HAL
open science

Push-relabel based algorithms for the maximum transversal problem

Kamer Kaya, Johannes Langguth, Fredrik Manne, Bora Uçar

To cite this version:

Kamer Kaya, Johannes Langguth, Fredrik Manne, Bora Uçar. Push-relabel based algorithms for the maximum transversal problem. [Research Report] RR-8093, 2012, pp.27. hal-00739360v1

HAL Id: hal-00739360

https://inria.hal.science/hal-00739360v1

Submitted on 8 Oct 2012 (v1), last revised 31 Oct 2012 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Push-relabel based algorithms for the maximum transversal problem

Kamer Kaya, Johannes Langguth, Fredrik Manne, Bora Uçar

RESEARCH REPORT
N° 8093
October 2012
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8093--FR+ENG

Push-relabel based algorithms for the maximum transversal problem

Kamer Kaya, Johannes Langguth, Fredrik Manne, Bora Uçar

Project-Team ROMA

Research Report n° 8093 — October 2012 — 27 pages

Abstract: We investigate the push-relabel algorithm for solving the problem of finding a maximum cardinality matching in a bipartite graph in the context of the maximum transversal problem. We describe in detail an optimized yet easy-to-implement version of the algorithm and fine-tune its parameters. We also introduce new performance-enhancing techniques. On a wide range of real-world instances, we compare the push-relabel algorithm with state-of-the-art augmenting path-based algorithms and the recently proposed pseudoflow approach. We conclude that a carefully tuned push-relabel algorithm is competitive with all known augmenting path-based algorithms, and superior to the pseudoflow-based ones.

Key-words: Bipartite graphs, matching, push-relabel-based algorithms

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes de type Push-Relabel pour le problème de couplage maximum

Résumé : Nous étudions le problème de couplage maximum dans des graphes bipartis. Nous décrivons en détail une version optimisée de l'algorithme en ajustant ses paramètres. L'algorithme est facile à mettre en œuvre. Nous introduisons également de nouvelles techniques pour améliorer la performance de l'algorithme. Sur un large éventail de cas du monde réel, nous comparons l'algorithme Push-Relabel avec des algorithmes basés sur les concepts de chemins augmentants et de pseudoflot récemment proposés. Nous concluons qu'un algorithme de type Push-Relabel soigneusement réglé est en concurrence avec tous les algorithmes connus de type chemins augmentants, et supérieur à ceux de type pseudoflot.

Mots-clés : couplage, graphes bipartis

1 Introduction

We study algorithms for finding the maximum cardinality matching in bipartite graphs, a problem which arises in a large number of applications. Our main motivation is the problem of finding a maximum transversal, i.e., obtaining a maximum number of nonzeros in the diagonal of a sparse matrix by permuting its rows and columns, which can be solved via bipartite matching algorithms [8]. Other applications can be found in many fields such as bioinformatics [3], scheduling [24], and chemical structure analysis [14].

There are several different algorithms for computing a maximum matching in a bipartite graph. One class of algorithms is based on augmenting paths. Duff et al. [10] discuss the design, analysis and implementation of eight augmenting path-based algorithms. Push-relabel-based algorithms form the second class. We implement and study the `FIFO` version of these algorithms using the same data structures as the algorithms described by Duff et al. [10]. Based on our experiments which are described in the technical report [16], we found the `FIFO` version to be superior to its alternatives. A third class, pseudoflow algorithms, is based on more recent work [12] whose implementations are described by Chandran and Hochbaum [6].

Our contributions in this study are threefold. First, we present the push-relabel algorithm for the maximum cardinality matching problem in bipartite graphs in its elegance and simplicity. As the push-relabel algorithm was designed for the maximum flow problem, its usual presentations are much more complicated than necessary for bipartite matching. We give a pseudocode that, like most matching algorithms, is easy to implement and avoids unnecessary complexities. Our second contribution is an experimental comparison of the push-relabel algorithm with the most recent alternatives. Our experiments focus on maximum traversal problems arising in real life applications. We report thorough results on all large enough problems corresponding to matrices from the University of Florida Sparse Matrix Collection. Our third contribution is the adaptation of a simple strategy proposed by Duff et al. [10] to the push-relabel algorithm, as well as an additional modification, which speed up the algorithm noticeably. In the accompanying technical report [16], we investigated the performance of the different push-relabel-based algorithms and performed comparisons with the augmenting path-based ones. We will use the results presented in this report to short-cut some experimental investigations (the reader will be alerted to check the report whenever necessary).

The rest of this paper is organized as follows. We briefly discuss the notation and the background in Section 2 and present the the push-relabel (`PR`) algorithm 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 other algorithms used for comparison. Starting from Section 6, we describe the experimental setup and present our experimental results, along with their discussion and conclusions in Section 7.

2 Notation and background

In a bipartite graph $G = (V_1 \cup V_2, E)$, the vertex sets V_1 and V_2 are disjoint and for all edges in E, one of the endpoints belongs to V_1 and the other belongs

to 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 $\Gamma(v) \subseteq V_2$, similarly, 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 in 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 in 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 $|\mathcal{M}|$ is the cardinality of $\mathcal{M}$. Furthermore, if $|\mathcal{M}| = |V_1| = |V_2|$, $\mathcal{M}$ is called 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 [19].

For a given $m \times n$ matrix $\mathbf{A}$, we define $G_{\mathbf{A}} = (V_R \cup V_C, E)$ where $|V_R| = m$, $|V_C| = n$, and $E = \{\{i, j\} \in V_R \times V_C : a_{i,j} \neq 0\}$ as the bipartite graph derived from $\mathbf{A}$. Assuming $\mathbf{A}$ is a square matrix having full structural rank, $G_{\mathbf{A}}$ has a perfect matching, and a *transversal* in $\mathbf{A}$ corresponds to a perfect matching $\mathcal{M}^*$ in $G_{\mathbf{A}}$. Based on this 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 used by Duff et a. [10]. The number of edges in $G_{\mathbf{A}}$ is equal to the number of nonzeros in $\mathbf{A}$ and denoted by τ.

We use the two common data structures for storing sparse matrices [9, Section 2.7] to store the bipartite graphs in our implementations of the matching algorithms. These are called the compressed column storage (CCS) and the compressed row storage (CRS). They store edges of the bipartite graph as the neighborhoods of column or row vertices, respectively. 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 $rids$ where $cptrs[n+1] = \tau + 1$. The row indices of the nonzeros in column j are stored in $rids[cptrs[j], \ldots, cptrs[j+1] - 1]$.

We refer to $rids$ and $cptrs$ as the CCS arrays. 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 $rptrs$, of size respectively τ and $m + 1$, with functions similar to those of the above.

2.1 Maximum cardinality matching algorithms for bipartite graphs

Let $\mathcal{M}$ be a matching in G. 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 a basis for the augmenting path-based algorithms for the maximum matching problem in the literature.

Theorem 1 ([4]). *Let G be a graph (bipartite or not) and let $\mathcal{M}$ 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.*

There are three prominent classes of bipartite matching algorithms: augmenting path-based ones, push-relabel-based ones, and the recently proposed pseudoflow-based ones. Below we briefly mention the main characteristics of these algorithms, and defer further details to later sections.

Algorithms based on augmenting paths follow a common pattern. 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 Theorem 1. 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 $\mathcal{P}$. Since both the first and the last edge of $\mathcal{P}$ were unmatched in $\mathcal{M}$, we have $|\mathcal{M} \oplus E(\mathcal{P})| = |\mathcal{M}| + 1$. 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. In this paper, we use `PFP`, the fastest augmenting path-based matching algorithm identified by Duff et al. [10]. `PFP` is a variant of the algorithm of Pothen and Fan [22], and is described in Section 5.1.

