Maximum Matching in General Graphs Without Explicit Consideration of Blossoms

Norbert Blum
Informatik IV, Universität Bonn
Römerstr. 164, D-53117 Bonn, Germany
email: blum@cs.uni-bonn.de

October 26, 1999

Abstract

We reduce the problem of finding an augmenting path in a general graph to a reachability problem in a directed, bipartite graph. Furthermore, we show that a slight modification of depth-first search leads to an algorithm for finding such paths. This new point of view enables us to develop algorithms for the solution of matching problems without explicit analysis of blossoms, nested blossoms, a.s.o. A variant of Edmonds' primal-dual method for the weighted matching problem which uses the modified depth-first search instead of Edmonds' maximum matching algorithm as a subroutine is described. Furthermore, a straightforward $O(nm \log n)$-implementation of this algorithm is given.

1 Introduction and motivation

Since Berge's theorem in 1957 [4] it has been well known that for constructing a maximum matching, it suffices to search for augmenting paths. But until 1965, only exponential algorithms for finding a maximum cardinality matching in nonbipartite graphs were known. The reason was that one did not know how to treat odd cycles, the socalled "blossoms" in alternating paths. In his pioneering work, Edmonds [7] solved this problem by shrinking these odd cycles. In [2, 10, 16, 19], it is shown how to avoid explicit shrinking of odd cycles. All these algorithms need $O(n^3)$ or $O(nm)$ time, where n is the number of nodes, and m is the number of edges in the graph.

The first polynomial algorithm for the weighted matching problem also depends on Edmonds [8]. Its run time is $O(n^4)$. Gabow [9] and Lawler [16] have develloped $O(n^3)$ implementations of Edmonds algorithm. Ball and Derigs [3] gave an $O(nm \log n)$ implementation. The best implementation of Edmonds algorithm uses $O(n(m + n \log n))$ time and is also given by Gabow [11]. For a more detailed description of the known weighted matching algorithms see [14, 16, 17].

Our goal is to avoid sophisticated explicit analysis of (nested) blossoms. For getting this, we reduce the problem of finding an augmenting path to a reachability problem in a directed, bipartite graph. It is shown, how to solve this reachability problem by a modified depth-first search. The algorithm obtained is not fundamentally different from previous algorithms which use Edmonds' traditional terminology of blossoms. But we believe that this new point of view, which avoids the explicit consideration of blossoms, simplifies the situation considerably. We have described a simplified realization of the Hopcroft-Karp approach [15] for the computation of a maximum cardinality matching in general graphs in [6]. Furthermore, we show how to use the modified depth-first search algorithm in the primal step of Edmonds' maximum weighted matching algorithm. This approach allow to implement Edmonds' algorithm without shrinking and expanding the socalled "blossoms". A straightforward $O(nm \log n)$ implementation will be described, too.

In Section 2, definitions and the general method are given. We will describe the reduction to a reachability problem in a directed, bipartite graph in Section 3. This reachability problem is solved in Section 4. We will prove the correctness in Section 5. In Section 6, we will present an efficient imple-

mentation of the solution. In Section 7, we show how to use the modified depth first search as subroutine in Edmonds' maximum weighted matching algorithm. The implementation of this approach will be given in Section 8.

2 Definitions and the general method

A *graph* $G = (V, E)$ consists of a finite, nonempty set of *nodes* V and a set of *edges* E. G is either *directed* or *undirected*. In the (un-)directed case, each edge is an (un-)ordered pair of distinct nodes. A graph $G = (V, E)$ is *bipartite* if V can be partitioned into disjoint nonempty sets A and B such that for all $(u, v) \in E$, $u \in A$ and $v \in B$, or vice versa. Then we often write $G = (A, B, E)$. A *path* P from $v \in V$ to $w \in V$ is a sequence of nodes $v = v_0, v_1, \ldots, v_k = w$, which satisfies $(v_i, v_{i+1}) \in E$, for $0 \leq i < k$. The *length* $|P|$ of P is the number k of edges on P. P is *simple* if $v_i \neq v_j$, for $0 \leq i < j \leq k$. For conveniences, P will denote the path $v_0, v_1, \ldots, v_k$, the set of nodes $\{v_0, v_1, \ldots, v_k\}$, and the set of edges $\{(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\}$. If there exists a path from v to w (of length 1) v is called a *(direct) predecessor* of w, and w is called a *(direct) successor* of v.

