Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique

Experiments on Push-Relabel-based Maximum Cardinality Matching Algorithms for Bipartite Graphs

K. Kaya, J. Langguth, F. Manne and B. Uçar

Technical Report TR/PA/11/33

Publications of the Parallel Algorithms Team

http://www.cerfacs.fr/algor/publications/

Abstract

We report on careful implementations of several push-relabel-based algorithms for solving the problem of finding a maximum cardinality matching in a bipartite graph and compare them with fast augmenting-path-based algorithms. We analyze the algorithms using a common base for all implementations and compare their relative performance and stability on a wide range of graphs. The effect of a set of known initialization heuristics on the performance of matching algorithms is also investigated. Our results identify a variant of the push-relabel algorithm and a variant of the augmenting-path-based algorithm as the fastest with proper initialization heuristics, while the push-relabel based one having a better worst case performance.

1 Introduction

We study algorithms for finding the maximum cardinality matching in bipartite graphs. A subset $\mathcal{M}$ of edges in a graph is called a *matching* if every vertex of the graph is incident to at most one edge in $\mathcal{M}$. The maximum cardinality matching problem asks for a matching $\mathcal{M}$ that contains the maximum number of edges. This problem arises in many applications. For example, finding a transversal, i.e., the problem of obtaining a zero-free diagonal of a sparse matrix after permuting its rows and columns, can be solved via bipartite matching algorithms [12]. Other applications can be found in divers fields such as bioinformatics [3], statistical mechanics [4], and chemical structure analysis [10].

There are many different algorithms for finding a maximum matching in a bipartite graph. In [14], the performance of several augmenting-path-based algorithms was studied on a wide range of instances. In that paper, special attention was given to the effect of different initialization heuristics on the runtimes of these algorithms. The aim of the current paper is to review the push-relabel-based algorithms and compare their performance with the promising augmenting-path approaches identified by Duff et al. [14]. To this end, we implement several push-relabel-based algorithms that essentially cover all established heuristics for and modifications of the original push-relabel algorithm. In addition, we apply several new techniques and evaluate the results.

For the experimental comparison, we use implementations of augmenting-path-based algorithms by Duff et al. [14], and we carefully implement all push-relabel-based algorithms using the same data structures as in [14]. Without this sort of uniformity, comparisons between the algorithms might not be fair, which would render the computational results inconclusive.

Our results indicate that the performance of the different push-relabel-based algorithms varies widely. The fastest versions are superior to the augmenting-path-based algorithms. However, the latter class of algorithms profit strongly from good initialization heuristics, which is sufficient to render them competitive.

The rest of this paper is organized as follows. We briefly discuss the notation and the background in Section 2 and present the main algorithmic idea common to all push-relabel (PR) variants in Section 3. Detailed descriptions of the different techniques used in the PR variants can be found in Section 4. In Section 5, we describe the augmenting-path-based algorithms used for comparison. Starting from Section 6, we describe the experimental setup and the results, along with their discussion and conclusions in Sections 7 and 8. Tables containing detailed experimental results can be found in the appendix.

2 Notation and Background

In a bipartite graph $G = (V_1, V_2, E)$, the vertex sets V_1 and V_2 are disjoint, and each edge in E has one endpoint in V_1 and the other one in V_2. For a vertex $v \in V_1 \cup V_2$, the *neighborhood* of v is defined as $\Gamma(v) = \{u : \{u, v\} \in E\}$. Clearly if $v \in V_1$ then $\Gamma(v) \subseteq V_2$, otherwise, if $v \in V_2$ then $\Gamma(v) \subseteq V_1$.

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 vertex $v \in V$ is *matched* (by $\mathcal{M}$) if it is incident on an edge in $\mathcal{M}$; otherwise, it is *unmatched*. A maximal matching $\mathcal{M}$ is called *maximum* if $|\mathcal{M}| \geq |\mathcal{M}'|$ for every matching $\mathcal{M}'$, where $|\cdot|$ denotes the cardinality of a set. Furthermore, if $|\mathcal{M}| = |V_1| = |V_2|$, then $\mathcal{M}$ is called a *perfect* (complete) matching. A perfect matching $\mathcal{M}$ is maximum, and each vertex in V is incident to exactly one edge in $\mathcal{M}$. Clearly not all bipartite graphs have a perfect matching. The *deficiency* of a matching $\mathcal{M}$ is the difference between the cardinality of a maximum matching and $|\mathcal{M}|$. A good discussion on matching theory can be found in Lovasz and Plummer's book [21].

An important application, which forms our motivation, of the maximum cardinality matching on the bipartite graphs problem arises in sparse matrix computations. For a given $m \times n$ matrix $\mathbf{A}$, we define $G_{\mathbf{A}} = (V_R, V_C, E)$ where $|V_R| = m$, $|V_C| = n$, and $E = \{\{v_i, v_j\},\ v_i \in V_R,\ v_j \in V_C : a_{ij} \neq 0\}$ as the bipartite graph derived from $\mathbf{A}$. In this respect, the transversals of a square matrix $\mathbf{A}$ correspond to the perfect matchings of $G_{\mathbf{A}}$. Based on this important correspondence, we adopt the term *column* for a vertex in V_C and *row* for a vertex in V_R, maintaining consistency with the notation in [14]. The number of edges in $G_{\mathbf{A}}$ is equal to the number of nonzeros in $\mathbf{A}$, and it is denoted by τ.

2.1 Bipartite Cardinality Matching Algorithms

Let $\mathcal{M}$ be a matching in G. A path in G is $\mathcal{M}$-*alternating* if its consecutive 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 by Berge [5] forms the basis for all augmenting-path-based algorithms for the maximum matching problem.

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

In this paper, we use three augmenting-path-based matching algorithms. The first one, `PFP`, is a variant of the algorithm of Pothen and Fan [24]. It was presented in [14]. The second algorithm, `HKDW` [15], is a variant of the algorithm proposed by Hopcroft and Karp [17]. The third one, `ABMP`, was introduced by Alt et al. [1].

Algorithms based on augmenting paths uses the following approach. Given a possibly empty matching $\mathcal{M}$, this class of algorithms searches for an $\mathcal{M}$-augmenting path $\mathcal{P}$. If none exists, then the matching $\mathcal{M}$ is maximum by the theorem above. Otherwise, the alternating path $\mathcal{P}$ is used to increase the cardinality of $\mathcal{M}$ by setting $\mathcal{M} = \mathcal{M} \oplus E(\mathcal{P})$ where $E(\mathcal{P})$ is the edge set of a path $\mathcal{P}$, and $M \oplus E(\mathcal{P}) = (\mathcal{M} \cup E(\mathcal{P})) \setminus (\mathcal{M} \cap E(\mathcal{P}))$ is the symmetric difference. This inverts the membership in $\mathcal{M}$ for all edges of $\mathcal{P}$, and since both the first and the last edge of $\mathcal{P}$ were unmatched in $\mathcal{M}$, $|\mathcal{M} \oplus E(\mathcal{P})| = |\mathcal{M}|+1$. The way in which augmenting paths are found constitutes the main difference between the algorithms based on augmenting-path search, both in theory and in practice.

Push-relabel algorithms on the other hand search and augment together. They do not explicitly construct augmenting paths. Instead, they repeatedly augment the prefix of a speculative augmenting path $\mathcal{P}_2 = (v, u, w)$ in G where u is matched to w and $v \in V_C$ is an unmatched column. Augmentations are performed by unmatching w and matching v to u. If the neighbor of an unmatched column is also unmatched, the suffix of an augmenting path has been found, allowing the augmentation of $|\mathcal{M}|$. This operation is repeated until no further suffixes can be found. The speculative augmentations are guided by assigning a label ψ to every vertex which provides an estimate of the distance to the nearest potential suffix.

The original push-relabel algorithm by Goldberg and Tarjan [16] was designed for the maximum flow problem. Since bipartite matching is a special case of maximum flow, it can be solved by pushing and relabeling.

In fact, it is one of the fastest algorithms for bipartite matching, as was shown in [7]. In this paper, we study the performance of several variants of the push-relabel algorithm and compare these to the best augmenting path algorithms available. We will discuss the simple push-relabel algorithm (`PR`) and its extensions in detail in the next two section.

For a short summary about other algorithms and approaches for the bipartite graph matching problem, we refer the reader to Section 3.4 of [14].

2.2 Initialization Heuristics

Both push-relabel and augmenting-path-based algorithms start with an empty matching and find matchings of successively increasing size by exploiting augmenting paths. Thus, these algorithms can be initiated with a non-empty matching. In order to exploit this, several efficient and effective heuristics, which find considerably large initial matchings, have been proposed in the literature [18, 20, 22].

In this paper, we use three initialization heuristics. The first one, which we call *simple greedy matching* (`SGM`), examines each unmatched column $v \in V_C$ in turn and matches it with an unmatched row $u \in \Gamma(v)$, if such a row exists. Although it is the simplest heuristic in the literature, `SGM` is probably the most frequently used one in practice. The second heuristic is `KSM` [18]. It is similar to `SGM`, but it keeps track of the vertices with a single unmatched vertex in their neighborhood and immediately matches these with their neighbors. Theoretical studies [2, 18] show that `KSM` is highly likely to find perfect matchings in random graphs, and in practice, it is significantly more effective than `SGM`. The last heuristic, `MDM`, is the minimum-degree-matching heuristic which always matches a vertex having a minimum number of unmatched neighbors with a neighbor that again has a minimum number of unmatched neighbors. These heuristics are experimentally investigated in [14, 20, 22]. The reader can also find the extended versions of these heuristics in [20, 22]. Following a remark in [7], we also initialize the algorithms with an empty matching denoted as `NONE` for comparison.

For `SGM`, the name simple greedy matching can be misleading, since all heuristics presented here pursue a comparatively simple greedy strategy. However, while `KSM` and `MDM` greedily pick the vertices by using partial information and update the information accordingly, `SGM` acts completely uninformed. As previous experiments show [14, 20, 22], `SGM` performs significantly worse than `KSM` and `MDM`. For these reasons, in this paper, we call `KSM` and `MDM` *elaborate* initializations whereas `SGM` and `NONE` are called *basic*.

Note that all three heuristics have a time complexity of $\mathcal{O}(\tau)$. However, since `SGM` does not use any information, it runs significantly faster than `KSM` in practice. `KSM` is a simplified version of `MDM`, and it is generally faster.

3 The Push-Relabel Matching Algorithm

The push-relabel algorithm described in [16] was originally designed for the maximum flow problem. Since the bipartite matching problem is a special case of the maximum flow problem, we use a simplified version of the push-relabel algorithm similar to the one described in [7]. This simplified version is given in Algorithm 1 and will be denoted as `PR`. Section 4 contains several extensions of `PR` which were used in the experiments.

Algorithm 1 `PR`: Push-relabel algorithm for the bipartite matching problem

Input: A bipartite graph $G = (V_R, V_C, E)$ with a (possibly empty) matching $\mathcal{M}$

Output: A maximum cardinality matching $\mathcal{M}^\star$

```
Set ψ(u) = 0 ∀u ∈ V_R
Set ψ(v) = 1 ∀v ∈ V_C
Set all v ∈ V_C unmatched by M active
while an active column v exists do
    Find row u ∈ Γ(v) of minimum ψ(u)
    if ψ(u) < 2n then
        ψ(v) ← ψ(u)+1        ▶ Relabels v if {u, v} is not an admissible edge
        if {u, w} ∈ M then
            M ← M\{u, w}     ▶ Double push
            Set w active
        M ← M∪{u, v}         ▶ Push
        ψ(u) ← ψ(u)+2        ▶ Relabel u to obtain admissible incident edge
    Set v inactive
return M* = M
```

Let $\psi : V_R \cup V_C \rightarrow \mathbb{N}$ be a distance labeling used to estimate the distance and thereby the direction of the closest free row for each vertex. This labeling constitutes a lower bound on the length of an alternating path from a vertex v to the next free row. Note that if v is a free column, such a path is also an augmenting path. During initialization, the algorithm sets $\psi(v) = 1$ $\forall v \in V_C$ and $\psi(v) = 0$ $\forall v \in V_R$. Now, as long as there are free columns, the algorithm repeatedly selects one of them and performs the *push* operation on it.