Push-relabel algorithms on the other hand search and augment simultaneously. 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 $|\mathcal{M}|$. The speculative augmentation operations are performed until no further suffixes can be found. These operations are guided by assigning a label to every vertex which provides an estimate of the distance to the nearest unmatched row (i.e., to a potential suffix).

The original push-relabel algorithm by Goldberg and Tarjan [11] was designed for the maximum flow problem. Since bipartite matching is a special case of maximum flow, it can be solved by this algorithm. In fact, it is known to be one of the fastest algorithms for bipartite matching, as was shown by Cherkassky et al. [7]. In this paper, we study the performance of the best variant identified in our technical report [16]. We will discuss the simple push-relabel algorithm (`PR`) and its extensions in detail in Sections 3 and 4.

The pseudoflow-based bipartite matching algorithms progress in a way similar to `PR`. In the matching context, they can be described as building trees containing prefixes and suffixes of augmenting paths. When a prefix- and a suffix-tree connect, an augmenting path is found. Different variants of the pseudoflow algorithm differ in the size of the trees constructed, as well as in the fashion of constructing them. Like `PR`, the pseudoflow approach was originally developed for the maximum flow problem.

For a short summary on some other algorithms and approaches for the bipartite graph matching problem, we refer the reader to reference [10, Section 3.4].

2.2 Initialization heuristics

Almost all matching algorithms start with an empty matching and find matchings of successively increasing size in some fashion, the algorithms studied here being no exception. These successive steps are self-contained. Thus, these algorithms can be initiated with a non-empty matching. In order to exploit this,

several efficient and effective heuristics, which find initial matchings of considerable size, have been proposed in the literature [15, 18, 20, 22, 23].

In this paper, we use two different 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`. It was proposed by Karp and Sipser [15]. It is similar to `SGM`, but it keeps track of the vertices with a single unmatched adjacent vertex and immediately matches these. Theoretical studies by Aronson et al. [2] and Karp and Sipser [15] show that `KSM` is highly likely to find perfect matchings in random graphs, and in practice, it is significantly more effective than `SGM`. The `SGM` algorithm needs only CCS (or CRS), however, `KSM` needs both data structures.

These heuristics have seen extensive experimental investigations, among others by Duff et al. [10], Langguth et al. [18], and Magun [20]. There are extended version of these heuristics [18, 20]. However, none of the extended heuristics could be shown to consistently provide performance superior to `KSM` or `SGM`. Therefore, only these two are considered here.

3 The push-relabel algorithm for bipartite matching

Cherkassky et al. [7] describe the (simplified) push-relabel algorithm for the bipartite matching problem. In the following, we carefully portray this algorithm in a ready-to-implement pseudocode form as shown in Algorithm 1. This algorithm will be referred to as `PR` throughout the paper. Section 4 contains several extensions of `PR` which were used in the experiments.

Input: A bipartite graph $G = (V_R \cup V_C, E)$ and a,
possibly empty, matching $\mathcal{M}$ in G
Output: A maximum cardinality matching $\mathcal{M}^\star$

```
Set ψ(u) = 0 for all u ∈ V_R
Set ψ(v) = 1 for all v ∈ V_C
Set all v ∈ V_C unmatched by M to active
while an active column v exists do
    Find a row u ∈ Γ(v) of minimum ψ(u)
    if ψ(u) < m + n 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        ►Relabels u to obtain an admissible incident edge
    Set v inactive
return M* = M
```

Algorithm 1: `PR: Push-Relabel Algorithm for Bipartite Matching`

Let $\psi : V_R \cup V_C \to \mathbb{N}$ be a distance labeling used to estimate the distance and thereby the direction of the closest unmatched row for each vertex. This

labeling constitutes a lower bound on the length of an alternating path from a vertex v to an unmatched row. If v is an unmatched column, such a path is also an augmenting path. During initialization, the algorithm sets $\psi(v) = 1$ for all $v \in V_C$ and $\psi(v) = 0$ for all $v \in V_R$. We call unmatched columns active. Now, as long as there are active columns, the algorithm repeatedly selects one of them and performs the *push* operation on it.

To perform a push on an unmatched column v, we search $\Gamma(v)$ for a row $u \in \Gamma(v)$ with the minimum $\psi(u)$. Note that $\psi(v) - 1$ is the infimum for the value of $\psi(u)$. This holds after the initialization ($\psi(u) = 0$ and $\psi(v) = 1$ for all $u \in V_R$ and $v \in V_C$) and is maintained throughout the algorithm as an invariant. 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 $u \in \Gamma(v)$ has minimum ψ and is not matched, 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 operation is called a *single push*. On the other hand, if u is matched to a column vertex w, we perform a *double push*. This operation removes $\{w, u\}$ from $\mathcal{M}$, adds $\{v, u\}$ to $\mathcal{M}$, and makes w active. The double pushes ensure that once a row is matched, it can never become unmatched again—the cardinality of $\mathcal{M}$ can never decrease. Note that $\psi(u) = 0$ for an unmatched row vertex u, i.e., such a vertex will always have the minimum ψ value.

If there is no admissible row u among the neighbors of v, i.e., any row u having minimum $\psi(u)$ has $\psi(u) > \psi(v) - 1$, we set $\psi(v)$ to $\psi(u) + 1$. 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, it must contain some $u' \in \Gamma(v)$. 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 its closest unmatched row, and $\psi(v)$ is updated accordingly.

By the same token, u is relabeled by increasing $\psi(u)$ by two 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 two after a single push. In case of a double push, any alternating path from u to a closest unmatched row now contains v. As any such path starts with an unmatched edge on an unmatched row and 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 two yields a correct new lower bound. Clearly, this increase maintains 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.

If $\psi(u) \geq m + n$ for the minimum $\psi(u)$ among the neighbors of v, instead of performing a push or relabel, v is considered unmatchable and marked as inactive. The reason for this is that the maximum length of any augmenting path in G is at most $\min(2m, 2n) - 1$. Since ψ is a lower bound on the length of a path to an unmatched row, and $\psi(u) \geq m + n$ for all neighbors of v, no

augmenting path can start at v. As v remains 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. Using Theorem 1, it is easy to show that in this case $\mathcal{M}$ is a maximum matching. The time complexity of the algorithm is $\mathcal{O}(n\tau)$ [11].

As discussed above, one needs to store the column labels. In order to reduce jumps and arithmetic operations, we stored the row labels as well. Our implementation therefore uses $m + n$ integer space in addition to the CCS arrays. We also keep the matching partners of rows and columns in arrays, requiring additional $m + n$ space.

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 update the distance labeling ψ. Both are well studied in the literature [7, 17]. 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 according to 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 restrict ourselves to `FIFO` ordering which was found to be superior (see the technical report [16]). An additional memory space of size n is required to implement the `FIFO` ordering, making the total memory requirement $m + 2n$ integers (on top of the CCS and matching arrays).

4.2 Global relabeling

The performance of the `PR` algorithm can be improved by periodically setting all labels to exact distances. This is called *global relabeling* and is accomplished by running a BFS starting from the unmatched rows, as shown in Algorithm 2. 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 $\psi(w) = m + n$, thereby removing it from further consideration.

In order to keep track of the number of pushes executed, a counter is incremented every time the value of $\psi(v)$ is changed in Line 7 of Algorithm 1. Thus, pushes along admissible edges are not counted. Note that single pushes

are always along admissible edges. When the counter reaches a predetermined threshold, we call the `Global Relabeling` procedure.