Let $G = (V, E)$ be an undirected graph. $M \subseteq E$ is a *matching* of G if no two edges in M have a common node. A matching M is *maximal* if there exists no $e \in E \setminus M$ such that $M \cup \{e\}$ is a matching. A matching M is *maximum* if there exists no matching $M' \subseteq E$ of larger size. Given an undirected graph $G = (V, E)$, the *maximum matching problem* is finding a maximum matching $M \subseteq E$. A path $P = v_0, v_1, \ldots, v_k$ is *M-alternating*, if it contains alternately edges in M and in $E \setminus M$. A node $v \in V$ is *M-free* if v is not incident to an edge in M. Let $P = v_0, v_1, \ldots, v_k$ be a simple M-alternating path. P is *M-augmenting* if v_0 and v_k are M-free. Let P be an M-augmenting path in G. Then $M \oplus P$ denotes the *symmetric difference* of M and P; i.e. $M \oplus P = M \setminus P \cup P \setminus M$. It is easy to see that $M \oplus P$ is a matching of G, and $|M \oplus P| = |M| + 1$.

The key to most algorithms for finding a maximum matching in a graph is the following theorem of Berge [4].

Theorem 1 *Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ be a matching. Then M is maximum if and only if there exists no M-augmenting path in G.*

Berge's theorem directly implies the following general method for finding a maximum matching in a graph G.

Algorithm 1

Input: An undirected graph $G = (V, E)$, and
a matching $M \subseteq E$ (possibly $M = \emptyset$)

Output: A maximum matching M_{max}
Method:

while there exists an M-augmenting path
do
construct such a path P;
$M := M \oplus P$
od;
$M_{max} := M$.

The key problem is now this: How to find an M-augmenting path P, if such a path exists? We solve this key problem in the following way.

1. We reduce the key problem to a reachability problem in a directed, bipartite graph $G_M = (V', E_M)$.

2. We solve this reachability problem constructively.

3 Reduction to a reachability problem

In the bipartite case, we construct from $G = (A, B, E)$ and a matching $M \subseteq E$ a directed graph $G_M = (V', E_M)$ by directing the edges in M from A to B, and directing the edges in $E \setminus M$ from B to A. Additionally, we add two new nodes s and t to $A \cup B$, add for each M-free node $b \in B$ the edge (s, b) to E_M, and add for each M-free node $a \in A$ the edge (a, t) to E_M. It is easy to prove that there is an M-augmenting path in G if and only if there is a simple path from s to t in G_M. This reachability problem can be solved by performing a depth-first search (DFS) of G_M with start node s.

Now we will consider the general case. Let $G = (V, E)$ be an undirected graph, and $M \subseteq E$ be a matching. Let $V_M = \{x \in V \mid x \text{ is } M\text{-free}\}$. For the definition of G_M we have the following difficulty.