To perform a push on a free column v, we search $\Gamma(v)$ for a row $u \in \Gamma(v)$ with minimum $\psi(u)$. Since $\psi(u) = 0$ and $\psi(v) = 1$ after the initialization, the minimum value for $\psi(u)$ is $\psi(v) - 1$. This relation is maintained throughout the algorithm as an invariant. Thus, as soon as an edge $\{v, u\}$ having $\psi(v) = \psi(u) + 1$ is found, the search stops. Such an edge is called *admissible*.

If u is unmatched it can be matched to v immediately by adding $\{v, u\}$ to $\mathcal{M}$ and thereby increasing the cardinality of $\mathcal{M}$ by one. This is called a *single push*. On the other hand, if u is matched to w we perform a *double push* by removing $\{w, u\}$ from $\mathcal{M}$ and then adding $\{v, u\}$ to $\mathcal{M}$, thereby matching v to u and making w active. This ensures that once a row is matched, it can never become unmatched again. Thus, the cardinality of $\mathcal{M}$ never decreases. Note that $\psi(u) = 0$ for any unmatched row u, i.e., such a row will always have minimum ψ.

If there is no admissible row u among the neighbors of v, i.e., the row u having minimum $\psi(u)$ has $\psi(u) > \psi(v) - 1$, we set $\psi(v)$ to $\psi(u) + 1$ (in case $\psi(u) \leq 2n$). This is referred to as a *relabel* on v. Clearly, doing so does not violate the above invariant due to the minimality of $\psi(u)$. To understand the motivation for a relabel on v, remember that $\psi(u)$ is a lower bound on the length of an alternating path from u to a closest unmatched row. Now, even though no path between v and its closest unmatched row necessarily contains u, such a path must contain a $u' \in \Gamma(v)$. Since $\psi(u)$ was minimum among the labels of all the neighbors of v, we have $\psi(u') \geq \psi(u)$. Thus, $\psi(u) + 1$ is a lower bound on the length of a path between v and the closest unmatched row, and $\psi(v)$ is updated accordingly.

By the same token, u is relabeled by increasing $\psi(u)$ by 2 following a push. For a single push, this means that we have $\psi(u) = 2$ now. Since G is bipartite and u is no longer an unmatched row, it is clear that the distance to the next unmatched row must be at least 2 after a single push. In case of a double push, consider that any alternating path from u to a closest unmatched row now contains v, since such a path starts with an unmatched edge on the unmatched row and as G is bipartite, the path contains only matched edges going from columns to rows and only unmatched edges from rows to columns. Thus, the actual distance for u must be at least $\psi(v) + 1$. Because $\psi(v)$ was either relabeled to $\psi(u) + 1$ prior to the push or had this value to begin with, increasing $\psi(u)$ by 2 yields a correct new lower bound. Note that this increase will not violate the invariant $\psi(u) \geq \psi(v) - 1$.

When implementing the push-relabel algorithm, we can eschew storing the row labels, since $\psi(u)$ will always be either 0 if u is unmatched or equal to $\psi(w) + 1$ if u is matched to w.

We refer to free columns as active vertices. If $\psi(u) \geq 2n$ for the minimum $\psi(u)$ among the neighbors of v, instead of performing a push, v is considered unmatchable and marked as inactive. As it is unmatched, it can never become active again via a double push. Thus, it will not be considered any further by the algorithm.

The push and relabel operations are repeated until there are no active vertices left, either because they have been matched or because they were marked as inactive. Since the maximum length of any augmenting path in G is bounded by $2n$, no augmenting path that starts at an inactive column v can exist in $\mathcal{M}$ at that time because ψ is a lower bound on the length of a path to an unmatched row and $\psi(u) \geq 2n$ for all neighbors of v. Using Theorem 2.1, it is easy to show that in this case $\mathcal{M}$ is a maximum matching. The time complexity of the algorithm is $\mathcal{O}(n\tau)$ [16].

4 Modifications to the Push-Relabel Algorithm

We now consider several modifications to the push-relabel algorithm described in Section 3 in order to optimize its performance. The modifications include applying a strict order of push operations and heuristics that optimize the distance labeling ψ. Both are well studied in the literature [7, 19]. In addition, we experiment with new techniques inspired by the augmenting-path algorithms.

4.1 Push Order

The push-relabel algorithm repeatedly selects an active column on which it performs a push operation, but the order in which active columns are selected is not fixed. Any implementation needs to define a rule according to which active columns are selected for pushing. A simple solution for this is to maintain a stack or queue of active columns and select the first or topmost element, resulting in `LIFO` (last-in-first-out) or `FIFO` (first-in-first-out) push order. Alternatively, each active column v can be sorted into a priority queue ordered by its label value $\psi(v)$. Maintaining the priority queue costs some extra effort, but it allows processing the active columns in ascending or descending dynamic order of their labels. In this study, we compare the performance of all four approaches. In the results section, the codes using `LIFO`-, `FIFO`-, and highest- and lowest-label-first order are denoted as `PR-LIFO`, `PR-FIFO`, `PR-High`, and `PR-Low`, respectively.

4.2 Global Relabeling

The performance of the `PR` algorithm can be improved by setting all labels to exact distances. This is done by periodically running a BFS starting from the unmatched rows. The label of each vertex v visited by the BFS is set to the minimum distance from v to any unmatched row. Each vertex w not visited by the BFS is assigned a label $\psi(w) = 2n$, thereby removing it from further consideration. Executing this BFS once per $\mathcal{O}(n)$ pushes, in which a column label was changed, improves the algorithms runtime complexity to $\mathcal{O}(\sqrt{n}\tau)$ [19]. It is a well-established fact that the global relabeling operations are essential for practical performance, and preliminary tests confirmed this. Thus, all our `PR` codes use periodic global relabeling. Following [7], we use n as the standard frequency of global relabels. For rectangular matrices, this becomes $(m+n)/2$, but for simplicity, we will denote it as n in the text. Furthermore, preliminary experiments showed no noticeable difference between using relabeling frequencies of m and n.

We study the effect of the frequency of the global relabeling by executing a BFS once per every $n/2^k$, $k \in \{0, 1, 2, 3\}$ pushes in which a column label is changed. All codes that do not use the standard relabeling frequency have the appropriate frequency included in their names. For example, assuming `RF` is the standard frequency n, `RF/2` indicates a frequency of $n/2$. Since the higher frequencies yielded interesting results, we also implemented a `PR-FIFO` variant using the frequency of `3RF/4`.

4.3 Gap Relabeling

If, at any time during the execution of the algorithm, there is no vertex having a label of k, all vertices v with $\psi(v) > k$ can be removed from consideration by setting $\psi(v) = 2n$. This heuristic, called gap relabeling was proposed independently in [6] and in [9]. It allows the algorithm to quickly skip over subgraphs for which a non-perfect maximum matching has been found. However, a global relabeling has the same effect, since it sets the labels of all unreachable vertices to $2n$. Therefore, the benefit from the gap relabeling operation is limited, and it diminishes with the increasing frequency of global relabelings.

We do not use gap relabeling in this study for multiple reasons. First, it requires more elaborate data structures. For the `High` and `Low` push orders, a priority queue of inactive vertices must be maintained in addition to the queue of active vertices. For `LIFO` and `FIFO` orders, both queues must be added. Second, it affects primarily the `LIFO` and `High` orders that were

identified as inferior before [7] and also in our preliminary experiments. Finally, in [7], the `PR-Low` variant using the gap relabeling was found to be equivalent to the `PR-FIFO` variant without the gap relabeling operation. Therefore, we do not include gap relabeling in the current study.

4.4 Fairness

By default, our `PR` implementations always search through adjacency lists in the same order when selecting a neighbor of minimum ψ. This raises the question whether the algorithm could be improved by encouraging fairness in neighbor selection. In [14], this was proposed for improving the Pothen and Fan (`PF`) algorithm [24], which is discussed in Section 5.1. By varying the direction of search through the adjacency list for selecting a neighbor of an active column, the likelihood of the algorithm repeatedly pursuing an unpromising direction of search is reduced. For the `PF` algorithm, this technique resulted in a significant performance gain. It can also be applied in the `PR` algorithm during the neighbor selection process. However, as the labels already suggest which directions are promising and should be further pursued, the fairness heuristic is less likely to produce noticeable improvements.

In addition to providing a more even distribution of the neighbor selected, this technique can also reduce the time spent on neighbor selection. During the execution of the push operation on an active column, the push-relabel algorithm needs to search for an adjacent row with the minimum label. If an admissible edge $\{v, u\}$ is found, the search stops because no neighbor can have a lower label than $\psi(u)$. Varying the direction of search increases the likelihood of finding such an edge early. Following [14], this technique is called fairness, and the `PR` codes using it are marked with `Fair` in their names.

4.5 Multistage Algorithms

The `ABMP` algorithm (summarized later in Section 5.3) introduced the idea of switching from a label guided search to a pure augmenting-path search once the algorithm nears completion. The aim of this technique is to find the few remaining long augmenting paths quickly. As described in [1], `ABMP` uses a lowest label guided approach similar to the push-relabel algorithm in the first stage and continues with the Hopcroft-Karp algorithm in the second stage, achieving an improved worst-case runtime of $\mathcal{O}(n^{1.5}\sqrt{\tau/\log(n)})$.

For practical purposes, Mehlhorn and Näher [23] suggested replacing

the Hopcroft-Karp algorithm of the second stage with a breath-first search for augmenting paths. In this paper, we replace the algorithm for the first stage with PR-FIFO and PR-Low, which were the best PR variants identified in our preliminary experiments. In the second stage, we switch to the PFP algorithm (which we describe in Section 5.1) since [14] identified it to be the best augmenting-path-based algorithm for most of the cases.

We stop the push-relabel phase after 99.9% of the columns have been matched and, for the PR-Low variant, we also stop as soon as the current lowest label reaches $0.1\sqrt{n}$ as suggested in [23]. Both codes of this type use fairness, and they are denoted as PR-FIFO+Fair+PFP and PR-Low+Fair+PFP.

5 Augmenting-path-based Algorithms

In this section, we briefly describe the augmenting-path-based algorithms used for experimental comparison. For a more detailed description, we refer the reader to [1, 14, 17, 23, 24]. Two of the algorithms, ABMP and HK are well studied in the literature. They have good worst case runtimes, and their high performance in practice has been confirmed in many studies. The third algorithm, PFP, is a modification of the algorithm by Pothen and Fan [24], and was introduced in [14]. Its runtime complexity $\mathcal{O}(n\tau)$ is inferior to that of ABMP, HK, and PR with global relabeling (see Section 4.2), but it outperformed most of the augmenting-path-based algorithms in the experiments described in [14]. An experimental comparison between PFP and the best PR codes is a principal objective of this study.

Note that PR, especially with global relabeling, does indeed find augmenting paths and can augment along such paths. However, it can also perform pushes, i.e., speculative augmentations that might not follow a valid augmenting path and have to be undone later. In contrast, the algorithms presented in this section only augment if an entire valid augmenting path has been discovered. Therefore, these algorithms are referred to as *"augmenting-path-based"* in the literature.

5.1 PFP: A Modification of Pothen and Fan's Algorithm

The Pothen-Fan algorithm [24], denoted as PF, is based on repeated phases of depth-first searches (DFSs). Each PF phase performs a maximal set of vertex disjoint DFSs each starting from a different unmatched column. A vertex can only be visited by one DFS during a phase. Any DFS that succeeds in finding an unmatched row immediately suggests an augmenting path. As soon as all the searches have terminated, the matching $\mathcal{M}$ is augmented

along all the augmenting paths found in this manner. A new phase starts after those augmentations take place.

As long as there is any $\mathcal{M}$-augmenting path in G, at least one is found during each phase. When a phase finishes without finding such an augmenting path, the algorithm terminates. It also stops if no free columns remain after performing the augmentations. Clearly, the maximum number of phases is n, and each phase can be performed in $\mathcal{O}(\tau)$ time, giving the algorithm a runtime of $\mathcal{O}(n\tau)$.

Note that in each DFS 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, a pure DFS-based algorithm may need to explore a large part of the graph. To alleviate this problem, a mechanism called *lookahead* is used [12, 24]. This works as follows. When a column v is visited, the algorithm first checks if v has an unmatched row u in its adjacency list, and if there is one, it uses the corresponding augmenting path. Otherwise, it continues with the usual DFS process.