Input: A bipartite graph $G = (V_C \cup V_R, E)$ and a matching $\mathcal{M}$ in G
Output: An accurate distance labeling ψ w.r.t. $\mathcal{M}$

```
Q ← u for all unmatched u ∈ V_R
Set ψ(v) = m + n for all v ∈ V_C
Set ψ(u) = m + n for all matched u ∈ V_R
while Q not empty do
    u ← POP u from Q
    for all v ∈ Γ(u) do
        if ψ(v) = m + n then
            ψ(v) ← ψ(u) + 1
            if {v, w} ∈ M then
                ψ(w) ← ψ(v) + 1
                PUSH w to Q
return ψ
```

Algorithm 2: : Global Relabeling

It is a well-established fact that global relabelings are essential for practical performance, and preliminary tests reconfirmed this. Thus, our `PR` codes use periodic global relabeling. In [7], a threshold of n was suggested as the standard frequency of global relabels.

Since our implementation makes use of the double push technique, we need to adopt a counting scheme that differs slightly from the standard `PR` algorithm. We only count the number of double pushes in which the first edge was not admissible. The second edge, which started out as matched, would always require a relabel prior to a push, unless its label was changed by a global relabeling between it becoming matched and the current double push. Since the row vertex is relabeled immediately after a double push due to performance reasons, it is impossible to accurately reflect this in the count. Therefore, we count double pushes only once and reflect this in the thresholds used. Note that single pushes always use admissible edges, and are therefore never counted against the threshold.

Since we use rectangular matrices as test instances, we must consider the case $m \neq n$. Preliminary experiments showed no noticeable difference between using relabeling frequencies of m and n. Thus, we use a base threshold of $m+n$. The corresponding relabeling frequency is denoted as `RF=1`.

In our experiments, we compare the base relabeling frequency with multiples thereof. We use `RF=1.5`, `RF=2`, `RF=4`, and `RF=8` in our experiments. As suggested in the technical report [16], these values are likely to produce good results for the test instances studied.

The global relabeling operation requires another array of size m to maintain a queue of the discovered row vertices. Furthermore, as the BFS is run from the row vertices, the CSR storage is also required. Therefore, in addition to CCS (with $1+n+\tau$ integers), CSR ($1+m+\tau$ integers), and matching arrays ($m+n$ integers), a total of $2m + 2n$ integer space is required to implement `PR-FIFO` with global relabeling.

4.3 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 of whether the algorithm could be improved by encouraging fairness in neighbor selection. This was proposed by Duff et al. [10] for improving the Pothen and Fan (PF) algorithm [22] and resulted in significant performance gain (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. This technique can be implemented without extra storage requirement.

The fairness technique can also be applied in the PR algorithm during the neighbor selection process. We study this Fair variant and compare it to other PR implementations in Section 6.

4.4 Search spread

In push-relabel algorithms designed for the maximum flow problems, a different technique is used to equilibrate searches over the adjacency lists. In our setting, this can be described as follows. Every vertex v maintains a pointer $p(v)$ which is set to its first incident edge on initialization. The search for a neighbor of minimum label always starts with the edge to which $p(v)$ points. If an admissible edge is found, the search is stopped and $p(v)$ is set to the next edge in the list of edges incident to v. This guarantees that the search is spread out more evenly among incident edges, making it more likely that an admissible edge is found quickly. If a search starting from $p(v)$ reaches the end of the adjacency list belonging to v without finding an admissible edge, it continues at the start of the adjacency list and proceeds up to $p(v)$. However, if a neighbor u with $\psi(u) = \psi(v) + 1$ is found, this latter part can be skipped since no admissible edge, and thus no neighbor with $\psi(u) < \psi(v) + 1$ exists. To see this, remember that a neighbor u having $\psi(u) = \psi(v) - 1$ implies an admissible edge $\{v, u\}$, and that $\psi(u)$ is always incremented by 2.

Preliminary experiments showed promising results for the fairness technique described in Section 4.3. Therefore, we combined both techniques, obtaining the Fair-Spread variant of push relabel. This leads to a somewhat more complicated implementation because $p(v)$ now switches between acting as the starting point and the endpoint of a search.

In order to improve clarity of the code, we implemented the combined technique described above using two additional arrays of size n each. However, only one additional array is required.

5 Other algorithms

In this section we discuss the algorithms that we use for comparison with PR. We use only the fastest known algorithms for our experiments. PFP, the modified Pothen-Fan algorithm was found to be superior to all other augmenting path-based algorithms [10, 16]. We also established [16] that the PR-FIFO variant is superior to other PR algorithms. We briefly describe PFP, the augmenting path-based algorithm used for experimental comparison. For a more detailed

description, we refer the reader to [10]. Descriptions of other augmenting path-based algorithms can be found in e.g., [1, 10, 13, 21, 22].

An alternative approach is the pseudoflow algorithm, which was introduced by Hochbaum [12]. It is similar to PR in that it maintains distance labels and performs a specialized type of push. Chandran and Hochbaum [6] found the *free arcs* variant to be superior for bipartite matching on difficult instances. Below, we will briefly describe this variant. We will also discuss two other variants which we use for comparison in our experiments.

5.1 PFP: a matching algorithms based on augmenting paths

The Pothen-Fan algorithm, denoted as PF, is based on repeated phases of depth-first searches [22]. At a phase, PF 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 each 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. After this, a new phase starts.

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 unmatched 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 time complexity of $\mathcal{O}(n\tau)$.

In each DFS, the rows adjacent to a column are visited according to their order in the adjacency list. This is true even if there is an unmatched row among them. In order to reach such an unmatched row, 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 [8, 22]. It works as follows: every vertex v maintains a lookahead pointer $l(v)$. Initially $l(v)$ is set to the first neighbor in the adjacency list of v. When v is visited, the algorithm first checks if $l(v)$ is unmatched. If not, it iterates over the adjacency list of v until an unmatched row is found. If there is such a vertex, an augmenting path has been discovered and the current search stops. Otherwise, if $l(v)$ reaches the end of the adjacency list the search continues with the usual DFS process. In that case $l(v)$ is not considered any further.

The above algorithm using the lookahead technique is known as Pothen and Fan's algorithm. Duff et al. [10] found the algorithm to be efficient for matrices from real life applications, except that its running times vary widely when row or column permutations have been applied to the matrix. To alleviate this, they suggested to modify the order of visiting the rows in the adjacency lists of columns by applying an alternating scheme called *fairness*. It is identical to the technique described in Section 4.3. Note that fairness neither changes the complexity nor the memory requirements of PF. It usually improves the performance of PF and in some cases it results in remarkable speedups, while the required overhead remains negligible [10]. This algorithm is referred to as PFP. We use this variant of PF exclusively.

The implementation of PFP given by [10] uses integer arrays of total size $m + 4n$ in addition to the CCS and matching arrays. The algorithm does not

need CRS itself. However, we always initialize it using KSM, which needs both CCS and CRS.

5.2 The pseudoflow algorithm

The pseudoflow algorithm was introduced by Hochbaum [12] for the maximum flow problem. It incorporates notions developed for the push-relabel algorithm, among them the distance labeling ψ and the admissible edge definition. In the bipartite matching context, the algorithm can be simplified. We describe the simplified free-arcs variant of the pseudoflow algorithm, as this one was the fastest (reported in [6]) for the bipartite matching problem.

All vertices v start out as unmatched and having $\psi(v) = 1$. Similar to PR, unmatched columns are marked as active and processed in a given order, e.g., the lowest label first. An active vertex v scans its adjacency list for admissible edges and, if necessary increases its label $\psi(v)$ such that an edge leading to a lowest labeled neighbor becomes admissible. Let $\{v, u\}$ be an admissible edge found in this manner. If u is unmatched, v and u are matched along $\{v, u\}$ and v becomes inactive. This is equivalent to a single push. Otherwise, u becomes overmatched. Let w be the original matching partner of u. Unlike during a double push in the PR algorithm, w now remains matched to u. Next, v and w both become active and $\psi(u)$ is increased to $\psi(v) + 1$.