A priori, we cannot divide the set of nodes V into two sets A and B such that an M-augmenting path exists in G if and only if there exists an M-augmenting path, using alternately nodes from A and from B. Hence, for defining G_M we introduce for each node $v \in V$ two nodes $[v, A]$ and $[v, B]$ such that an analogous construction of a graph G_M is possible. Both edges $([v, A], [w, B])$ and $([w, A], [v, B])$ are in G_M if and only if $(v, w) \in M$. Both edges $([x, B], [y, A])$ and $([y, B], [x, A])$ are in G_M if and only if $(x, y) \in E \setminus M$. Additionally, we add for each M-free node $v \in V$ the edges $(s, [v, B])$ and $([v, A], t)$ to G_M, where s and t are two new nodes. More formally, let $G_M = (V', E_M)$ where

$$\begin{aligned} V' &= \{[v, A], [v, B] \mid v \in V\} \cup \{s, t\} \qquad s, t \notin V,\ s \neq t \\ E_M &= \{([v, A], [w, B]), ([w, A], [v, B]) \mid (v, w) \in M\} \\ &\quad \cup \{[x, B], [y, A]), ([y, B], [x, A]) \mid (x, y) \in E \setminus M\} \\ &\quad \cup \{(s, [v, B]), ([v, A], t) \mid v \in V_M\}. \end{aligned}$$

Analogously to the bipartite case, we have directed the edges in M "from A to B" and the edges in $E \setminus M$ "from B to A". Since the distinct nodes $[v, A]$ and $[v, B]$ in V' correspond to the same node v in V, it does not suffice to construct a simple path from s to t in G_M for finding an M-augmenting path in G. Hence, we define strongly simple paths in G_M which cannot contain both nodes $[v, A]$ and $[v, B]$, for all $v \in V$. A path P in G_M is *strongly simple* if

a) P is simple, and
b) $\forall [v, A] \in V' : [v, A] \in P \Rightarrow [v, B] \notin P$.

Now we can formulate the reachability problem in G_M which is equivalent to the problem of finding an M-augmenting path in G.

Theorem 2 *Let $G = (V, E)$ be an undirected graph, $M \subseteq E$ be a matching, and $G_M = (V', E_M)$ be defined as above. Then there exists an M-augmenting path in G if and only if there exists a strongly simple path from s to t in G_M.*

Proof: "$\Leftarrow$": Let $P = s, [v_1, B], [v_2, A], [v_3, B], \ldots, [v_{k-1}, B], [v_k, A], t$ be a strongly simple path in G_M. Then $v_i \neq v_j$, $1 \leq i < j \leq k$, and $v_1, v_k \in V_M$. Hence, $P' = v_1, v_2, \ldots, v_k$ is an M-augmenting path in G.

"⇒": Let $Q = w_1, w_2, \ldots, w_{l-1}, w_l$ be an M-augmenting path in G. Then $w_i \neq w_j$, $1 \leq i < j \leq l$, and $w_1, w_l \in V_M$. Hence, by the construction of G_M, $Q' = s, [w_1, B], [w_2, A], \ldots, [w_{l-1}, B], [w_l, A], t$ is a strongly simple path in G_M. ∎

4 The solution of the reachability problem

Depth-first search (DFS) finds simple paths in a directed graph. Hence, we cannot use DFS directly for the solution of the reachability problem in G_M. We will modify the usual DFS such that the modified depth-first search (MDFS) finds precisely the strongly simple paths in G_M. Let $[v, \overline{A}] = [v, B]$, and $[v, \overline{B}] = [v, A]$. Remember that a DFS partitions the edges of the graph into four categories [1]. Similarly, the edges of G_M are partitioned into five categories by a MDFS of G_M:

1. *Tree edges*, which are edges leading to new nodes $[v, X]$, $X \in \{A, B\}$, for which $[v, \overline{X}]$ is not a predecessor during the search.

2. *Weak back edges*, which are edges leading to new nodes $[v, A]$, for which $[v, B]$ is a predecessor during the search.

3. *Back edges*, which go from descendants to ancestors during the search.

4. *Forward edges*, which go from ancestors to proper descendants but are not tree edges.

5. *Cross edges*, which go between nodes that are neither ancestors nor descendants of one another during the search.