The above algorithm using the *lookahead* technique is known as Pothen and Fan's algorithm. In [14], the algorithm was found to be efficient for matrices arising in real world applications, except that it is very sensitive to row and column permutations. To alleviate this, they suggested to modify the order of visiting the rows in the adjacency lists of columns by applying an alternating scheme. This is done by counting phases and traversing the adjacency list of a column from left to right in each odd numbered phase. During an even numbered phase, the adjacency lists are traversed in 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, which is called *fairness*, 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. Note that fairness does not change the complexity of `PF`, and the memory requirements are exactly the same in both algorithms. It 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 [14]. In fact, the success of this technique suggested using it in the `PR` codes, as discussed in Section 4.4. Since the modified `PF` algorithm is superior to the original one, we use it exclusively. The systematic name of the algorithm would be `PF+Fair`, but we shorten it and refer the algorithm as `PFP` to be consistent with the names used by Duff et al. [14].

5.2 HKDW: A variant of the Hopcroft-Karp Algorithm

Like PFP, the HK algorithm also organizes the searches for augmenting paths into phases [17]. In each phase, it uses a maximal set of vertex-disjoint shortest augmenting paths in the graph, and augments the matching. A HK phase consists of two parts. It first initiates a *combined BFS*, which is similar to the BFS used in the global relabeling for PR, from all unmatched columns to assign level values to the vertices. The unmatched columns are initialized as level zero. For an even level ℓ, the next level $\ell + 1$ contains all vertices reachable from level ℓ via edges not in the current matching, and for an odd level, the next level contains all vertices reachable via matching edges. This ensures that the BFS progresses along augmenting paths. The search continues until an odd level L containing at least one unmatched row is discovered. It stops after assigning a level value of L to all vertices in that level. Note that L is then the length of the shortest augmenting path in the graph. If no unmatched row is found, the algorithm terminates.

The second part of an HK phase uses the level values computed in the first part to find augmenting paths. A restricted DFS is initiated from each unmatched row in level L. The DFSs are in the reverse direction, i.e., from the unmatched rows in level L to the unmatched columns in level 0. The successor of a vertex of level ℓ in the restricted DFS must be of level $\ell - 1$. Furthermore, no vertex may be visited more than once, ensuring that the DFSs (and hence the augmenting paths) are vertex disjoint. Therefore, they will find a maximal set of vertex disjoint augmenting paths of length L in the graph. These are then used to augment the matching. Except for the level restriction of the DFSs, the second part of a HK phase is similar to a PFP phase where each vertex is also visited only once. After the augmentations, a new phase begins.

Hopcroft and Karp proved that a maximum matching is obtained after at most $\mathcal{O}(\sqrt{n})$ phases. Since the complexity of a HK phase is $\mathcal{O}(\tau)$, the runtime complexity of the overall algorithm is $\mathcal{O}(\sqrt{n}\tau)$.

Duff and Wiberg [15] 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. As this seems to be a waste of resources, they proposed to increase the number of augmentations in each phase by using extra DFSs from the remaining unmatched rows. Similar to original HK, the modified version starts with a combined BFS up to the highest level L containing an unmatched row and uses restricted DFSs from unmatched rows in level L to find a maximal set of vertex disjoint augmenting paths of length L in the graph and augments along these paths. Then, in an additional third

part, more DFSs are initiated from other unmatched rows. These additional DFSs are not restricted by level, and can use all the edges in the graph. However, they must follow alternating paths, and still, no vertex can be visited more than once per phase, similar to PFP. Hence the augmenting paths found by additional DFSs are still vertex disjoint among themselves and with the restricted DFSs from part two. In [14], this variant was found to have better performance than the standard HK algorithm. Therefore, we will use this variant for comparison. We refer to it as HKDW in the remainder of the paper.

5.3 ABMP: Alt et al.'s Algorithm

The algorithm by Alt et al [1] incorporates some techniques used in the original push-relabel algorithm [16] into the Hopcroft-Karp algorithm. It runs in two consecutive stages: In the first stage, a set of augmentations is 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) with the aid of level values that constitute lower bounds 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, much like the distance labels ψ in the push-relabel algorithm.

During initialization, each column and row is assigned a level value of 0 and 1, respectively. At all times, the level values of the rows are even and those of the columns are odd. Furthermore, all the unmatched columns are in levels L and $L + 2$ where $L \geq 1$ is an integer increasing over the course of the first stage. Thus, the pattern of progress in ABMP resembles to that of PR-Low (see Section 4.1).

In the first stage, a DFS is initiated from each unmatched column v in level L. These DFSs use the level information such that after visiting a level ℓ vertex w, the algorithm tries to *"advance"* the augmenting path to an adjacent vertex in level $\ell - 1$. If this is not possible, the algorithm

"retreats" to the predecessor of w on the path and attempts to advance to a different neighbor. If an unmatched row u is discovered, an augmenting path from v to u has been found, and the augmentation is performed. This is referred to as a *"breakthrough"*. On the other hand, if the path retreats from all neighbors of v, the level of v is increased by 2 and a DFS for a different column in level L begins. This DFS search behaves similarly to a LIFO push order in PR, except that it cannot relabel any vertices other than v. If no vertices of level L remain, L is increased by 2 and new DFSs are started from the unmatched columns in the new level L. The first stage continues until L, i.e., 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 5.2. 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.

Alt et al. [1] proved that maintaining the level information dynamically up to level $L = \sqrt{\tau \log n/n}$ is cheaper than the BFS plus DFS approach of HK in terms of time complexity. With this bound on L, the time complexity of ABMP becomes $\mathcal{O}\left(\min\left(\sqrt{n}\tau, n^{1.5}\sqrt{\tau/\log n}\right)\right)$.

In our implementation, as suggested in [23], the first stage continues until $L > 0.1\sqrt{n}$, or $50L$ is greater than the number of unmatched columns.

Note that the initial levels in the first stage are exact distances 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 as the initial level values makes the algorithm slow since, for the DFSs from an unmatched column v, the search will be unsuccessful until the level attribute of c is equal to the shortest augmenting-path length for v. Note that during the course of the first stage, the level attributes are not exact, i.e., they are only lower bounds in the same manner as the labels in PR. 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. Such an update scheme may be time consuming when the difference between the lower bounds and exact values are large. To avoid this problem, as suggested by Setubal [25], we periodically execute a global update procedure which makes the lower bounds exact. This global update procedure is essentially identical to the global relabelings in PR (see Section 4.2) and similar to the combined BFS part of HK. The difference compared to the combined BFS part is that

it does not stop after the first level containing an unmatched column has been found. Instead, it continues the update procedure as described in the previous paragraph until level L has been reached. The global update procedure is run at the beginning once and then rerun after a total number of n level updates have been performed since the last run.

The similarities between `ABMP` and `PR` prompted the question whether it is worthwhile to use a two stage approach where `PR` is combined with a simple augmenting-path algorithm in order to obtain a faster combined algorithm. We have therefore added such a code to study this experimentally (see Section 4.5).

6 Experimental Setup

We implemented all of the algorithms and heuristics in the C programming language and compiled them with `gcc` version 4.4.1. We perform experiments by combining the discussed algorithms with the initialization heuristics summarized in Section 2.2. We perform experiments with these combinations on a large set of bipartite graphs which include both randomly generated and real-world instances. We also report on the stability of the algorithms over the entire data set. All of the experiments were conducted on an Intel 2.4Ghz Quad Core computer, equipped with 4 GB RAM and Linux operating system.

6.1 Data Set

We have two disjoint sets of data. The first one contains randomly generated bipartite graph instances corresponding to random square sparse matrices. The second set contains instances corresponding to sparse matrices from real-world applications. These two sets are referred to as random instances and matrix instances in the following.

To generate the random instances, we used `sprand` command of Matlab (the random matrix generator function) and created $n \times n$ sparse matrices, and then used the associated bipartite graphs. As input arguments, `sprand` takes n, which is the number of rows and columns, and a value d which is the probability of an entry in the matrix being nonzero. That is, each of the n^2 potential nonzeros is present in the random matrix with the same probability d. For our experiments, we used $n = 10^6$ and $d \in \{k \times 10^{-6} : 1 \leq k \leq 10\}$. Hence, k, the expected number of nonzeros in a row and column is between 1 and 10. A similar generator is also used in [22].

For each k, we first create ten random instances and execute all of the algorithms on these matrices. We then compute the average of the ten runtimes for each algorithm and use it as the runtime of that algorithm for the random instance generated by using the parameter k.

We chose a set of real-world matrices from the University of Florida Sparse Matrix Collection [8] and used the associated bipartite graphs to build the matrix instances data set. The chosen matrices satisfy the following properties: $m \times n$ where $\min(m, n) \geq 10^5$, $\max(m, n) \leq 2 \times 10^6$; have τ nonzeros where $\tau \leq 8 \times 10^6$—here again m, n, τ corresponds to the number of rows, columns, and the nonzeros for a given matrix. In order to avoid a bias or skew in the data set, we choose at most 16 matrices from each matrix group. At the time of experimentations, a total of 157 matrices satisfied these assertions, where 33 of them were rectangular. A complete list of the names of the matrices and their groups are given in Table 8 in the appendix.

For each real-world matrix, we performed four sets of experiments. Firstly, we execute all algorithms with the different heuristics on the original matrix. Secondly, we apply ten random column permutations to the original matrix and execute the algorithms for each column-permuted matrix. Thirdly, we apply ten random row permutations to the original matrix and execute the algorithms for each row-permuted matrix. Lastly, we apply ten random row and column permutations and execute the algorithms for each totally permuted matrix. For each algorithm, the average of the running time on each of those groups of ten permutations is stored as the running time of the algorithm on a matrix instance with a given permutation type.

The matrices in the experiments are initially stored in CCS (compressed-column-storage) format. This gives direct access to the adjacency lists of the columns. Due to global relabeling, the `PR` algorithms also require accessing the adjacency lists of the rows, and therefore a CRS (compressed-row-storage) data structure must also be provided. These structures are constructed during the course of the algorithm and hence the reported runtimes include the associated overhead. The `KSM` and `MDM` initializations, as well as the `ABMP` and `HKDW` algorithms also require CRS. For more information on these storage formats, we refer the reader to [13, Section 2.7].

7 Experimental Results

In this section, we present the most important experimental results. Further tables containing detailed results are found in the appendix.

Table 1: The performance for the highest-label first approach compared to that of lowest-label first. Matrix and Random are the average performance values for the matrix and random instances, respectively. Average is the average runtime over both experiments in seconds, and Ratio is the factor between an average value and the best average value in this table. It is interesting to note that `MDM` initialization works much better than `KSM` for `PR-High`, but `PR-Low` is still about three times faster.

Algorithm	Heur.	No perm.	Row perm.	Column perm.	Row and col. perm.	Matrix	Random	Average	Ratio
`PR-High`	`KSM`	26.86	32.17	9.35	9.27	19.41	1.29	10.35	9.38
	`MDM`	6.11	4.75	4.20	4.39	4.86	1.62	3.24	2.94
`PR-High`	`KSM`	25.86	31.28	8.77	9.11	18.76	1.30	10.03	9.08
`+Fair`	`MDM`	6.58	4.68	4.10	4.33	4.92	1.61	3.27	2.96
`PR-Low`	`KSM`	1.08	1.19	1.33	1.23	1.21	1.00	1.10	1.00
`+Fair`	`MDM`	2.14	1.30	1.20	1.37	1.50	1.26	1.38	1.25

7.1 Preliminary Experiments on `PR-High`

Preliminary experiments with the `PR-High` variants showed that they often require very high runtimes. Therefore, due to the large number of instances, we ran `PR-High` only on a reduced set of instances. The results from this experiment are shown in Tables 1 and 2. Clearly, `PR-High` is not competitive with faster codes such as `PR-Low`. This matches corresponding findings in [7]. Therefore, implementing the `PR-High` variant is not recommended.

Conceptually, working with the highest labels first means searching for augmenting paths among those vertices where such a search has had little success so far, since high labels are a result of repeated pushes from the same vertex. Thus, it is likely that some of the work done is futile, since it will be superseded by more successful searches that will happen later. Note that unlike the implementation in [7], we do not use gap relabeling. Still, we obtain a similar result.

