Hausmann. An analysis of the greedy algorithm for independence systems. *Annals of Discrete Mathematics*, 2:65–74, 1978.

[23] J. Langguth, F. Manne, and P. Sanders. Heuristic initialization for bipartite matching problems. *Journal of Experimental Algorithmics*, 15:1.1–1.22, February, 2010.

[24] L. Lovasz and M. D. Plummer. *Matching Theory*. North-Holland mathematics studies. Elsevier Science Publishers, Amsterdam, Netherlands, 1986.

[25] Jakob Magun. Greeding matching algorithms, an experimental study. *Journal of Experimental Algorithmics*, 3:6, 1998.

[26] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 1999.

[27] R. Motwani. Average-case analysis of algorithms for matchings and related problems. *Journal of ACM*, 41:1329–1356, 1994.

[28] Marcin Mucha and Piotr Sankowski. Maximum matchings via Gaussian elimination. In *45th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2004)*, Rome, Italy, 2004. IEEE Computer Society.

[29] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions of Mathematical Software*, 16:303–324, 1990.

[30] J. C. Setubal. New experimental results for bipartite matching. Technical Report DCC-07/92, Univ. of Campinas, November 1992.

[31] J. C. Setubal. Sequential and parallel experimental results with bipartite matching algorithms. Technical Report IC-96-09, Univ. of Campinas, September 1996.