Like DFS, MDFS uses a stack K for the organization of the search. Analogously to DFS, the MDFS-stack K defines a tree, the *MDFS-tree* T. Before describing MDFS in detail, we will describe the algorithm informally. TOP(K) denotes the last node added to the MDFS-stack K. In each step, MDFS considers an edge (TOP(K), $[w, Y]$) which was not considered previously. Let $e = ([v, X], [w, \overline{X}])$ be the edge under consideration. We distinguish two cases:

1. $X = A$, i.e. $(v, w) \in M$. *tree edge*

2. $X = B$, i.e. $(v, w) \in E \setminus M$

2.1 $[w, A] \in K$ — *back edge*

2.2 $[w, A] \notin K$ but $[w, B] \in K$

i) $[w, A]$ has been in K previously — *cross edge*

ii) $[w, A]$ has not been in K previously — *weak back edge*

2.3 $[w, A] \notin K$ and $[w, B] \notin K$

i) $[w, A]$ has been in K previously — *forward* or *cross edge*

ii) $[w, A]$ has not been in K previously — *tree edge*

MDFS differs from DFS only in Cases 2.2.ii and 2.3.i. Next, we will discuss both of these cases.

Case 2.2.ii: Since $[w, A]$ has not been in K before, DFS would perform the operation PUSH($[w, A]$). Since $[w, B] \in K$, and MDFS should only construct strongly simple paths in G_M, MDFS does not perform the operation PUSH($[w, A]$).

Case 2.3.i: Since $[w, A]$ has been in K before, DFS would perform no PUSH-operation. But the different treatment of Case 2.2.ii can cause the following situation. MDFS has found a path from $[w, A]$ to a node $[u, A]$. But the node $[u, B]$ was in K, and hence by Case 2.2.ii, the operation PUSH($[u, A]$) has not been performed. But now, $[u, B] \notin K$. As we will prove later, the paths P from s to $[v, B]$ and Q from $[w, A]$ to $[u, A]$ are *strongly disjoint*; i.e. there is no $[r, X] \in P$, $X \in \{A, B\}$ such that $\{[r, A], [r, B]\} \cap Q \neq \emptyset$. Hence, the path P,Q is strongly simple. Since MDFS has found a strongly simple path from s to $[u, A]$, MDFS now performs the operation PUSH($[u, A]$).

Note that with respect to depth-first search, the DFS-stack contains exactly the current search path. With respect to the modified depth-first search, the situation is different. In Case 2.3.i, the node $[u, A]$ is pushed. But to obtain a current search path, between the nodes $[v, B]$ and $[u, A]$, we have to insert any path $[w, A], Q$ such that a path

$$P = s, \ldots, [u, B], [v, B], [w, A], Q, [u, A]$$

with $([x, B], [u, A]) \in E_M$ where $Q = Q', [x, B]$ is constructed by the algorithm. Since we do not want to forgot the information about the first node

on the path which we have to add between the nodes $[v, B]$ and $[u, A]$, we create the artificial tree edge $([v, B], [u, A])_{[w,A]}$. Such an edge is called *extensible edge*. It is possible that there exists various such paths Q. Hence, after the performance of PUSH($[u, A]$), the number of corresponding current search path can increase.

Always if we consider *one current search path* we mean that we can take an arbitrary corresponding current search path. If we add to the constructed MDFS-tree T all forward and all cross edges and replace every extensible edge $([v, B], [u, A])_{[w,A]}$ by all possible paths $[w, A], Q$, then we obtain the *expanded MDFS-tree* T_{exp}.

We say that MDFS has *constructed* a strongly simple path P if T_{exp} contains P. We say that MDFS has *found* a strongly simple path $P', [v, B], [w, A]$ if the path $P', [v, B]$ is constructed by MDFS and the edge $([v, B], [w, A])$ is a considered weak back edge.