7.2 Overview of the Main Results

Due to the fact that the performance of all algorithms varies largely over the instances, it is difficult to identify a single fastest algorithm. To evaluate overall performance $\overline{\rho}(A)$ of an algorithm A, let $\rho_I(A) = t_I(A)/t_I(A_I^*)$ be the runtime factor of A where $t_I(A)$ is the average runtime taken by algorithm A on instance I, and A_I^* is the fastest algorithm for this instance. We compute averages $\overline{\rho}(A)_R$ and $\overline{\rho}(A)_M$ for the random and real-world matrix instances, respectively. To obtain $\overline{\rho}(A)$, we simply take the average over

Table 2: The highest (Worst) and the average of the five highest (Avg. 5) runtimes in seconds on the matrix instances in all four permutation schemes for the highest-label first and lowest-label first approaches. Clearly, PR-High is much more likely to give very high runtimes compared to PR-Low or other PR codes.

		No perm.		Row perm.		Column perm.		Row and col. perm.	
Algorithm	Heur.	Worst	Avg. 5	Worst	Avg. 5	Worst	Avg. 5	Worst	Avg. 5
PR-High	KSM	219.93	152.47	304.77	243.02	246.73	141.28	330.00	169.94
	MDM	119.58	52.60	121.33	59.77	121.95	59.88	123.73	69.10
PR-High	KSM	215.92	147.67	300.40	236.29	240.69	138.07	320.30	166.61
+Fair	MDM	116.68	51.35	118.83	58.74	119.44	58.30	121.01	67.90
PR-Low	KSM	1.15	0.81	6.70	3.41	15.77	5.68	8.11	5.89
+Fair	MDM	1.47	1.12	21.11	6.12	4.02	2.93	16.56	6.30

$\overline{\rho}(A)_R$ and $\overline{\rho}(A)_M$. This means that both the random and matrix data sets have the same weight, even though the latter contains more instances. Note that an algorithm A refers to a combination of an exact algorithm and an initialization heuristic. Table 3 gives an overview of the averages $\overline{\rho}(A)_R$ and $\overline{\rho}(A)_M$, while $\overline{\rho}(A)$ is given in Table 4. More detailed results can be found in Tables 5 and 6 in the appendix.

For the matrix instances, PFP using the basic initialization SGM is clearly superior to all alternatives. The HKDW, ABMP, and PR-LIFO algorithms perform best using MDM initialization, while most other PR variants work better using SGM. For PR-FIFO with the highest relabeling frequency, using no initialization is slightly better than SGM.

For the random instances, KSM clearly dominates all other initializations. Using KSM, all algorithms are close, but the augmenting-path algorithms show slightly better performance, even though the best PR codes are faster than PFP. For MDM, the situation is similar, although the runtimes are approximately 60% higher. With the basic initialization heuristics NONE and SGM, all PR codes, except LIFO, are superior to augmenting-path algorithms. This is consistent with previous studies such as [7], which did not include elaborate initializations.

7.3 Stability

In several previous studies, it was observed that almost all matching instances were "easy", i.e., the average performance of all algorithms was always much better than their worst case runtimes would suggest. Thus, average performance does not provide a good guideline of performance on the most difficult instances one might encounter.

Table 3: Average relative runtimes in the matrix and random instances for all combinations of heuristics and algorithms.

	Matrix				Random			
Algorithm	NONE	SGM	KSM	MDM	NONE	SGM	KSM	MDM
PFP	1.80	1.63	3.92	6.35	7.27	7.21	1.36	2.13
HKDW	16.55	11.93	8.69	6.57	13.16	16.36	1.29	2.07
ABMP	14.89	9.88	9.10	7.96	4.97	3.71	1.28	2.02
PR-LIFO	352.56	61.08	10.20	7.72	29.43	16.69	1.40	2.09
PR-FIFO	5.18	3.41	4.37	7.37	1.69	1.67	1.40	2.10
PR-Low	49.19	5.16	4.96	7.89	2.40	2.14	1.55	2.21
PR-LIFO+Fair	413.50	61.07	9.99	8.61	29.77	16.81	1.39	2.09
PR-FIFO+Fair	8.45	3.21	4.15	6.96	1.63	1.71	1.40	2.10
PR-Low+Fair	26.80	4.86	4.72	6.86	2.46	2.17	1.54	2.21
PR-FIFO+Fair+RF/4	3.31	3.34	4.28	6.34	1.45	1.46	1.31	2.06
PR-FIFO+Fair+RF/2	3.94	3.21	4.13	6.36	1.42	1.49	1.32	2.05
PR-FIFO+Fair+3RF/4	5.29	3.18	4.11	6.74	1.52	1.61	1.37	2.08
PR-FIFO+Fair+2RF	12.53	3.63	4.71	7.43	2.13	2.05	1.56	2.18
PR-Low+FairRF/2	14.06	4.00	4.69	6.71	2.09	1.91	1.43	2.15
PR-Low+Fair+PFP	13.35	4.09	4.49	6.84	2.59	2.16	1.62	2.26
PR-FIFO+Fair+PFP	3.86	3.48	4.40	6.59	3.19	3.11	1.60	2.32
PR-FIFO+Fair+PFP+RF/2	3.89	3.47	4.41	6.58	3.23	3.05	1.60	2.32

In order to obtain an estimate on how much case-by-case runtimes deviate from average runtimes, we compute the standard deviation $\sigma(A)$ from the average runtime $\overline{t}(A)$ over both experiments. Again, both experiments with the random and matrix instances are weighted equally. This gives us a statistical estimate on how the runtimes of a given algorithm vary over the problem instances. Given several different algorithms that deliver high performance, we assume that in practice, one would prefer to use an algorithm that is unlikely to take significantly more than the average time. We refer to this measurement as *stability*.

Arguably, a more suitable measure to compare stability over all algorithms would be the relative deviation $\sigma(A)/\overline{t}(A)$, since higher runtimes incur a higher absolute deviation and thus appear less stable than they really are. However, we assume that an algorithm with high performance and low stability is always preferable to one with low performance and high stability. Thus we use stability only to discriminate between algorithms of high performance, giving precedence to those that also have the highest stability among these. Stability values $\sigma(A)$ are shown in Table 4.

In addition to the stability, we also report the averages of the worst ten instances for each algorithm and permutation in the matrix experiment in

Table 4: Average performance over the random and matrix instances and standard deviations for all combinations of heuristics and algorithms. Lower values indicate higher performance and stability.

	Performance				Stability			
Algorithm	NONE	SGM	KSM	MDM	NONE	SGM	KSM	MDM
PFP	4.54	4.42	2.64	4.24	12.6	12.5	2.1	3.7
HKDW	14.86	14.15	4.99	4.32	27.5	28.1	8.6	3.9
ABMP	9.93	6.79	5.19	4.99	11.0	6.2	6.7	5.4
PR-LIFO	191.00	38.89	5.80	4.91	414.9	65.7	8.1	4.8
PR-FIFO	3.43	2.54	2.88	4.73	3.3	1.4	2.3	5.1
PR-Low	25.79	3.65	3.25	5.05	61.6	2.7	2.7	5.4
PR-LIFO+Fair	221.64	38.94	5.69	5.35	511.5	65.9	8.1	6.2
PR-FIFO+Fair	5.04	2.46	2.78	4.53	10.5	1.3	2.2	4.6
PR-Low+Fair	14.63	3.51	3.13	4.53	25.4	2.5	2.5	4.1
PR-FIFO+Fair+RF/4	2.38	2.40	2.79	4.20	1.5	1.6	2.4	3.8
PR-FIFO+Fair+RF/2	2.68	2.35	2.73	4.21	2.4	1.4	2.3	3.8
PR-FIFO+Fair+3RF/4	3.40	2.40	2.74	4.41	4.7	1.3	2.2	4.3
PR-FIFO+Fair+2RF	7.33	2.84	3.13	4.81	17.3	1.5	2.5	4.7
PR-Low+Fair+RF/2	8.08	2.96	3.06	4.43	12.2	1.8	2.6	4.1
PR-Low+Fair+PFP	7.97	3.13	3.05	4.55	12.2	1.8	2.3	4.2
PR-FIFO+Fair+PFP	3.53	3.30	3.00	4.45	1.4	1.4	2.2	3.8
PR-FIFO+Fair+PFP+RF/2	3.56	3.26	3.00	4.45	1.5	1.5	2.2	3.8

Table 7 in the appendix. Interestingly, MDM has superior worst case runtimes, followed by KSM and SGM, which indicates that the low deficiency provided by MDM does indeed help all matching algorithms, but, as shown in Table 3 and 4, the extra effort outweighs the gains. As expected, using no initialization results in very high worst case runtimes.

Concerning the algorithms, PR, using frequent global relabelings, has the best worst case runtimes. This is consistent with the good stability results for these algorithms, as shown in Table 4.

7.4 Performance Profiles

In addition to the runtimes and their averages, we present some comparisons using performance profiles. Introduced in [11], performance profiles allow comparison of algorithmic performance at a more detailed level than presentation of runtime averages only. For each instance I, we define $\rho_I(A) = t_I(A)/t_I(A_I^*)$ as in Section 7.2. For a range of $1 \leq \theta \leq \kappa$, we plot the fraction of instances where $\rho_I(A) \leq \theta$ for each algorithm A to obtain its performance profile. In all the figures, the range of θ is always plotted on

the x-axis, with the values of κ ranging from 1.3 to 5, depending on the spread among the algorithms. For the fraction of instances, we always plot the full range between 0 and 1 on the y-axis.

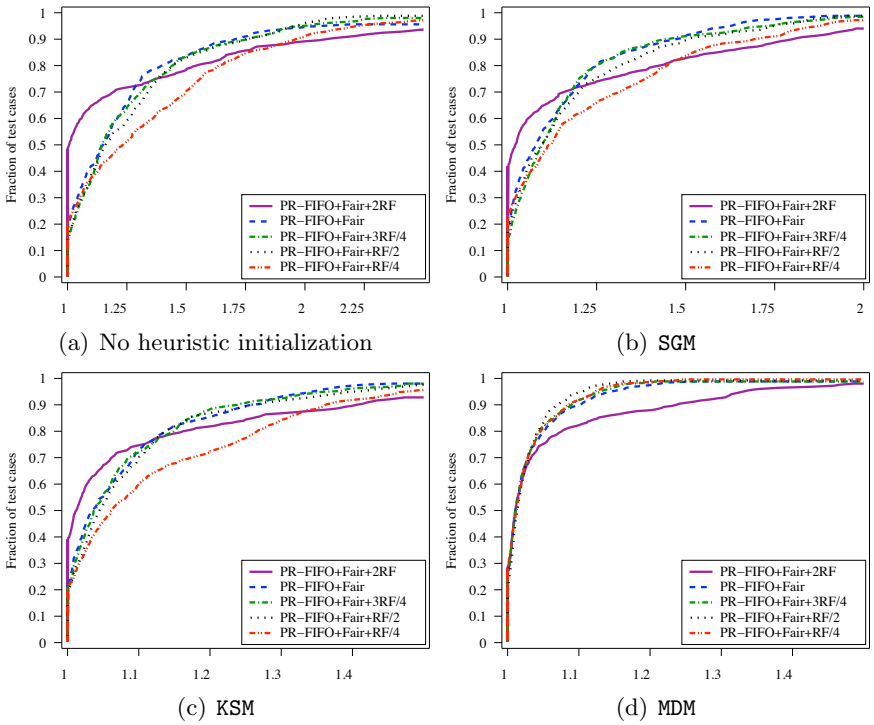
Due to the large number of algorithm/heuristic combinations studied in this paper, we restrict ourselves to the performance profiles that illustrate the most interesting comparisons we encountered. For each plot, the algorithm A_I^* is the best among the algorithms shown in that plot, not the best algorithm encountered in this study. Therefore, a performance profile is only comparable to one in the same plot.

Figure 1 contains the first series of performance plots. It illustrates the effect of global relabeling frequency on the performance of `PR-FIFO` for each of the four initializations. Only the matrix instances are used for this comparison.