Now assume w (or equivalently v) is processed and an admissible edge $\{w, x\}$ is found. If x is unmatched, then $\{w, u\}$ becomes unmatched, while $\{w, x\}$ becomes matched, resulting in two standard matching edges $\{v, u\}$ and $\{w, x\}$. If x was already matched to some other vertex y, $\{v, u\}$ becomes a standard matching edge while v, x, and y now form a new path of length two where v and x are active.

The process continues until all active vertices have been matched along standard matching edges, thereby becoming inactive or their labels have increased to $m+n$. Similar to PR, an active vertex v with $\psi(v) = m+n$ is set to inactive. If no active vertices remain, the algorithm terminates. The worst-case running time of this algorithm is $O(n\tau)$.

In addition to the free-arcs variant, several alternatives are described by Chandran and Hochbaum [6]. The difference compared to the free-arcs variant described above lies in the fact that these algorithms are able to build larger trees than the length two paths described above. Similar to PR, the pseudoflow algorithm repeatedly processes active vertices. Thus, different strategies of selecting active vertices are possible. Both the highest-label-first and the lowest-label-first are used by Chandran and Hochbaum [6]. Active vertices are kept in buckets. The buckets can be implemented either as FIFO queues or as LIFO stacks. We select the highest-label-first variant with LIFO buckets, which is referred to as the HI_WAVE variant in [6]. We also use LO_LIFO, the lowest-label-first variant with LIFO buckets. The free-arcs variant is referred to as LO_FREE. Other variants were found to be inferior in [5]. We confirmed this in preliminary experiments by studying the HI_FIFO, LO_FIFO, and HI_FREE variant. Overall performance was about 20% inferior to the HI_WAVE, LO_LIFO, and LO_FREE versions that we study in this paper.

We use the implementation of the pseudoflow algorithms accompanying the paper by Chandran and Hochbaum [6] which were available at http://riot.ieor.berkeley.edu/Applications/Pseudoflow/maxflow.html at the time of

writing. The implementation given there uses eight fields (mixture of integers and pointers) per vertex to store once class of vertices (say row vertices) and uses ten fields (mixture of integer and pointers) per vertex to store the other class of vertices. Furthermore, adjacency lists are stored for the rows and for the columns. Four additional arrays, of size n (or m) each, are used during the algorithm. Overall, the total memory requirement is $8m + 14n + 2\tau$. We identified one field in each vertex class as redundant for our applications in sparse matrices; however we did not see an easy way to reclaim the space used by other fields and the four arrays. Therefore, we deem it accurate to state that the space requirement of a reasonable pseudoflow-based matching algorithm is $7m + 13n + 2\tau$.

6 Experiments

6.1 Experimental setup

All of the algorithms and heuristics are implemented in the C programming language. Codes are called via a Matlab interface. We compiled the codes with *mex* of Matlab using *gcc* version 4.4.2 with the optimization flag -O and ran the compiled codes on a machine with a 2.4 Ghz AMD Opteron 250 processor and 8 Gbytes of RAM. As an additional test system, an Intel Xeon E5520 Quad Core computer running at 2.27 Ghz was used. Differences in the results were marginal, and thus they are not presented here.

For the experiments, we use real life $m \times n$ matrices from the University of Florida Sparse Matrix Collection (`http://www.cise.ufl.edu/research/sparse/matrices/`). We only consider matrices having more than 50000 columns. Due to the comparatively steep memory requirements of the pseudoflow codes, it was necessary to limit the maximum number of columns to 12 million and the maximum number of nonzeros to 120 million. Currently a total of 437 matrices satisfy these assertions, with 53 among them being rectangular.

On average, the matrices have approximately $600,000$ rows and an equal number of columns and close to 10 million edges. The respective median values are $155,000$ and 2.7 million.

For each matrix, we perform four sets of experiments. First, we execute all algorithms on the original matrix (denoted by "No perm"). Second, we apply five random row permutations (denoted by "Row perm") to the original matrix and execute the algorithms for each row-permuted matrix. Third, we apply five random column permutations (denoted by "Col perm") to the original matrix and execute the algorithms for each column-permuted matrix. Fourth, we apply five random row and column permutations (denoted by "Row+Col perm") and execute the algorithms for each totally permuted matrix. For each algorithm, the average running time and operation counts of five permutations is stored as the running time or operation count of the algorithm on a matrix with a given permutation type.

Although our focus is on the maximum transversal problem for real-life instances, we perform smaller confirmation experiments on bipartite random instances. These are described later.

6.2 Measurements

To compare the algorithms, we measure both running times and operation counts. However, due to the different structure of the algorithms, not all operations are comparable across the algorithms.

For `PFP` and `PR`, the total running time is divided into three parts. First A^T, the transpose of the input matrix A must be computed in order to obtain the CRS array of A. It is necessary for global relabelings in `PR` and for the `KSM` initialization heuristic. Thus, the first step requires an identical amount of work for `PFP` and `PR` algorithms. We then obtain the time required for the `KSM` or `SGM` initialization heuristic and then the time for the main algorithm.

It is possible to initialize `PFP` with `SGM`. In this case the first step can be skipped, because neither `PFP` nor `SGM` require adjacency lists for the rows. However, this was found to be inferior for non-trivial instances [10, 16]. Thus, we do not use this combination here.

The pseudoflow algorithms also perform three steps. The first step consists of building up sophisticated data structures, and thus requires considerably more effort than taking the transpose. The second step is always an `SGM` style initialization that works on these data structures. In the third step, the actual algorithm is called and its running time is measured.

In addition to the running times reported above, we considered machine independent operation counts as a measurement of algorithm performance. All algorithms repeatedly search through adjacency lists in order to check adjacent vertices. Therefore, such edge operations, which are commonly referred to in the literature as *Arc Fetches* or *Arc Scans* are counted for all algorithms. For `PFP`, the second relevant operation is obtaining the symmetric difference between the current matching $\mathcal{M}$ and an augmenting path. We refer to matching an edge while unmatching another as an augmentation. Since the number of augmenting paths to be used is equal to the deficiency of the initialization, the number of augmentations depends on the length of the augmenting paths found. We report this number for `PFP`.

For `PR`, instead of augmentations we have double pushes which require slightly larger effort because labels have to be updated. On the other hand, arc scans tend to be less expensive as often the entire adjacency list of a vertex is scanned, which is quite cache-efficient. The same is true for arc scans performed during a global relabel, which progresses in a BFS fashion, as opposed to the DFS of `PFP`. Furthermore, for `PFP` it is necessary to mark the augmenting path currently under construction during the search.

For the pseudoflow algorithms, we again report the number of arc scans. In addition, the codes report the number of additional operations, namely *relabels*, *pushes*, and *mergers*. These are not directly comparable to double pushes, but each of them is at least as expensive as an arc scan.

Because the initialization heuristics usually match at least 95% of the vertices, the operation of matching a vertex for the first time, i.e., a single push in `PR` or its equivalent in the other algorithms is not counted since its number is hardly significant.

6.3 Running time

The running time results over the entire test set are given in Table 1. We measure running time in seconds. Reported timings include the time to build up data structures, and heuristic initialization, but not file reading. We give both the average and the median running time. Since all algorithms have superlinear running times, it is clear that average values are significantly higher than median values. The ratio of the average running time to the median running time is 10.54 on average (Table 5 in the appendix lists all these ratios). Algorithms for which the discrepancy between average and median is low can be regarded as stable with respect to different instances. In general, PR shows higher ratios than the other algorithms. Note however that the correlation between instance size and running time is rather weak. Figure 4 in the appendix shows scatter-plots of running times given instance size, illustrating their weak correlation.