Next we shall describe MDFS more in detail. We have to solve the following problem: How to find node $[u, A]$ in Case 2.3.i? For the solution of this problem, we assume that MDFS is organized such that for all nodes $[w, A] \in V'$, the following holds true:

After performing the operation POP($[w, A]$), MDFS has always computed a set $L_{[w,A]}$ of nodes such that $L_{[w,A]}$ contains exactly those nodes $[u, A] \in V'$ satisfying the requirements that

1. MDFS has found a path $P = [w, A], Q, [u, A]$ with $[u, B] \notin Q$,
2. PUSH($[u, A]$) has never been performed, and
3. POP($[u, B]$) has been performed.

Before the performance of POP($[w, A]$), we fix $L_{[w,A]} = \emptyset$.

In the description of MDFS we assume for all $[w, A] \in V'$ that $L_{[w,A]}$ is computed correctly. As we will prove later, always $|L_{[w,A]}| \leq 1$. The computation of $L_{[w,A]}$, as well as an efficient implementation of MDFS, can be found in Section 6. For $v \in V'$, $N[v]$ denotes the adjacency list of v.

Algorithm 2 (MDFS)

Input: $G_M = (V', E_M)$
Output: A strongly simple path P from s to t, if such a path exists.
Method:

```
PUSH(s);
while K ≠ ∅ and no path from s to t is found
    do
        SEARCH
    od.
```

SEARCH is a call of the following procedure.

```
            procedure SEARCH;
            if TOP(K) = t then
                reconstruct a strongly simple path P from s to t
                which has been constructed by the algorithm
            else
                mark TOP(K) "pushed";
                for all nodes [w, Y] ∈ N[TOP(K)]
                do
(Case 1)            if Y = B then
                        PUSH([w, B]);
                        SEARCH
(Case 2)            else
(Case 2.1)              if [w, A] ∈ K then
                            no PUSH-operation is performed
                        else
(Case 2.2)                  if [w, B] ∈ K then
                                no PUSH-operation is performed
(Case 2.3)                  else
(Case 2.3.i)                    if [w, A] is marked "pushed" then
                                    while L_[w,A] ≠ ∅
                                    do
                                        choose any [u, A] ∈ L_[w,A];
                                        PUSH([u, A]);
                                        SEARCH
                                    od
(Case 2.3.ii)                   else
                                    PUSH([w, A]);
                                    SEARCH
                                fi
```

fi
fi
fi
od
POP
fi.

5 The correctness proof of MDFS

The correctness proof of MDFS is inspired by the correctness proof of DFS. First we will prove some lemmas. The first lemma implies that the first PUSH-operation which destroys the property "strongly simple" must push a node with second component A.

Lemma 1 *As long as MDFS construct only strongly simple paths, the following holds true: After the operation PUSH($[v, A]$) where v is not M-free, the operation PUSH($[w, B]$) where $([v, A], [w, B]) \in E_M$ always follows without destroying the property "strongly simple".*

Proof: After the performance of the operation PUSH($[v, A]$), MDFS always consider the unique edge $([v, A], [w, B]) \in E_M$ and performs the operation PUSH($[w, B]$). If this operation destroys the property "strongly simple", then $[w, A]$ and hence, $[v, B]$ would be on a current search path. But then the operation PUSH($[v, A]$) would have destroyed the property "strongly simple", a contradiction. ∎

The next lemma shows that MDFS constructs a path from s to a node $[x, A]$ if in a specific situation a strongly simple path from s to this node exists.

Lemma 2 *Let $[u, B] \in V'$ be a node for which MDFS performs the operation PUSH($[u, B]$). Furthermore, at the moment when POP($[u, B]$) is performed by MDFS, only strongly simple paths have been constructed by MDFS. Let $[x, A] \in V'$ such that at the moment when PUSH($[u, B]$) is performed, there is a strongly simple path $P = [u, B], [v, A], Q, [x, A]$ with $[z, X], [z, \overline{X}] \notin K$, for all $[z, X] \in P$. Then PUSH($[x, A]$) has been performed before POP($[u, B]$).*

Remark: Lemma 2 implies that either PUSH($[x, A]$) and POP($[x, A]$) have been performed before the performance of PUSH($[u, B]$), or both operations have been performed between the operations PUSH($[u, B]$) and POP($[u, B]$).