All initializations except `MDM` are optimal about in 40% of the instances, and in almost 95% of all cases, they take less than twice the best runtime, further illustrating the stability of `PR-FIFO`. Elaborate initializations level the differences between the variants of the algorithm to some extent. The θ values, i.e., the factors of the best runtime within which 90% of the instances can be solved clearly decreases from Figure 1(a) using no initialization to figures 1(b) (`SGM`), 1(c) (`KSM`), and 1(d) (`MDM`), where all variants except `2RF` perform nearly identical. `2RF` solves easy instances quickly, but takes longer time for the harder cases. The opposite is true for `RF/4`, which attains good average performance in Table 4. Since a small fraction of the instances are "difficult" and thus require high runtimes which cannot be shown in the plots (see Table 7), especially robust variants such as `RF/4` will appear to have lower performance in these profiles.

In Figure 2, we compare the effects of initialization on the augmenting-path algorithms and on `PR-FIFO` using the standard relabeling frequency of n, by using the real-world matrix instances as the basis for comparison. For `ABMP`, there is a clear hierarchy from `MDM` initialization providing the best results down to using no initialization, which yields very poor performance. This is also true for `HKDW`, although it works significantly better than `ABMP` without initialization, and `MDM` does not pay off on the easier instances in `HKDW`. The situation is very different for `PFP` and `PR-FIFO`. Neither works well using `MDM` initialization. `PFP` clearly profits from using `SGM`, while `PR-FIFO` only draws marginal benefit from this initialization. Like `MDM`, `KSM` initialization does not pay off on the matrix instances. However, it provides the best performance on random instances (see Table 3).

In Figures 3 and 4, four different comparisons between algorithms using



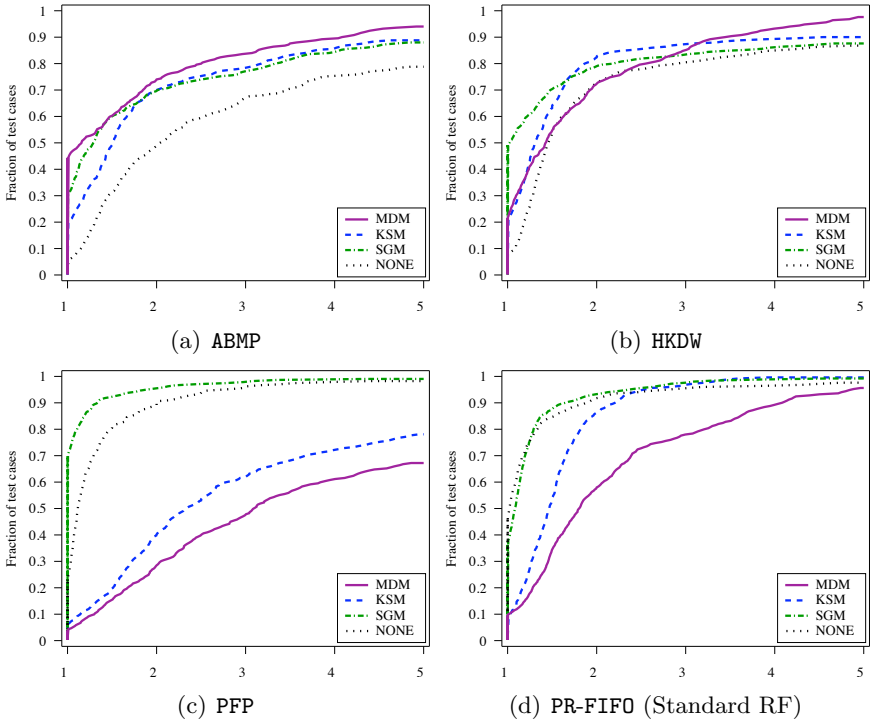
(a) No heuristic initialization (b) SGM

(c) KSM (d) MDM

Figure 1: Performance profiles for PR-FIFO using different initializations and relabeling frequencies. Elaborate initializations KSM and MDM level the differences between the variants of the algorithm PR-FIFO to some extent. The variant 2RF solves easy instances quickly, but takes longer time for the harder cases. The opposite is true for RF/4, which attains good average performance in Table 4 with good results on hard instances (see Table 7).

their most efficient initializations are shown. Figure 3 gives results for the matrix instances. The same comparisons using results from the random instances are given in Figure 4. Figures 3(a) and 4(a) compare augmenting-path algorithms and PR-Low. Clearly, PFP using KSM works fastest most of the time, thereby outstripping PR-Low. ABMP and HKDW show a slow but steady growth that is typical for the MDM initialization. The behavior is consistent for both types of instances, although KSM provides even greater benefits on the random instances.

Figures 3(b) and 4(b) illustrate the effect of fairness on PR-Low and PR-FIFO. All four algorithms use SGM initialization. It was chosen in order



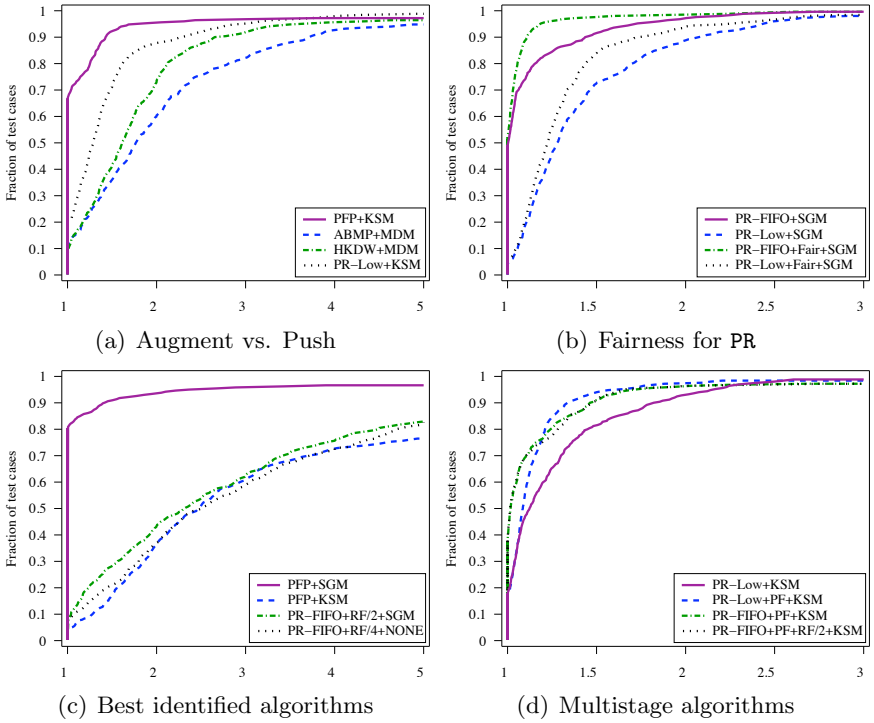
(a) ABMP (b) HKDW

(c) PFP (d) PR-FIFO (Standard RF)

Figure 2: Performance profiles for three augmenting-path-based algorithms and PR-FIFO with differing initialization algorithms on the matrix instances. For ABMP (a) and HKDW (b), MDM initialization provides superior results, while SGM yields the best performance for PFP (c) and PR-FIFO (d). The latter algorithms using SGM are also more robust, as only few instances take more than twice the minimum time.

to avoid the leveling effect of the elaborate initializations observed in Figure 1. For the matrix instances, fairness yields a noticeable improvement on a small number of instances. For the random matrix instances, the fair algorithms are slightly slower. The average values in Table 4 suggest that fairness improves performance by a small amount, most noticeably on the hard instances shown in Table 7. Still, the difference between PR-Low and PR-FIFO is much larger than the effect of fairness.

Figures 3(c) and 4(c) compare the best algorithms identified in this study, i.e., PFP and PR-FIFO with high relabeling frequency. Clearly, PFP using SGM initialization stands out, as it is much faster on most "easy" matrix instances, but slower on others. Interestingly, PFP using KSM, PR-FIFO with



(a) Augment vs. Push

(b) Fairness for PR

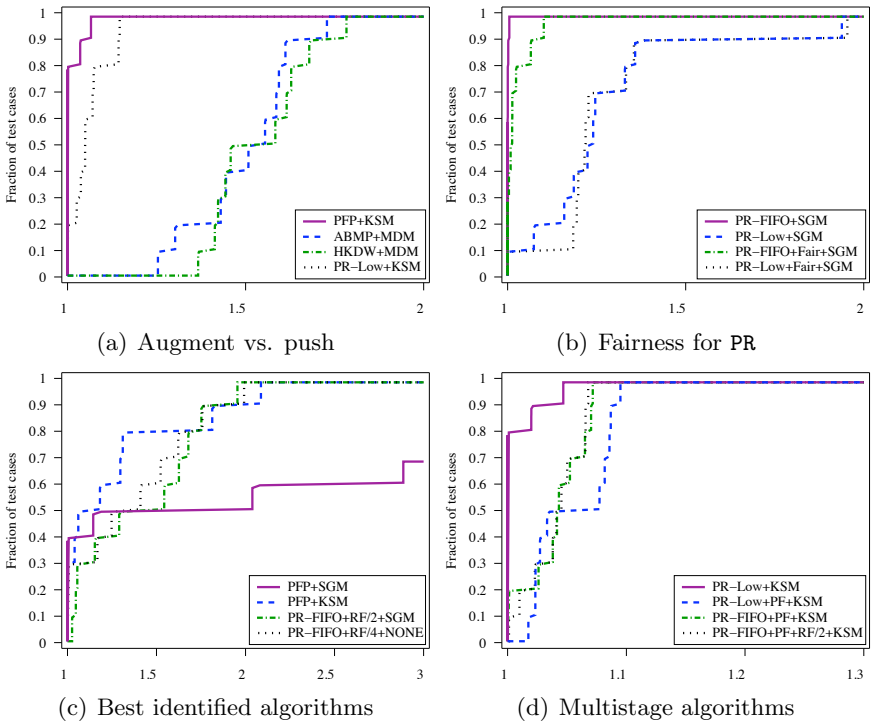
(c) Best identified algorithms

(d) Multistage algorithms

Figure 3: Performance profiles for comparing selected algorithms using their most efficient initializations on matrix instances: (a) compares augmenting-path algorithms and PR-Low; (b) illustrates the effect of fairness on PR-Low and PR-FIFO; (c) compares the best algorithms identified in this study, i.e., PFP and PR-FIFO with a high relabeling frequency; (d) studies the behavior of the multistage PR algorithms. Figure 4 contains results for the same experiments on the random instances.

frequency RF/2 using SGM and uninitialized PR-FIFO with frequency RF/4 behave similarly even though the algorithms are quite different. Clearly, using either PFP with KSM initialization or PR-FIFO provides a great deal of stability over both types of instances.

Finally, Figures 3(d) and 4(d) study the behavior of the PR multistage algorithms. All algorithms in these figures use KSM. Interestingly, the relabeling frequency for the multistage PR-FIFO algorithm has next to no effect. This indicates that the algorithms switch to PFP before the second global relabeling starts (the first global relabeling is performed after initialization).



(a) Augment vs. push

(b) Fairness for PR

(c) Best identified algorithms

(d) Multistage algorithms

Figure 4: Performance profiles for comparing selected algorithms using their most efficient initializations on random instances: (a) compares augmenting-path algorithms and PR-Low; (b) illustrates the effect of fairness on PR-Low and PR-FIFO; (c) compares the best algorithms identified in this study, i.e., PFP and PR-FIFO with high relabeling frequency; (d) studies the behavior of the multistage PR algorithms. Figure 3 contains results for the same experiments on the matrix instances.

For the matrix instances, PR-Low benefits from introducing the PFP stage, while on the random instances, the opposite is true. In both cases, the difference between PR-Low+PFP and PR-FIFO+PFP is small.

However, based on the results in Figure 3(b) and Table 4, even though the multistage approach looks promising, it cannot keep up with the standard PR-FIFO approach using a high relabeling frequency. This suggests that the push-relabel approach might often be faster than PFP when matching a small number of remaining unmatched vertices. Therefore, the multistage approach does not seem to be worthwhile, since the increased relabeling fre-

quency provides better results overall. Comparing the performance of ABMP and HKDW using KSM or MDM initialization in Table 4 confirms this finding. For PR-Low, switching to PFP improves performance in the matrix experiments, but Figure 3(b) indicates that PR-FIFO is faster to begin with and thus does not profit from the multistage technique.

7.5 General Observations

As the results vary widely over the test instances, it is difficult to extract general statements from the results. However, we can reliably observe the following.