Table 1: Median and average running times in seconds over the entire test set. Values contain data setup, initialization and main algorithm time. Detailed results are given for the various permutation types. Optimum values are denoted in **boldface**

Algorithm	RF	No perm		Row perm		Col perm		Row + Col perm		Average	
		Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.
PR	1	1.13	0.102	3.41	0.316	10.15	0.510	**3.74**	0.442	4.61	0.308
SGM	1.5	1.05	0.096	3.47	0.302	9.79	0.490	3.86	**0.440**	4.54	0.298
	2	0.99	0.094	3.23	0.324	9.47	0.492	3.97	**0.440**	4.42	0.292
	4	0.94	0.090	3.24	0.400	4.27	0.524	4.32	0.574	3.19	0.341
	8	0.95	0.088	3.10	0.368	4.57	0.498	4.12	0.488	3.18	0.316
PR-Fair	1	1.11	0.096	4.90	0.322	4.36	0.374	4.93	0.448	3.82	0.278
SGM	1.5	1.00	0.094	4.94	**0.302**	4.35	0.368	5.04	0.452	3.83	**0.274**
	2	0.98	0.088	3.68	0.330	**3.12**	**0.364**	5.06	0.424	3.21	0.277
	4	**0.92**	**0.086**	3.16	0.368	3.88	0.404	4.19	0.528	3.04	0.303
	8	0.93	0.088	3.31	0.412	3.41	0.464	4.32	0.552	2.99	0.328
PRFair	1	1.12	0.100	2.88	0.320	3.25	0.386	3.77	0.454	**2.76**	0.283
-Spread	1.5	1.03	0.098	2.95	0.318	3.33	0.384	3.88	0.454	2.79	0.284
SGM	2	1.01	0.098	3.10	0.342	3.32	0.386	4.06	0.462	2.87	0.288
	4	0.96	0.088	3.10	0.374	3.94	0.494	4.03	0.490	3.01	0.316
	8	0.94	0.100	3.34	0.422	3.47	0.460	4.35	0.558	3.02	0.336
Other algs.											
PFP		1.21	0.128	**2.74**	0.364	3.79	0.480	3.86	0.570	2.90	0.362
HI_WAVE		5.95	0.814	6.83	1.054	7.72	1.160	6.17	0.898	6.67	0.973
LO_LIFO		6.71	0.756	9.47	1.144	9.38	1.238	6.99	0.848	8.14	0.999
LO_FREE		4.75	0.772	3.46	0.650	8.16	1.198	4.92	0.946	5.32	0.888

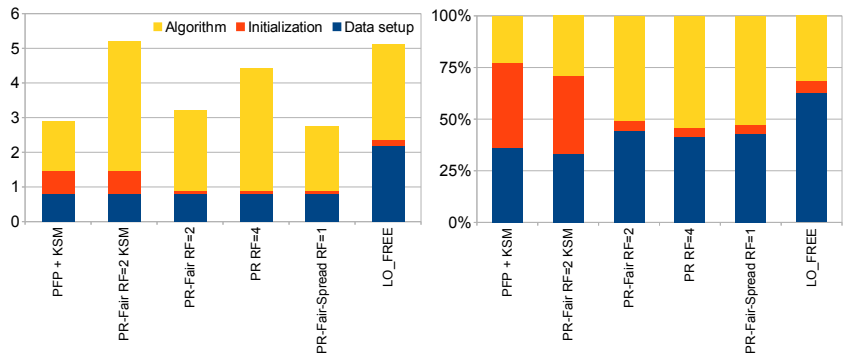
Overall, PRFair-Spread with low relabeling frequency shows the best results. It dominates PFP, and shows far better average and only slightly worse median results than PRFair. It is also superior to PR without fairness. However, all these algorithms are relatively close in performance. The pseudoflow codes show far lower performance. They also show smaller relative variance in running time. Due to the fact that KSM initialization consistently provides much better initializations than SGM, PFP shows a good average running time, but its median performance is lower than that of most PR codes.

We observed that using KSM initialization is not competitive for PR (see also the report [16]). Interestingly, KSM initialization not only leads to higher median running times, but also to very high average values. Timing results corresponding to Table 1 can be found in Table 3 in the appendix. It was also observed [16] that using SGM initialization followed by a global relabeling is generally prefer-

able to starting PR with an empty matching. Therefore, we do not consider this alternative here.

The PR algorithm is quite sensitive to the frequency of the global relabelings. If the fairness technique is not used, a setting of $RF = 4$ or $RF = 8$ is preferable. With fairness, we observe maximum performance at $RF = 2$ or lower. If this is used, the fairness technique clearly improves median performance. However, the best results are obtained by using search spread and $RF = 1$. Therefore, in the following we will focus on PR-Fair-Spread at $RF = 1$ and PR-Fair at $RF = 2$ when discussing the PR algorithm. In addition, we will study PR without fairness at $RF = 4$. Interestingly, for the unpermuted matrices, setting $RF = 8$ yields the best results, while the fairness and the search spread mechanisms have no noticeable effect.

Concerning permuted matrices, we observe that PFP is faster than PR on row permuted matrices, roughly equal on row-column permuted matrices, and slower on column permuted or original matrices. These differences are the result of the different algorithmic techniques. On original matrices, the BFS based global relabeling used in PR is obviously very effective. Increasing its frequency makes PR even faster. However, as it works on the row adjacency lists, PR is slower than PFP under pure row permutations, since PFP does not work with the row adjacency lists at all, except during KSM initialization.



(a) Average time in seconds (b) Median percentages

Figure 1: Comparison of the division of 1(a) average running time (in seconds) and 1(b) median running time of the different parts of the principal algorithms studied. In each bar, the lowest segment is the data set up time, the middle segment is the initialization heuristic's running time, and the highest segment is the algorithm's running time.

On the other hand, the fact that PFP works only on the column adjacency lists makes it more susceptible to pure column permutations than PR, where augmentations are guided by labels which are updated during global relabelings. Still, PR is affected considerably by column permutations. Consequently, having both row and column permutations is the hardest case for PR, and its performance is lowest here. PFP is not affected by the addition of row permutations, and shows roughly the same performance as for pure column permutation, which is comparable to that of PR on such instances. Note that without fairness, PR shows very low performance under pure column permutations. In the

absence of column permutations, fairness has little effect.

The division of running time for the different algorithms is illustrated in Figure 1. We observe that transposing the matrix consumes a significant amount of running time. However, as both PR and PFP require this step, it does not affect their relative comparison. Setting up the data structures for LO_FREE is even more expensive as shown in Figure 1(a).

Next, we see that KSM initialization is quite expensive, taking up almost half the time for the main computation of PFP. On the other hand, using SGM is very fast. However, KSM initialization results in PFP having the shortest main computation, making it competitive with the Fair PR codes while LO_FREE has the longest main computation time, rendering it uncompetitive.

In addition to the averages, we study the behavior on the worst cases, i.e., on the original matrices or permutations of them that are the most time consuming for each algorithm. Figure 2 indicates that on the 50 most difficult instances PFP is slightly faster than PR. LO_FREE starts about 50% slower, but on the hardest 5 instances its running time increases considerably. PR without the fairness mechanism is generally slower than PR-Fair. Both variants have very high running times on permutations of the instance *circuit5M* (see Table 6). Otherwise, the overall worst case picture closely matches the findings obtained from studying the average running times.

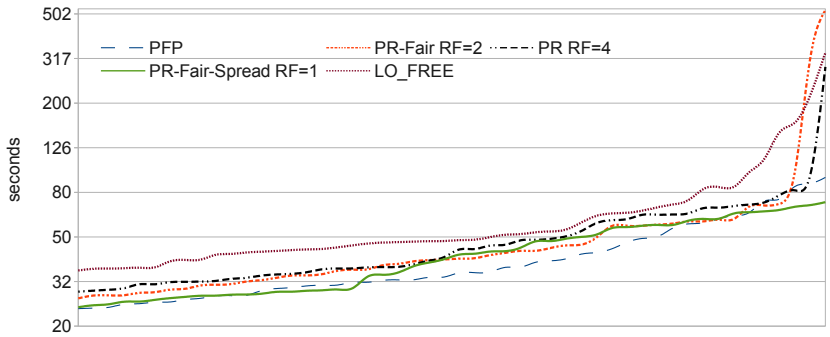


Figure 2: Comparison of the 50 most time consuming instances and permutations for each algorithm, sorted along the x-axis. Running time is given in seconds; grid lines are on a logarithmic scale. PFP and PR-Fair-Spread are very fast, even for their respective worst cases. LO_FREE starts about 50% slower, but on the hardest 5 instances its running time increases considerably. PR without the fairness mechanism is generally slower than PR-Fair, but both variants attain extremely high worst case running time.