Proof: Let $P = [u, B], [v, A], [v', B], Q', [x, A]$ be such a path of shortest length for which PUSH($[x, A]$) has *not* been performed before POP($[w, B]$). It is clear that edge $e = ([u, B], [v, A])$ has been considered before the performance of POP($[u, B]$). If the operation PUSH($[v, A]$) is performed according to this consideration of edge e, then by the assumption that P is a shortest path such that the assertion is not fulfilled, PUSH($[x, A]$) has been performed before POP($[v', B]$), and hence, before POP($[u, B]$). Hence, MDFS is in Case 2.3.i and performs the corresponding while-statement. Consider the moment when MDFS finishes this while-statement, i.e. $L_{[v,A]} = \emptyset$.

Let $[z, A] \in P$ be the first node on P for which PUSH($[z, A]$) has not been performed. Since $[x, A]$ has this property, node $[z, A]$ exists. Let $P = [u, B], [v, A], Q_1, [z, A], Q_2, [x, A]$. Then

1. MDFS has found the path $[v, A], Q_1, [z, A]$;
2. PUSH($[z, A]$) has never been performed; and
3. POP($[z, B]$) has been performed (since $[z, B] \notin K$ when PUSH($[u, B]$) is performed).

Hence, $[z, A] \in L_{[v,A]}$, and hence, $L_{[v,A]} \neq \emptyset$. But this contradicts $L_{[v,A]} = \emptyset$. ∎

For $w \in V'$, we denote

$$r(w) = \begin{cases} [v, \overline{X}] & \text{if } w = [v, X] \\ t & \text{if } w = s \\ s & \text{if } w = t \end{cases}$$

Let $S = w_1, w_2, \ldots, w_k$ be a path in G_M. The *backpath* $r(S)$ of S is defined by

$$r(S) = r(w_k), r(w_{k-1}), \ldots, r(w_1).$$

Lemma 3 *Let $[u, B] \in V'$ be a node for which MDFS performs the operation PUSH($[u, B]$). Furthermore, at the moment when POP($[u, B]$) is performed by MDFS, only strongly simple paths have been constructed by MDFS. If there*

exists a strongly simple path $P = [v, A], Q, [w, B]$ *such that at the moment when PUSH*$([u, B])$ *is performed,* $[z, X], [z, \overline{X}] \notin K$, *for all* $[z, X] \in P$, *and* $([u, B], [v, A]), ([w, B], [u, A]) \in E_M$, *then for all* $[z, X] \in P$, *the operations PUSH*$([z, X])$ *and PUSH*$([z, \overline{X}])$ *have been performed before the operation POP*$([u, B])$.

Proof: For the nodes $[z, X] \in P$ consider the path $[u, B], P$ and apply Lemma 2. For the nodes $[z, \overline{X}]$ consider the path $[u, B], r(P)$ and apply Lemma 2. ∎

By the definition of $L_{[u,A]}$ and Lemma 3, we obtain for all $[u, A] \in V'$: $|L_{[u,A]}| > 0$ implies that PUSH$([u, A])$ and POP$([u, A])$ have been performed.

Lemma 4 *MDFS maintains the following invariants:*

Invariant 1: MDFS constructs only strongly simple paths.

Invariant 2: $|L_{[w,A]}| \leq 1$, *for all* $[w, A] \in V'$.

Invariant 3: Assume that the algorithm performs the assignment $L_{[w,A]} := [u, A]$. *Then after the performance of PUSH*$([u, A])$, *always* $L_{[w,A]} = L_{[u,A]}$.

Remark: Invariant 2 and Invariant 3 are not needed for the correctness proof of MDFS. But we will need these invariants for the efficient implementation of the algorithm. Moreover, the proof of Invariant 1 is easier if we prove all invariants simultaneously.