- **The Karp-Sipser heuristic is superior to the minimum-degree heuristic for initialization**

  Although MDM usually produces matchings with lower deficiency, it is noticeably slower than KSM, and the improved initialization does not compensate for this, resulting in slower overall algorithms. This effect appears consistently throughout the experiments. This matches corresponding results from [20]. Note however that results in [14] indicate that MDM can be better than KSM for some augmenting-path algorithms and for some difficult random instances. However, in addition to PR-High, ABMP and HKDW show good performance using MDM in Table 3. Furthermore, judging from the good worst case performance shown in Table 7, MDM is quite robust and provides maximum stability for ABMP, HKDW, and the PR-LIFO codes.

- **Initialization for push-relabel competes with relabeling frequency**

  Some previous results [7] claimed that the push-relabel algorithm does not benefit from the simple greedy initialization, but in our experiments this was not universally true. For the random instances, KSM initialization (which provides matchings of far smaller deficiency than SGM) consistently sped up the combined PR algorithms. For the matrix instances, performance and stability decrease for the fast FIFO algorithms and increase for the slower LIFO and Low algorithms when using KSM instead of SGM. With the increasing relabeling frequency, the performance when using SGM or no initialization increases significantly. Therefore, the PR approach that performs best on average does not use KSM, but our results indicate that in some cases using KSM initialization pays off. However, the gains in performance are noticeably

smaller than those for PFP.

- **The gains from introducing fairness to the push-relabel algorithm are small**

  Since the push-relabel algorithm already has a guiding mechanism, we do not observe the same improvements as in PFP compared to the standard Pothen-Fan algorithm. However, fairness, i.e., varying the search direction when selecting a neighbor for an active vertex, is easy to implement and slightly improves performance. This improvement can be found consistently among all initializations and push orders. Fairness also improves stability.

- **The HKDW and ABMP algorithms perform well with elaborate initializations**

  Both algorithms show superior performance on the random instances when using KSM initialization. On the matrix instances, performance is worse than that of PFP and most PR algorithms, with MDM initialization providing the best results. Therefore, overall stability of these algorithms is low, although they consistently perform well on the harder instances. Consistent with the findings in [20], ABMP profits more from KSM initialization than the push-relabel codes.

- **The initialized PFP algorithm works well**

  With no or with the simple greedy initialization, PFP is extremely fast on some of the matrix instances. Note that when an instance is easy, PFP equipped with NONE and SGM only requires CCS and no additional data structure. However, due to the bad performance of these initialization heuristics, it is very slow on some of the random instances for which the deficiency of the initial matching is high. With KSM initialization, its overall performance is almost equal to the best push-relabel algorithm, and it is almost as stable. The algorithm along with the KSM heuristic is relatively simple to implement.

- **The gain in multistage algorithms is low**

  The multistage algorithms PR-FIFO+Fair+PFP and PR-Low+Fair+PFP work reasonably well. In fact, they improve slightly upon the underlying algorithms PR-FIFO+Fair and PR-Low+Fair, respectively. However, these algorithms benefit significantly from the increased global relabeling frequency, while the multistage algorithms seem to improve

only marginally. Therefore, we conclude that this approach ultimately does not pay off, and that the extra effort involved in implementing this approach is highly likely to outweigh any possible gains.

7.6 Comparison Between Push-Relabel Algorithms

Since preliminary experiments discussed in Section 7.1 indicated that the `High` variant is inferior to other `PR` codes, it was discarded before.

The performance of `LIFO` codes was also inferior in our experiments. Conceptually, `LIFO` push order means that after a double push that unmatches column w and matches column v to row u, w is selected as the next active vertex. This process continues until a free row vertex is found or none can be found. It thus resembles to DFS-based algorithms with additional overhead due to the labels and due to not using the lookahead mechanism. Even though `KSM` and `MDM` initializations speed up the algorithm tremendously, it is still inferior to `FIFO` and `Low`, as well as `PFP`. It is interesting to note that both `LIFO` and `High` work better with `MDM` initialization as compared to `KSM`.

Out of the push-relabel variants, we recommend `FIFO` push order with fairness and frequent global relabelings. In our experiments, `FIFO` performed better than `Low` while in [7] both algorithms performed equally well. This difference might be partially due to differences in data structures. However, considering that the advantage `FIFO` has over `Low` is larger on real-world instances, it is likely that the difference is mainly due to the different test sets, as [7] used only randomly generated instances. In any case, the expected overall performance of `FIFO` is at least as good as that of `Low`. This, in addition to the fact that `FIFO` is easier to implement, this suggests that `FIFO` is the first choice among all push orderings. Furthermore, its performance is also the most stable among the fast matching algorithms. This also implies that using gap relabeling does not need to be considered.

The results also clearly indicate that frequent global relabelings are worthwhile. However, with the increased global relabeling frequency, the effect of elaborate initialization declines so far that at the optimum global relabeling frequency of `RF/2`, using `KSM` is no longer worthwhile, and `SGM` is the best. For an even higher frequency of `RF/4`, even `SGM` does not improve performance anymore, and using no initialization becomes the best alternative. For the random instances, this is even true for frequencies of `RF` or higher.

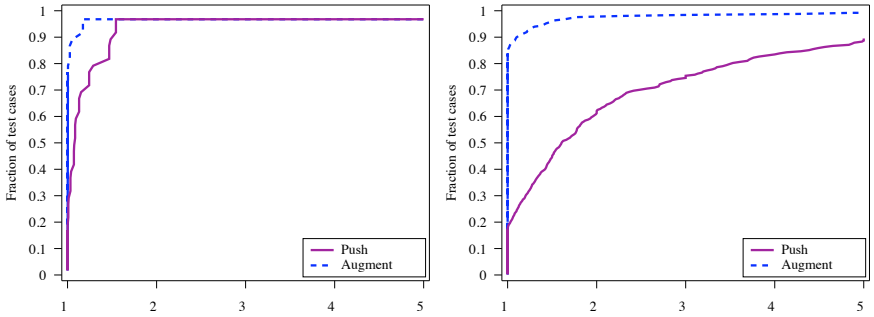
In [7], for a relabeling frequency of `RF`, `SGM` initialization was found to be detrimental to overall performance. This is not surprising considering that

the study used only randomly generated instances, even though their generators differ from those used in our experiments. Clearly, frequent relabeling tends to offset the gains from the elaborate initializations, and therefore, even though both techniques are helpful, the returns from using both at the same time are clearly diminishing. Finding a combination of an initialization heuristic and a relabeling scheme that can be shown to reliably improve performance remains an open question. For now, the overall effect of initializations depends too strongly on the instances, which prevents us from giving a clear recommendation. Using `SGM` with a relabeling frequency of `RF/2` showed the best performance, but the difference to using no initialization with a relabeling frequency of `RF/4` is very small, and therefore, even though the effort incurred in implementing `SGM` is minimal, it still might not be worthwhile.

7.7 Augment or Push?

In [7], it was investigated whether pushing is preferable to augmenting and concluded that push-relabel type algorithms usually perform better than the augmenting-path-based ones. While our study generally confirms their findings, [7] does not use elaborate initialization heuristics. In [14, 20], the strong positive effect of the `MDM` and `KSM` heuristics on the augmenting-path-based algorithms for the more difficult instances was established, and our results in Section 6 show that `PFP`, the best augmenting-path algorithm identified, can compete with the best `PR` algorithm when `KSM` is used.

Note that even though there exist `PR-FIFO` versions with a better average performance, `PFP` using `KSM` initialization is faster on the random instances, and thus the "winner" depends entirely on the data set. Furthermore, `PR-FIFO` beats `PFP` only when using a relabeling frequency that is higher than the standard of n recommended in [7]. However, the experiments reliably show that the competitive `PR` algorithms consistently have better stability than the augmenting-path-based algorithms whose behavior varies widely between the two types of instances. Thus, without further knowledge about the instance, it is likely that `PR-FIFO` will perform better on average. On the other hand, this means that if the type of instances is known beforehand, it is possible to exploit these properties. For the real-world instances, which are relatively easier than the random instances, the `SGM` initialized `PFP` was far better than all alternatives, and on the random instances the Karp-Sipser initialized augmenting-path algorithms perform best. Thus, for all except the most difficult instances, the fastest algorithm is usually an augmenting-path-based one, as shown in Figure 5.



(a) Performance on random matrices (b) Performance on real-world matrices

Figure 5: Performance profiles comparing the best augmenting-path algorithm with the best push-relabel algorithm for each instance on (a) random and (b) real-world matrix instances. Clearly, within the range of these profiles, the augmenting-path algorithms show better performance. This changes for very difficult instances though (see Table 7).

8 Concluding Remarks

We have investigated the performance of push-relabel algorithms in comparison with the augmenting-path based methods. Our findings illustrate that the difference between augmenting and pushing is rather small compared to the difference among either the augmenting-path-based algorithms or the push-relabel variants. Furthermore, elaborate initialization heuristics often have a greater impact than different algorithmic techniques.

Our results are largely consistent with earlier studies performed on randomly generated problem instances. By experimenting thoroughly on large problem instances arising in real-world applications, we were able to draw several clear conclusions. First of all, the augmenting-path based algorithm `PFP` [14] equipped with the `SGM` or `KSM` initialization heuristics and the push-relabel variant `PR-FIFO` are preferable to all the others. The performance of the `PR-FIFO` algorithm is very good and also very stable over the different classes of instances. On the other hand, depending on the application, `PFP` with suitable initialization can be noticeably faster. Further results include that the Karp-Sipser initialization heuristic yields the best results among the alternatives studied and that the `PR` algorithms can often benefit from initialization heuristics.

References

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

[2] Jonathan Aronson, Alan Frieze, and Boris G. Pittel. Maximum matchings in sparse random graphs: Karp-sipser revisited. *Random Structures and Algorithms*, 12:111–177, March 1998.

[3] Ariful Azad, Johannes Langguth, Youhan Fang, Alan Qi, and Alex Pothen. Identifying rare cell populations in comparative flow cytometry. In *Proceedings of the 10th international conference on Algorithms in bioinformatics*, WABI'10, pages 162–175, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Rodney J Baxter. *Exactly Solved Models in Statistical Mechanics*. Academic Press, London, 1982.

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

[6] B. V. Cherkassky. A fast algorithm for computing maximum flow in a network. In A. V. Karzanov, editor, *Collected Papers, Issue 3: Combinatorial Methods for Flow Problems*, pages 90–96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in AMS Translations, Vol. 158, pp. 23-30. AMS, Providence, RI, 1994.

[7] 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. *ACM Journal of Experimental Algorithmics*, 3, September 1998.

[8] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection http://www.cise.ufl.edu/research/sparse. *ACM Transactions on Mathematical Software*, 2010.

[9] U. Derigs and W. Meier. Implementing Goldberg's max-flow algorithm - a computational investigation. *ZOR - Methods and models of Operations research*, 33:383–403, 1989.

[10] Jerry Ray Dias and George W A Milne. Chemical applications of graph theory. *Journal of Chemical Information and Computer Sciences*, 32(1):1–1, 1992.

[11] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002. 10.1007/s101070100263.

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

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

[14] I. S. Duff, K. Kaya, and B. Uçar. Design, implementation, and analysis of maximum transversal algorithms. Technical Report TR/PA/10/76, CERFACS, France, 2010.

[15] 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.

[16] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the Association for Computing Machinery*, 35:921–940, 1988.

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

[18] 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.

[19] Robert J. Kennedy, Jr. *Solving unweighted and weighted bipartite matching problems in theory and practice*. PhD thesis, Stanford University, Stanford, CA, USA, 1995. UMI Order No. GAX96-02908.

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

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

[22] Jakob Magun. Greedy matching algorithms, an experimental study. *ACM Journal of Experimental Algorithmics*, 3, September 1998.

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

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

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

APPENDIX

In the following, we present detailed results for our experiments.

Table 5: Runtimes relative to the best performance for the random instances. Numbers in the first row denote the average vertex degree.