In order to study the instances that are solved quickly by most algorithms we give the performance profile of the algorithms in Figure 3. All PR algorithms have minimum running time over all algorithms in about 30% of the instances. For PFP, this figure is about 20%. PFP remains consistently slower than the PR algorithms by a small margin. Among these, we see that PR-Fair without Spread has a slightly better performance. This is not surprising since the Spread technique costs some additional overhead which only pays off in the worst cases, as seen in Figure 2. Consistent with its average performance, LO_FREE remains far slower than all alternatives. LO_FREE's running time is smaller than twice the

running time of the fastest algorithm in only about 20% of the cases, whereas the corresponding percentages for PFP and PR-Fair are 85% and 95%, respectively. In conclusion, we see that algorithms which show good performance in this profile also have low median running times in Table 1.

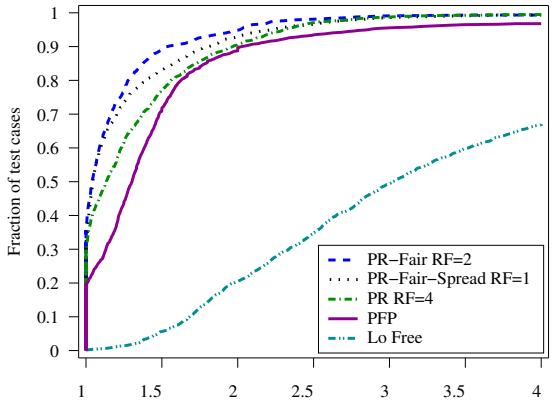


Figure 3: Performance profile for the principal algorithms. It denotes the fraction of instances for which an algorithm is within a given factor of the best algorithm for that instance. The factors are denoted on the x-axis, while the y-axis shows the fraction of instances an algorithm solves within this limit.

In addition to the actual running times, we take a closer look at instances where the running times of different algorithms diverge. Table 6 in the appendix lists all these instances where the slowest algorithm's running time is 100 times larger than those of the fastest one. In most cases, this happens due to fast running times for PR or PFP on the original matrices. In addition, HI_WAVE and LO_LIFO require very long running times to solve any of the permutations of *rajat29* and *t2em*. For some PR codes, *circuit5M* is extremely time consuming.

6.4 Operation counts

The differences in operation counts for the various algorithms largely resemble the differences in running time. Average results over the entire test set are given in Table 2. We first observe that the PFP algorithm requires a comparatively small amount of arc scans and matching operations. However, since this number does not include approximately τ arc scans required by the KSM initialization, we have to add the average number of edges in the test set, which is approximately 10 million, to the average of 31 million arc scans performed by the algorithm. This puts PFP+KSM close to the best PR code, which applies about 38 million arc scans and 2 million double pushes on average. Now, considering that the BFS based relabelings and lowest label searches are somewhat more cache efficient than the DFS based operations in PFP, we see that the operation counts are well comparable between the algorithms and are suited to gauge their performance.

The performance of the PR algorithm in terms of operation counts is also quite sensitive to the frequency of the global relabelings. A higher relabeling frequency means significantly more global relabel arc scans, but also a greatly reduced number of regular arc scans and double pushes. Using a least squares

Table 2: Average operation counts over the entire test set. Detailed results are given for the various permutation types. AS denotes *arc scans* and DP denote *double pushes* in push relabel algorithms. For other algorithms, AS and the sum of other operations is given. PR results for KSM initializations can be found in Table 4 in the appendix.

Algorithm	RF	No perm		Row perm		Col perm		Row + Col perm		Average	
		AS	DP	AS	DP	AS	DP	AS	DP	AS	DP
PR	1	28.95	1.319	59.25	2.376	149.94	4.579	81.88	2.017	79.82	2.569
SGM	1.5	26.93	1.133	60.28	2.336	123.77	3.985	70.30	2.064	70.22	2.376
	2	24.39	0.868	65.90	2.200	104.88	3.199	51.09	1.943	61.48	2.051
	4	24.40	0.619	63.69	1.692	93.55	2.193	47.25	1.442	57.13	1.485
PR-Fair	1	25.81	1.151	62.49	2.552	84.56	2.933	67.14	2.248	59.93	2.219
SGM	1.5	23.01	0.869	59.88	2.519	71.82	2.767	56.52	2.238	52.72	2.096
	2	24.57	0.830	60.65	2.265	63.64	2.285	47.56	1.973	49.07	1.836
	4	23.71	0.552	62.53	1.690	64.32	1.701	48.81	1.438	49.77	1.344
PRFair	1	23.68	1.003	47.97	2.546	48.38	2.695	33.40	2.094	38.34	2.083
-Spread	1.5	25.44	0.838	49.27	2.447	52.60	2.718	37.68	2.077	41.22	2.018
SGM	2	23.59	0.775	53.80	2.186	54.13	2.328	38.25	1.905	42.42	1.797
	4	23.37	0.522	63.00	1.649	62.95	1.690	46.95	1.416	49.03	1.317
Other algs.											
PFP		11.82	0.493	34.99	1.900	59.24	2.057	18.54	0.943	31.10	1.345
HI_WAVE		23.59	9.368	67.21	15.908	67.51	14.601	21.06	10.828	44.80	12.671
LO_LIFO		30.84	13.114	103.56	23.969	97.30	18.315	20.50	12.975	63.01	17.084
LO_FREE		130.49	24.403	27.12	8.383	157.14	31.988	16.92	6.467	83.04	17.824

estimate on the experimental data, we asserted that in our implementation the computational cost of a double push is roughly equivalent to that of ten arc scans. Therefore, frequent global relabels are likely to pay off as long as they can substantially reduce the number of double pushes.

For the pseudoflow based codes, the operation count comparison is somewhat more difficult due to the nature of underlying complex operations. Nonetheless, the arc scan operation can still be used for comparisons. The low performance of the LO_FREE code compared to PFP and PR can be explained by the fact that it performs about 82 million arc scans. However, HI_WAVE and LO_LIFO use significantly fewer arc scans than LO_FREE, yet their performance is inferior since the cost of the other operations is higher than that of the arc scans. Therefore, these counts cannot explain the difference in performance. The difference is due to the different trees generated by the corresponding split operations [6].

The operation counts are also well suited to indicate the difference between the PR algorithms. We observe that the fairness and the search spread mechanisms reduce the average number of operations. However, in the case of search spread, this benefit comes at the cost of slightly slower search through the adjacency lists, which is not captured by the operation counts. We also observe that using higher global relabeling frequencies universally reduces the number of double pushes. However, the effects on the total number of arc scans are varied.

6.5 Pseudoflow algorithms

We observe that the three pseudoflow algorithms perform comparatively poorly in this study. In contrast, their performance was found to be superior to all tested alternatives [6], including the PR algorithm. This discrepancy cannot be explained with differences in initialization since even the main algorithm's time for the pseudoflow algorithms is higher than the total time required for PR or

PFP solutions on the original matrices. Furthermore, we can assume that the pseudoflow codes do not suffer from insufficient algorithm engineering, since they were found to be faster than known good push-relabel codes in [6].

A likely explanation is that Chandran and Hochbaum [6] only use the lowest-label variant of PR, not the FIFO variant for comparison, which was found to be substantially faster in our technical report [16].