Proof: Consider the first situation in which one of the three invariants is not maintained. Three cases are to be considered.

Case 1: Invariant 1 is not maintained.

Only a PUSH-operation can destroy the property "strongly simple". Note that a PUSH-operation cannot affect Invariant 2 or Invariant 3.

Lemma 1 implies that this PUSH-operation occurs during the consideration of an edge $e = ([v, B], [w, A])$. Then e corresponds to edge $(v, w) \in E \setminus M$.

If $[w, A]$ is not marked "pushed", then Case 2.3.ii of MDFS applies, and PUSH$([w, A])$ is performed. The only possible situation in which this PUSH-operation destroys the property "strongly simple" is the following:

On a current search path there is a subpath Q which is caused by an application of Case 2.3.i of MDFS such that $[w, B] \in Q$.

Hence, there exists $[u, A] \in V'$ such that the addition of Q to this current search path is caused by the operation PUSH($[u, A]$). By construction, the assumptions of Lemma 3 are fulfilled with respect to $[u, B], [w, B] \in P$. Hence, by Lemma 3, PUSH($[w, A]$) has been performed *before* POP($[u, B]$), and hence, *before* PUSH($[u, A]$), a contradiction.

Hence, $[w, A]$ is marked "pushed". Therefore, Case 2.3.i of MDFS applies, and for node $[u, A] = L_{[w,A]}$, the operation PUSH($[u, A]$) is performed. (Note that by Invariant 2, $|L_{[w,A]}| \leq 1$. We thus write $L_{[w,A]} = [u, A]$ instead of $L_{[w,A]} = \{[u, A]\}$.) Hence, the algorithm extends the current search paths by a path $[w, A], Q, [u, A]$. Note that only $[u, A]$ will be pushed. Later, the subpath $[w, A], Q$ must be reconstructed if needed. By the definition of $L_{[w,A]}$, and by Lemma 3, the operations PUSH($[z, X]$), POP($[z, X]$), PUSH($[z, \overline{X}]$), and POP($[z, \overline{X}]$) have been performed, for all $[z, X] \in Q$. Hence, the only possible situation in which PUSH($[u, A]$) destroys the property "strongly simple" is the following:

There is a node $[p, X] \in [w, A], Q, [u, A]$, and a subpath Q' of a current search path which is caused by an application of Case 2.3.i such that $[p, X] \in Q'$ or $[p, \overline{X}] \in Q'$. Since one end node of an edge in the current matching uniquely determines the other end node, we can choose $[p, X]$ such that $[p, A] \in Q'$.

Consider node $[u', A] \in K$ with PUSH($[u', A]$) is the operation which adds the subpath Q' to this current search path. By the definition of $L_{[p,A]}$, Lemma 3 and Invariant 2, before the performance of PUSH($[u', A]$), $L_{[p,A]} = [u', A]$. Hence, by Invariant 3, after the performance of PUSH($[u', A]$), always $L_{[p,A]} = L_{[u',A]}$. By the choice of $[p, A]$, $L_{[p,A]} = [u, A]$, and hence, $L_{[u',A]} = [u, A]$ in the situation under consideration. Hence, POP($[u', A]$) is performed. Hence, $[u', A] \notin K$, a contradiction.

Case 2: Invariant 2 is not maintained.

Then there is $[w, A], [p_1, A], [p_2, A] \in V'$ with the property that $L_{[w,A]} = \{[p_1, A]\}$ before the performance of POP($[p_2, B]$), and $L_{[w,A]} = \{[p_1, A], [p_2, A]\}$ after the performance of POP($[p_2, B]$). Hence, MDFS has found a path $P_1 = [p_1, B], Q, [p_1, A]$ with $[w, A] \in Q$ and found a path $P_2 = [p_2, B], Q', [p_2, A]$ with $[w, A] \in Q'$.