Algorithm	Heuristic	1	2	3	4	5	6	7	8	9	10	Average
PFP	SGM	1.00	1.00	40.42	14.87	5.51	3.51	2.39	1.36	1.02	1.00	7.21
	KSM	2.09	1.31	1.00	1.03	1.10	1.29	1.38	1.24	1.32	1.81	1.36
	MDM	3.06	1.96	1.53	1.63	1.70	1.81	2.06	2.08	2.25	3.18	2.13
	NONE	1.18	1.10	40.73	14.77	5.37	3.55	2.49	1.36	1.00	1.13	7.27
HKDW	SGM	1.97	2.15	87.89	32.20	13.72	8.55	6.20	4.40	3.24	3.30	16.36
	KSM	2.01	1.24	1.04	1.07	1.01	1.07	1.10	1.20	1.31	1.85	1.29
	MDM	2.97	1.91	1.58	1.66	1.58	1.70	1.84	2.02	2.22	3.23	2.07
	NONE	3.16	2.50	78.85	22.84	7.99	4.55	3.23	2.89	2.41	3.18	13.16
ABMP	SGM	1.43	3.29	10.83	5.34	3.30	2.63	2.43	2.39	2.31	3.11	3.71
	KSM	2.05	1.31	1.05	1.04	1.00	1.00	1.00	1.17	1.24	1.89	1.28
	MDM	2.98	1.98	1.59	1.63	1.59	1.56	1.69	1.93	2.13	3.13	2.02
	NONE	4.33	5.16	12.28	6.60	4.36	3.54	3.17	3.24	3.01	3.97	4.97
PR-LIFO +RF/2	SGM	37.65	61.85	37.85	14.74	6.00	3.02	1.65	1.36	1.32	1.49	16.69
	KSM	2.04	1.25	1.14	1.06	1.15	1.30	1.37	1.32	1.40	1.94	1.40
	MDM	2.96	1.92	1.62	1.66	1.76	1.62	1.89	2.09	2.22	3.22	2.09
	NONE	135.08	86.95	41.92	14.70	5.72	3.24	2.07	1.58	1.29	1.75	29.43
PR-FIFO +RF/2	SGM	2.06	2.21	2.23	1.58	1.43	1.50	1.55	1.45	1.25	1.49	1.67
	KSM	2.04	1.25	1.16	1.07	1.16	1.30	1.37	1.32	1.40	1.94	1.40
	MDM	2.99	1.94	1.63	1.66	1.76	1.61	1.88	2.08	2.23	3.23	2.10
	NONE	2.82	2.01	2.03	1.65	1.32	1.31	1.37	1.31	1.32	1.78	1.69
PR-Low +RF/2	SGM	3.98	2.20	2.40	2.14	1.90	1.86	1.89	1.72	1.56	1.73	2.14
	KSM	2.42	1.35	1.26	1.14	1.27	1.46	1.53	1.43	1.53	2.10	1.55
	MDM	3.35	2.03	1.70	1.73	1.88	1.67	1.98	2.16	2.29	3.32	2.21
	NONE	5.01	2.87	2.63	2.17	1.75	1.79	1.85	1.75	1.78	2.34	2.40
PR-FIFO +Fair+RF/2	SGM	2.06	2.25	2.26	1.74	1.43	1.50	1.55	1.46	1.34	1.51	1.71
	KSM	2.06	1.25	1.16	1.07	1.16	1.30	1.36	1.32	1.40	1.94	1.40
	MDM	2.98	1.93	1.63	1.67	1.76	1.61	1.88	2.08	2.22	3.22	2.10
	NONE	2.65	1.90	1.97	1.75	1.26	1.26	1.31	1.24	1.28	1.73	1.63
PR-Low Fair+RF/2	SGM	4.02	2.20	2.64	2.13	1.91	1.84	1.89	1.74	1.50	1.81	2.17
	KSM	2.39	1.34	1.25	1.13	1.27	1.45	1.52	1.43	1.52	2.09	1.54
	MDM	3.33	2.01	1.70	1.73	1.88	1.67	1.97	2.15	2.30	3.33	2.21
	NONE	5.02	2.91	2.68	2.57	1.83	1.79	1.84	1.75	1.80	2.37	2.46
PR-FIFO +Fair+RF/8	SGM	1.79	1.45	1.92	1.67	1.32	1.35	1.14	1.12	1.18	1.67	1.46
	KSM	2.32	1.24	1.03	1.00	1.01	1.09	1.13	1.19	1.28	1.80	1.31
	MDM	3.26	1.93	1.54	1.60	1.61	1.56	1.75	2.01	2.16	3.14	2.06
	NONE	1.99	1.41	1.76	1.56	1.37	1.22	1.17	1.19	1.19	1.62	1.45
PR-FIFO +Fair+RF/4	SGM	1.54	1.68	1.96	1.67	1.27	1.25	1.24	1.25	1.32	1.75	1.49
	KSM	2.05	1.24	1.07	1.02	1.06	1.16	1.21	1.23	1.32	1.83	1.32
	MDM	3.00	1.93	1.57	1.62	1.66	1.58	1.80	2.03	2.18	3.17	2.05
	NONE	1.98	1.46	1.88	1.63	1.26	1.32	1.16	1.00	1.07	1.47	1.42
PR-FIFO +Fair+3RF/8	SGM	1.77	1.99	2.08	1.79	1.36	1.35	1.39	1.37	1.33	1.71	1.61
	KSM	2.06	1.25	1.12	1.05	1.11	1.23	1.29	1.28	1.37	1.90	1.37
	MDM	3.00	1.93	1.60	1.64	1.70	1.59	1.84	2.06	2.21	3.20	2.08
	NONE	2.33	1.63	1.86	1.69	1.40	1.16	1.14	1.11	1.20	1.64	1.52
PR-FIFO +Fair	SGM	3.19	3.04	2.73	2.03	1.87	2.01	1.53	1.22	1.25	1.68	2.05
	KSM	2.05	1.25	1.31	1.16	1.34	1.59	1.67	1.49	1.59	2.10	1.56
	MDM	2.97	1.92	1.73	1.75	1.94	1.69	2.05	2.18	2.29	3.31	2.18
	NONE	3.86	3.18	2.93	1.83	1.65	1.79	1.87	1.52	1.19	1.47	2.13
PR-Low +Fair+RF/4	SGM	3.02	2.11	2.36	2.06	1.67	1.46	1.44	1.43	1.49	2.09	1.91
	KSM	2.39	1.35	1.15	1.08	1.14	1.24	1.30	1.30	1.42	1.95	1.43
	MDM	3.36	2.03	1.63	1.66	1.74	1.61	1.85	2.08	2.25	3.26	2.15
	NONE	3.83	2.41	2.50	2.16	1.85	1.82	1.44	1.40	1.50	2.04	2.09
PR-Low +Fair+PFP+RF/2	SGM	2.30	2.29	2.25	2.11	1.88	1.89	2.73	1.95	1.78	2.41	2.16
	KSM	2.45	1.37	1.30	1.16	1.31	1.49	1.68	1.54	1.66	2.27	1.62
	MDM	3.37	2.07	1.75	1.76	1.91	1.68	2.08	2.24	2.36	3.42	2.26
	NONE	5.20	2.54	2.59	2.12	1.85	1.91	3.04	2.18	1.93	2.54	2.59
PR-FIFO +Fair+PFP+RF/2	SGM	2.37	2.08	5.09	5.60	3.97	3.60	2.80	1.91	1.62	2.10	3.11
	KSM	2.37	1.44	1.20	1.22	1.32	1.54	1.63	1.49	1.60	2.15	1.60
	MDM	3.34	2.12	1.72	1.82	1.92	1.90	2.20	2.27	2.43	3.45	2.32
	NONE	2.91	2.23	4.85	5.63	4.31	3.54	2.88	1.83	1.72	2.00	3.19
PR-FIFO +Fair+PFP+RF/4	SGM	2.03	1.67	4.86	5.74	4.14	3.60	2.81	1.92	1.61	2.09	3.05
	KSM	2.39	1.44	1.20	1.21	1.32	1.53	1.63	1.49	1.60	2.14	1.60
	MDM	3.33	2.12	1.72	1.81	1.92	1.90	2.20	2.28	2.42	3.46	2.32
	NONE	2.62	1.84	5.22	5.71	4.34	3.69	2.94	2.05	1.91	1.99	3.23

Table 6: Runtimes relative to the best performance for the matrix instances. Total avg. denotes the average of the matrix average and the random average from Table 5.

Algorithm	Heuristic	No perm.	Row perm.	Col. perm.	Row + col.perm.	Matrix avg.	Total avg.
PFP	SGM	1.55	1.79	1.54	1.64	1.63	4.42
	KSM	4.05	3.47	3.46	4.68	3.92	2.64
	MDM	8.81	5.18	4.30	7.10	6.35	4.24
	NONE	1.94	1.98	1.57	1.73	1.80	4.54
HKDW	SGM	5.13	32.15	5.83	4.62	11.93	14.15
	KSM	4.55	19.80	5.38	5.05	8.69	4.99
	MDM	9.01	5.62	4.44	7.21	6.57	4.32
	NONE	42.64	12.32	4.91	6.34	16.55	14.86
ABMP	SGM	5.43	13.59	11.77	8.75	9.88	6.79
	KSM	4.52	10.56	11.99	9.35	9.10	5.19
	MDM	12.28	6.25	5.09	8.23	7.96	4.99
	NONE	27.52	12.32	8.89	10.82	14.89	9.93
PR-LIFO +RF/2	SGM	105.91	120.37	8.61	9.42	61.08	38.89
	KSM	12.75	16.27	5.42	6.35	10.20	5.80
	MDM	10.78	7.19	5.01	7.89	7.72	4.91
	NONE	761.08	591.68	30.16	27.33	352.56	191.00
PR-FIFO +RF/2	SGM	2.88	3.18	3.71	3.90	3.41	2.54
	KSM	4.01	3.89	4.50	5.07	4.37	2.88
	MDM	12.27	5.35	4.52	7.33	7.37	4.73
	NONE	7.98	3.14	5.60	4.00	5.18	3.43
PR-Low +RF/2	SGM	7.02	3.77	4.70	5.16	5.16	3.65
	KSM	4.78	4.42	5.13	5.52	4.96	3.25
	MDM	13.10	5.85	4.91	7.71	7.89	5.05
	NONE	135.28	29.41	18.24	13.82	49.19	25.79
PR-FIFO +Fair+RF/2	SGM	2.88	3.06	3.00	3.89	3.21	2.46
	KSM	3.95	3.78	3.85	5.03	4.15	2.78
	MDM	11.03	5.22	4.40	7.21	6.96	4.53
	NONE	24.22	2.78	3.10	3.70	8.45	5.04
PR-Low +Fair+RF/2	SGM	3.89	3.58	6.67	5.29	4.86	3.51
	KSM	4.71	4.28	4.42	5.49	4.72	3.13
	MDM	9.60	5.54	4.66	7.63	6.86	4.53
	NONE	55.63	23.51	15.12	12.95	26.80	14.63
PR-FIFO +Fair+RF/8	SGM	2.89	3.24	3.03	4.21	3.34	2.40
	KSM	3.89	3.91	3.90	5.40	4.28	2.79
	MDM	8.90	5.14	4.19	7.15	6.34	4.20
	NONE	3.06	3.05	3.11	4.03	3.31	2.38
PR-FIFO +Fair+RF/4	SGM	2.83	3.08	2.95	3.98	3.21	2.35
	KSM	3.88	3.80	3.73	5.13	4.13	2.73
	MDM	8.97	5.12	4.21	7.14	6.36	4.21
	NONE	6.17	2.80	2.98	3.82	3.94	2.68
PR-FIFO +Fair+3RF/8	SGM	2.90	3.04	2.89	3.90	3.18	2.40
	KSM	3.89	3.77	3.74	5.03	4.11	2.74
	MDM	10.38	5.15	4.27	7.15	6.74	4.41
	NONE	11.62	2.75	3.06	3.72	5.29	3.40
PR-FIFO +Fair	SGM	3.03	3.27	3.98	4.24	3.63	2.84
	KSM	4.06	4.53	5.01	5.24	4.71	3.13
	MDM	11.11	6.20	4.57	7.86	7.43	4.81
	NONE	39.09	3.45	3.38	4.22	12.53	7.33
PR-Low +Fair+RF/4	SGM	3.81	3.69	3.56	4.96	4.00	2.96
	KSM	4.63	4.26	4.27	5.62	4.69	3.06
	MDM	9.57	5.39	4.46	7.41	6.71	4.43
	NONE	27.93	10.75	9.03	8.53	14.06	8.08
PR-Low +Fair+PFP+RF/2	SGM	3.89	3.47	3.19	5.82	4.09	3.13
	KSM	4.84	3.97	3.91	5.23	4.49	3.05
	MDM	10.06	5.43	4.46	7.42	6.84	4.55
	NONE	8.78	27.56	3.74	13.31	13.35	7.97
PR-FIFO +Fair+PFP+RF/2	SGM	3.07	3.16	3.70	3.98	3.48	3.30
	KSM	4.22	3.78	4.56	5.04	4.40	3.00
	MDM	9.29	5.33	4.41	7.32	6.59	4.45
	NONE	4.00	3.27	4.13	4.05	3.86	3.53
PR-FIFO +Fair+PFP+RF/4	SGM	3.01	3.22	3.71	3.93	3.47	3.26
	KSM	4.16	3.81	4.61	5.05	4.41	3.00
	MDM	9.26	5.33	4.41	7.32	6.58	4.45
	NONE	3.84	3.30	4.42	4.00	3.89	3.56

Table 7: Average of the worst ten runtimes in seconds for the matrix instances with various permutations. MDM consistently provides very good results here. Ratio denotes the ratio between the average and the best average found among all algorithms.