To exclude the possibility of the results being a consequence of the different test sets, we performed an additional experiment using the *HiLo* and *rbg* random generators [7]. Similar to the largest bipartite graph instances in that study, our test cases have between 1.024 and 1.2 million vertices with average degrees of 5 or 10. Ten random instances were used for the test. Due to KSM initialization PFP was extremely fast here, taking an average of only 0.95 seconds. PR with KSM initialization was similarly fast. The standard SGM initialized PR codes took between 3.32 and 4.39 seconds, while LO_FREE took 9.86. HI_WAVE performed much better than LO_FREE, taking 5.64 seconds on average, while LO_LIFO took 23.45.

Based on the above results, we conclude that the difference in performance between PR and pseudoflow algorithms observed for real-world matrices is also evident in the mentioned random instances. Furthermore, they have higher memory requirements than PR or PFP. In agreement with the results in [6], LO_FREE was generally the fastest of the pseudoflow codes, followed by HI_WAVE and then LO_LIFO.

Finally, we note that their experiments use a relatively old Sun UltraSPARC workstation with a 270 MHz CPU and 192 MB of RAM, which differs substantially from our test systems. However, the high operation counts of LO_FREE indicate that this difference cannot account for the differences in performance.

6.6 Fairness and spread mechanisms

For the original matrices, the fairness mechanisms in the PR algorithm showed little or no effect. However, for the column permuted matrices, the fair PR algorithms were significantly faster. In total, depending on the relabeling frequency, fairness improves average running time by up to 30% or median running time by 10%.

Meanwhile, the search spreading technique yielded noticeable improvements to average running time. It reduces the number of arc scans and double pushes, but it also takes a small amount of extra running time. This manifests as a slight increase in median running time by about 3% w.r.t. PR-Fair, and a decrease in average running time by up to 30% due to much better running time on some hard instances. However, this technique requires more effort to implement, especially in combination with fairness the next arc pointer costs additional memory space.

In conclusion, we recommend that PR should be implemented using both techniques. Their effects on running time noticeably outweigh its cost.

6.7 Relabeling frequency

For the average values, the relabeling frequency has little impact. Optimum RF values lie between 1 and 2. When using the search spread technique, the

optimum value is 1. As observed in our technical report [16], when using techniques that improve the performance of the PR algorithm, relabeling frequency should generally be reduced.

Furthermore, it is interesting to note that for the original matrices, the highest relabeling frequency tested provided superior results, while on the permuted matrices, the lower frequencies generally work better.

We noticed an apparent effect of large matrices benefiting from high relabeling frequencies. However, this is due to the fact that the large matrices in the University of Florida Sparse Matrix Collection are biased towards being more "difficult" (i.e., more time consuming for their size) instances. Specifically, the recently introduced *DIMACS10* group contains many large and difficult matrices. A supplementary experiment on sparse uniformly distributed random matrices generated by Matlab's *sprand* command (i.e., bipartite Erdős-Rényi style graphs) of sizes between 2^{18} and 2^{23} columns and rows and an average degree of 3, 5, and 7 showed no correlation between matrix size and optimum global relabeling frequency. The experiment was performed on the Intel Xeon based secondary test system. Furthermore, as observed in the technical report [16], the optimum relabeling frequency for the random instances is higher than that for the real-world instances. Results can be found in Table 7 in the appendix.

7 Concluding remarks

We have presented the adaptation of the push-relabel algorithm for bipartite matching and introduced simple yet effective techniques to improve its running time. Using the FIFO version, we have investigated its performance in comparison with the state-of-the-art augmenting path- and pseudoflow-based methods on real-world instances.

By experimenting thoroughly on a large number of problem instances arising in real-life applications, we drew several clear conclusions. We established that the augmenting path based algorithm PFP equipped with the KSM initialization heuristic is competitive with the FIFO variant of the PR algorithm using SGM initialization, and these two are preferable to other techniques. Both are tied closely in running time and operation count, and their implementation requires comparable effort. However, using the additional techniques of spread and fairness, PR is slightly faster than PFP. Still, the difference between the augmenting and pushing approach is rather small. Furthermore, proper choice of initialization heuristics often has a greater impact than different algorithmic techniques. On the other hand, our results clearly show that on the real-world test instances used, the pseudoflow codes are not competitive with either PR or PFP.

Bibliography

[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] J. Aronson, A. Frieze, and B. G. Pittel. Maximum matchings in sparse

random graphs: Karp-Sipser revisited. *Random Structures and Algorithms*, 12:111–177, March 1998. ISSN 1042-9832.

[3] A. Azad, J. Langguth, Y. Fang, A. Qi, and A. 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] C. Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the USA*, 43:842–844, 1957.

[5] B. G. Chandran and D. S. Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations Research*, 57(2):358–376, 2009.

[6] B. G. Chandran and D. S. Hochbaum. Practical and theoretical improvements for bipartite matching using the pseudoflow algorithm. *CoRR*, abs/1105.1569, 2011.

[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. *Journal of Experimental Algorithmics*, 3:8, 1998.

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

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

[10] I. S. Duff, K. Kaya, and B. Uçar. Design, implementation, and analysis of maximum transversal algorithms. *ACM Transactions on Mathematical Software*, 38:13:1–13:31, January 2011.

[11] 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.

[12] D. S. Hochbaum. The pseudoflow algorithm and the pseudoflow-based simplex for the maximum flow problem. In *Proceedings of the 6th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 325–337, London, UK, 1998. Springer-Verlag.

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

[14] P. E. John, H. Sachs, and M. Zheng. Kekulé patterns and Clar patterns in bipartite plane graphs. *Journal of Chemical Information and Computer Sciences*, 35(6):1019–1021, 1995.

[15] 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.

[16] K. Kaya, J. Langguth, F. Manne, and B. Uçar. Experiments on push-relabel-based maximum cardinality matching algorithms for bipartite graphs. Technical Report TR/PA/11/33, CERFACS, France, 2011.

[17] R. 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.

[18] 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.

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

[20] J. Magun. Greedy matching algorithms, an experimental study. *Journal of Experimental Algorithmics*, 3:6, 1998.

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

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

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

[24] A. H. Timmer and J. A. G. Jess. Exact scheduling strategies based on bipartite graph matching. In *Proceedings of the 1995 European conference on Design and Test*, EDTC '95, pages 42–47, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7039-8.

APPENDIX

We give some detailed results here. Tables 3 and 4 list the performance of the PR codes in terms of running times and the operation counts, respectively, when the initialization heuristic is KSM. Table 7 gives the results for the optimum relabeling frequency experiment on random bipartite graphs.

Table 3: Median and average running time in seconds over the entire test set for KSM initialized PR codes. Results contain data setup, initialization and main algorithm time. Detailed results are given for the various permutation types.

Algorithm	RF	No perm		Row perm		Col perm		Row + Col perm		Average	
		Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.	Avg.	Med.
PR	1	1.37	0.132	10.06	0.380	19.42	0.647	8.76	0.562	9.90	0.385
KSM	1.5	1.35	0.126	8.09	0.376	17.97	0.634	7.97	0.570	8.84	0.384
	2	1.26	0.130	5.71	0.384	16.42	0.646	6.81	0.582	7.55	0.378
PR-Fair	1	1.30	0.122	13.79	0.382	4.27	0.506	13.00	0.558	8.09	0.371
KSM	1.5	1.28	0.122	8.39	0.366	5.10	0.502	9.46	0.570	6.06	0.366
	2	1.20	0.122	7.55	0.394	3.82	0.510	8.17	0.564	5.18	0.372
PR-Fair-Spread	1	1.36	0.128	3.37	0.386	3.78	0.514	4.45	0.570	3.24	0.377
KSM	1.5	1.34	0.124	3.31	0.382	3.88	0.514	4.58	0.582	3.28	0.373
	2	1.28	0.122	3.33	0.396	3.92	0.534	4.70	0.578	3.30	0.383

Table 4: Average running time in seconds over the entire test set for KSM initialized PR codes. Detailed results are given for the various permutation types. AS denotes *arc scans* and DP denote *double pushes* in push relabel algorithms. For other algorithms, AS and the sum of other operations is given.

Algorithm	RF	No perm		Row perm		Col perm		Row + Col perm		Average	
		AS	DP	AS	DP	AS	DP	AS	DP	AS	DP
PR	1	27.30	1.486	123.02	2.529	150.84	4.366	116.84	1.875	104.33	2.561
KSM	1.5	24.96	1.327	77.80	2.411	118.90	3.687	95.88	1.962	79.31	2.345
	2	22.45	0.982	56.64	1.998	99.05	3.004	49.77	1.782	56.88	1.939
PR-Fair	1	24.06	1.344	85.75	2.559	113.86	2.688	93.31	2.044	79.13	2.157
KSM	1.5	23.33	1.178	58.03	2.407	96.91	2.544	62.56	2.022	60.12	2.036
	2	21.49	0.898	54.23	2.038	61.09	2.213	40.52	1.783	44.27	1.732
PR-Fair-Spread	1	22.81	1.297	44.94	2.682	48.40	2.546	32.07	2.026	37.03	2.137
KSM	1.5	22.44	1.166	42.65	2.343	49.13	2.556	31.86	1.969	36.50	2.007
	2	20.56	0.865	47.22	2.004	51.29	2.186	33.74	1.701	38.17	1.688

Table 5: Ratio between average and median running time for different algorithms. In general, PR shows higher ratios than all other algorithms. Average and median over these ratios is approximately 10.

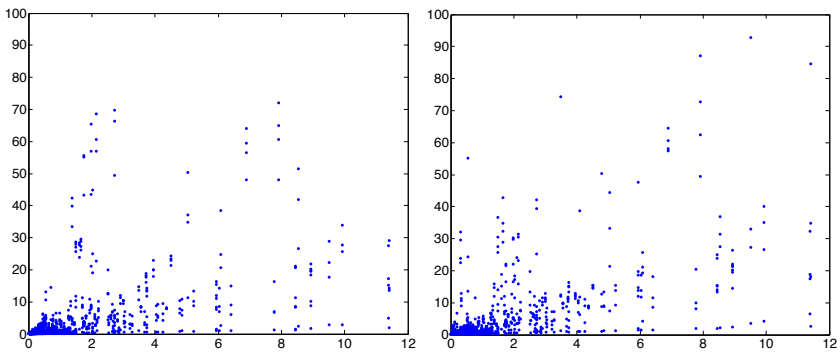
		No perm Avg/Med	Row perm Avg/Med	Col perm Avg/Med	Row+Col perm Avg/Med	Total Avg/Med
PR	1	11.04	10.79	19.91	8.45	14.95
SGM	1.5	10.95	11.50	19.98	8.78	15.25
	2	10.57	9.98	19.24	9.03	15.13
	4	10.42	8.10	8.15	7.52	9.36
	8	10.77	8.43	9.18	8.44	10.08
PR-Fair	1	11.51	15.20	11.66	11.01	13.76
SGM	1.5	10.67	16.35	11.82	11.16	13.99
	2	11.14	11.16	8.56	11.94	11.59
	4	10.70	8.59	9.59	7.93	10.02
	8	10.53	8.04	7.35	7.83	9.12
PRFair	1	11.18	9.00	8.43	8.31	9.74
-Spread	1.5	10.51	9.27	8.66	8.54	9.84
SGM	2	10.27	9.07	8.60	8.79	9.97
	4	10.91	8.29	7.98	8.23	9.52
	8	9.38	7.92	7.54	7.80	9.00
Other algs.						
PFP		9.42	7.52	7.89	6.78	8.01
HI_WAVE		7.31	6.48	6.66	6.87	6.85
LO_LIFO		8.87	8.28	7.58	8.24	8.15
LO_FREE		6.15	5.32	6.81	5.20	5.99
Average		10.12	9.44	10.29	8.47	10.54
Median		10.57	8.59	8.56	8.31	9.84

Table 6: Running times on matrices with large discrepancies between running times of the different algorithms. Running times are given in seconds. The full name of *DIMACS10/channel* is *channel-500x100x100-b050*.

	Alg.	PFP	PR-Fair RF=2	PR-Fair RF=2	PR RF=4	PR-Spread RF=1	HI_WAVE	LO_LIFO	LO_FREE
	Init.	KSM	+KSM	+SGM	+SGM	+SGM			
Group/Matrix	Perm.								
AMD/G3_circuit	NONE	0.27	0.16	0.16	0.16	0.17	2.86	2.88	16.66
GHS_indef/boyd2	COL	0.36	0.21	0.29	0.30	0.30	1.17	22.74	0.65
Freescale/transient	NONE	0.04	0.02	0.02	0.02	0.02	2.58	0.69	0.40
DIMACS10/channel	NONE	2.18	1.41	2.43	2.38	2.47	13.69	13.27	216.42
CEMW/t2em	NONE	0.12	0.07	0.07	0.07	0.07	13.62	21.51	1.26
Freescale/circuit5M	NONE	2.0	2.0	1.5	1.5	1.5	102.1	13.0	13.3
	COL	13.8	43.9	33.7	160.8	11.1	26.0	24.7	22.8
	ROW	12.0	1908.7	315.4	20.8	12.1	45.1	20.6	20.1
	ROWCOL	19.9	1604.2	522.0	32.3	16.3	62.1	31.2	35.4
Rajat/rajat29	ROW	0.93	12.93	13.89	1.95	0.78	356.97	455.51	1.48
	COL	0.97	9.77	12.02	1.16	0.74	597.13	550.25	1.72
	ROWCOL	1.67	11.23	10.70	3.03	0.79	780.51	875.90	1.91
	NONE	0.18	0.13	0.13	0.12	0.12	1029.31	1018.17	1.04

Table 7: Normalized running time of `PR-Fair` for random graphs generated by MATLAB *sprand* for different relabeling frequencies. D denotes the average degree. Optimum values are denoted in **boldface**.

Freq.	1	2	4	8	16	32
D=3						
18	1.32	1.03	**0.79**	0.87	0.87	1.12
19	1.31	1.06	0.85	0.89	**0.82**	1.07
20	1.58	1.00	0.87	0.83	**0.79**	0.93
21	1.20	1.02	1.01	0.86	**0.84**	1.08
22	1.41	1.01	**0.82**	0.83	0.87	1.06
23	1.35	0.98	**0.80**	0.87	0.94	1.06
D=5						
18	1.13	0.89	**0.84**	0.98	0.94	1.22
19	1.27	0.92	**0.77**	0.78	0.97	1.28
20	1.30	0.90	0.98	**0.80**	0.92	1.09
21	1.14	0.93	**0.84**	1.01	0.94	1.14
22	1.41	0.94	0.99	**0.78**	0.88	1.00
23	1.13	0.93	**0.79**	1.03	0.94	1.18
D=7						
18	1.02	1.02	0.88	**0.84**	1.12	1.12
19	0.87	1.12	0.94	**0.83**	1.15	1.10
20	1.07	1.04	0.89	**0.79**	1.12	1.08
21	1.48	1.08	0.83	**0.68**	0.88	1.04
22	1.11	1.07	0.88	**0.80**	1.09	1.06
23	1.15	0.99	0.82	**0.79**	1.00	1.25



(a) `PRFair-Spread` (b) `PFP`

Figure 4: Scatter-plot showing running time in relation to instance size for `PR-Fair-Spread` and `PFP`. The number of edges τ is denoted on the x-axis in multiple of 10 million, while the y-axis shows running time in seconds.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399