Algorithm	Heuristic	No perm.	Row perm.	Col. perm.	Row + col.perm.	Average	Ratio
PFP	SGM	3.00	5.95	4.69	8.58	5.55	3.31
	KSM	1.98	4.35	4.00	7.01	4.34	2.58
	MDM	2.10	4.43	3.63	7.12	4.32	2.57
	NONE	3.08	5.96	4.69	8.66	5.60	3.33
HKDW	SGM	4.17	53.53	9.79	9.38	19.22	11.45
	KSM	2.36	20.34	8.17	7.46	9.58	5.71
	MDM	2.37	3.80	5.10	7.89	4.79	2.85
	NONE	6.22	11.92	7.31	9.48	8.73	5.20
ABMP	SGM	2.46	37.02	28.23	16.57	21.07	12.55
	KSM	0.78	12.30	22.29	14.17	12.39	7.38
	MDM	1.84	8.31	6.97	12.59	7.43	4.42
	NONE	4.45	23.11	13.98	17.60	14.79	8.81
PR-LIFO +RF/2	SGM	362.51	910.62	25.14	32.54	332.70	198.16
	KSM	18.64	30.93	5.73	6.88	15.55	9.26
	MDM	7.94	11.91	5.08	6.74	7.92	4.72
	NONE	880.83	1553.39	103.90	125.01	665.78	396.54
PR-FIFO +RF/2	SGM	0.88	2.67	4.00	4.60	3.04	1.81
	KSM	0.68	2.41	4.01	4.02	2.78	1.66
	MDM	1.58	2.64	3.19	3.78	2.80	1.67
	NONE	1.67	2.12	3.59	4.04	2.85	1.70
PR-Low +RF/2	SGM	1.58	2.73	7.48	18.74	7.63	4.55
	KSM	0.85	3.17	4.81	4.90	3.43	2.05
	MDM	1.75	3.83	4.16	4.75	3.62	2.16
	NONE	121.15	160.60	143.83	165.61	147.80	88.03
PR-FIFO +Fair+RF/2	SGM	0.77	2.38	2.63	4.45	2.56	1.52
	KSM	0.60	1.92	2.72	3.64	2.22	1.32
	MDM	1.28	2.10	2.62	2.98	2.24	1.34
	NONE	4.65	1.73	2.15	3.97	3.13	1.86
PR-Low +Fair+RF/2	SGM	0.81	2.34	35.98	21.85	15.24	9.08
	KSM	0.63	2.59	3.33	4.64	2.80	1.67
	MDM	0.91	2.49	2.89	4.12	2.60	1.55
	NONE	42.29	127.07	107.95	148.77	106.52	63.44
PR-FIFO +Fair+RF/8	SGM	0.65	2.27	2.14	3.72	2.20	1.31
	KSM	0.53	1.69	2.37	3.57	2.04	1.21
	MDM	0.83	1.71	1.71	2.54	1.70	1.01
	NONE	0.59	2.21	2.22	3.71	2.18	1.30
PR-FIFO +Fair+RF/4	SGM	0.68	2.12	2.12	3.74	2.16	1.29
	KSM	0.53	1.61	2.31	3.47	1.98	1.18
	MDM	0.85	1.64	1.74	2.48	1.68	1.00
	NONE	1.22	1.83	2.21	3.76	2.25	1.34
PR-FIFO +Fair+3RF/8	SGM	0.69	2.08	2.13	3.94	2.21	1.32
	KSM	0.57	1.67	2.38	3.44	2.01	1.20
	MDM	1.17	1.81	2.03	2.58	1.90	1.13
	NONE	2.30	1.67	2.08	3.80	2.46	1.47
PR-FIFO +Fair	SGM	0.96	3.27	6.52	6.28	4.26	2.54
	KSM	0.71	5.13	7.80	4.84	4.62	2.75
	MDM	1.35	6.21	3.23	7.30	4.52	2.69
	NONE	7.63	3.96	3.40	6.70	5.42	3.23
PR-Low +Fair+RF/4	SGM	0.83	2.20	2.21	11.78	4.25	2.53
	KSM	0.60	2.05	2.50	4.44	2.40	1.43
	MDM	0.90	1.90	2.12	2.80	1.93	1.15
	NONE	11.45	41.37	43.07	57.14	38.26	22.79
PR-Low +Fair+PFP+RF/2	SGM	0.88	3.28	2.75	37.92	11.21	6.67
	KSM	0.67	2.41	2.74	5.32	2.79	1.66
	MDM	0.96	2.99	2.29	5.36	2.90	1.73
	NONE	0.98	164.18	2.70	170.38	84.56	50.36
PR-FIFO +Fair+PFP+RF/2	SGM	0.95	3.15	3.25	6.48	3.46	2.06
	KSM	0.84	2.66	3.47	5.35	3.08	1.84
	MDM	1.06	3.15	2.44	5.45	3.03	1.80
	NONE	0.98	3.51	3.21	6.38	3.52	2.10
PR-FIFO +Fair+PFP+RF/4	SGM	0.91	3.15	3.34	6.14	3.38	2.02
	KSM	0.84	2.63	3.55	5.35	3.09	1.84
	MDM	1.07	3.16	2.44	5.45	3.03	1.81
	NONE	1.01	3.77	4.34	6.27	3.85	2.29

Table 8: List of matrices from the UFL Sparse Matrix Collection used in the experiments.

Group	Name	Group	Name	Group	Name
ATandT	*pre2*	IBM_EDA	*trans4*	McRae	*ecology2*
ATandT	*twotone*	IBM_EDA	*trans5*	QLi	*largebasis*
Hamm	*hcircuit*	PARSEC	*CO*	CEMW	*t2em*
Hamm	*scircuit*	PARSEC	*Ga10As10H30*	CEMW	*tmt_unsym*
LPnetlib	*lp_ken_18*	PARSEC	*Ge87H76*	CEMW	*tmt_sym*
Ronis	*xenon2*	Rajat	*rajat21*	TKK	*engine*
Rothberg	*cfd2*	Rajat	*rajat23*	Um	*2cubes_sphere*
Norris	*lung2*	Rajat	*rajat24*	JGD_Forest	*TF19*
Norris	*stomach*	Andrianov	*ins2*	JGD_GL7d	*GL7d15*
Norris	*torso2*	Andrianov	*lp1*	JGD_GL7d	*GL7d23*
Norris	*torso3*	Rajat	*rajat29*	JGD_Homology	*ch7-8-b5*
vanHeukelum	*cage12*	Rajat	*rajat30*	JGD_Homology	*ch7-9-b4*
vanHeukelum	*cage13*	AMD	*G2_circuit*	JGD_Homology	*ch7-9-b5*
Schenk_IBMNA	*c-73*	Sandia	*ASIC_320k*	JGD_Homology	*ch8-8-b4*
Schenk_IBMSDS	*matrix_9*	Sandia	*ASIC_320ks*	JGD_Homology	*ch8-8-b5*
Schenk_IBMSDS	*matrix-new_3*	Sandia	*ASIC_680k*	JGD_Homology	*m133-b3*
Schenk_ISEI	*barrier2-10*	Sandia	*ASIC_680ks*	JGD_Homology	*mk13-b5*
Schenk_ISEI	*barrier2-11*	AMD	*G3_circuit*	JGD_Homology	*n4c6-b10*
Schenk_ISEI	*barrier2-12*	GHS_psdef	*apache2*	JGD_Homology	*n4c6-b7*
Schenk_ISEI	*barrier2-1*	Oberwolfach	*filter3D*	JGD_Homology	*n4c6-b8*
Schenk_ISEI	*barrier2-2*	Oberwolfach	*boneS01*	JGD_Homology	*n4c6-b9*
Schenk_ISEI	*barrier2-3*	Pajek	*IMDB*	JGD_Homology	*shar_te2-b3*
Schenk_ISEI	*barrier2-4*	Pajek	*internet*	JGD_Margulies	*kneser_10_4_1*
Schenk_ISEI	*barrier2-9*	Barabasi	*NotreDame_actors*	JGD_Margulies	*wheel_601*
Schenk_ISEI	*ohne2*	Barabasi	*NotreDame_www*	FreeFieldTech.	*mono_500Hz*
Schenk_ISEI	*para-10*	Pajek	*patents_main*	TSOPF	*TSOPF_FS_b39_c30*
Schenk_ISEI	*para-4*	Schenk_IBMNA	*c-big*	Yoshiyasu	*image_interp*
Schenk_ISEI	*para-5*	Schenk_IBMNA	*c-73b*	Botonakis	*thermomech_TC*
Schenk_ISEI	*para-6*	Mittelmann	*pds-70*	Botonakis	*thermomech_TK*
Schenk_ISEI	*para-7*	Mittelmann	*pds-80*	Botonakis	*thermomech_dM*
Schenk_ISEI	*para-8*	Mittelmann	*pds-90*	Botonakis	*thermomech_dK*
Schenk_ISEI	*para-9*	Mittelmann	*pds-100*	Freescale	*transient*
Kamvar	*Stanford*	Mittelmann	*sgpf5y6*	Um	*offshore*
Kamvar	*Stan._Berk.*	Mittelmann	*stormG2_1000*	SNAP	*email-EuAll*
Tromble	*language*	Mittelmann	*watson_1*	SNAP	*web-BerkStan*
Hamrle	*Hamrle3*	Mittelmann	*watson_2*	SNAP	*web-Google*
Lin	*Lin*	Mittelmann	*cont11_l*	SNAP	*web-NotreDame*
GHS_indef	*d_pretok*	Mittelmann	*cont1_l*	SNAP	*web-Stanford*
GHS_indef	*darcy003*	Mittelmann	*neos*	SNAP	*amazon0302*
GHS_indef	*helm2d03*	Mittelmann	*neos1*	SNAP	*amazon0312*
GHS_indef	*mario002*	Mittelmann	*neos2*	SNAP	*amazon0505*
GHS_indef	*turon_m*	Mittelmann	*neos3*	SNAP	*amazon0601*
GHS_psdef	*bmw7st_1*	UTEP	*Dubcova3*	SNAP	*roadNet-CA*
GHS_psdef	*ford2*	Botonakis	*FEM_3D_thermal2*	SNAP	*roadNet-PA*
DNVS	*ship_003*	Wissgott	*parabolic_fem*	SNAP	*roadNet-TX*
DNVS	*shipsec1*	Watson	*Baumann*	SNAP	*soc-sign-epinions*
DNVS	*shipsec5*	QLi	*crashbasis*	Gleich	*usroads-48*
DNVS	*shipsec8*	QLi	*majorbasis*	Gleich	*usroads*
GHS_indef	*boyd2*	Rajat	*Raj1*	Williams	*mac_econ_fwd500*
GHS_indef	*cont-300*	HVDC	*hvdc2*	Williams	*mc2depi*
IBM_EDA	*dc1*	Rucci	*Rucci1*	Williams	*cop20k_A*
IBM_EDA	*dc2*	McRae	*ecology1*	Williams	*webbase-1M*
IBM_EDA	*dc3*				