If MDFS has found the path P_2 after the performance of POP($[p_1, B]$), then $[w, A]$ can only be added to Q' in the following way:

An operation PUSH($[u, A]$), caused by an application of Case 2.3.i with respect to a node $[v, A]$ (i.e., $[u, A] \in L_{[v,A]}$) is performed such that the current search path is extended by a path $[v, A], \tilde{Q}, [u, A]$ with $[w, A] \in \tilde{Q}$. But then, $[u, A] \in L_{[w,A]}$ before the performance of PUSH($[u, A]$). PUSH($[u, A]$) is performed after POP($[p_1, B]$). Hence, $[u, A], [p_1, A] \in L_{[w,A]}$ between the performance of these two operations. This contradicts the assumption that we consider the situation in which Invariant 2 is not maintained *for the first time*.

Hence, MDFS has found the path P_2 before the performance of POP($[p_1, B]$). Note that $[p_1, B] \notin Q'$. Otherwise, by Lemma 3, PUSH($[p_1, A]$) is performed before POP($[p_2, B]$), and hence, $[p_1, A] \notin L_{[w,A]}$ after POP($[p_2, B]$). Let $[r, A]$ be the first node on Q' such that $[r, A] \in Q$, or $[r, B] \in Q$. Since node $[w, A]$ has this property, node $[r, A]$ exists. Let

$$Q' = Q'_1, [r, A], Q'_2 \text{ and } Q = \begin{cases} Q_1, [r, A], Q_2 & \text{if } [r, A] \in Q \\ Q_1, [r, B], Q_2 & \text{otherwise} \end{cases}$$

Consider the path

$$R = \begin{cases} Q'_1, [r, A], Q_2, [p_1, A] & \text{if } [r, A] \in Q \\ Q'_1, [r, A], r(Q_1), [p_1, A] & \text{otherwise} \end{cases}$$

Then Lemma 2 applies with respect to $[p_2, B]$, $[p_1, A]$, and the strongly simple path R. Hence, PUSH($[p_1, A]$) is performed before POP($[p_2, B]$), and hence, $[p_1, A] \notin L_{[w,A]}$ after POP($[p_2, B]$), a contradiction.

Case 3: Invariant 3 is not maintained.

After the performance of PUSH($[u, A]$), there holds $L_{[w,A]} = L_{[u,A]} = \emptyset$. We will prove that $L_{[w,A]} = L_{[u,A]}$ after the next POP-operation which changes $L_{[w,A]}$ or $L_{[u,A]}$. Then, the assertion follows because of Invariant 2 and the transitivity of the relation $=$.

Let POP($[p, B]$) be the next POP-operation which enlarges $L_{[w,A]}$ or $L_{[u,A]}$. $K_{[w,A]}$ denotes the current MDFS-stack, directly after the performance of PUSH($[w, A]$). Let $K' = K_{[w,A]} \cap K_{[u,A]}$. Note that $[u, B] \in K_{[w,A]} \setminus K'$. According to the location of $[p, B]$ with respect to $K_{[w,A]}$ and to $K_{[u,A]}$, we distinguish three cases.

By construction, $[p,B] \notin K_{[w,A]} \setminus K'$. Otherwise, POP($[p,B]$) would be performed before PUSH($[u,A]$).

Assume that $[p,B] \in K_{[u,A]} \setminus K'$. Let $[q,B]$ be the first node in $K_{[w,A]} \setminus K'$ such that $[q,A] \in K_{[u,A]} \setminus K_{[p,B]}$. Node $[q,B]$ exists since $[u,B]$ has the property that $[u,B] \in K_{[w,A]} \setminus K'$.

Consider the backpath of the path from node $[p,B]$ to node $[q,A]$. This backpath implies that $[q,B]$ and $[p,A]$ fulfill the assumptions of Lemma 2. Hence, PUSH($[p,A]$) occurs before POP($[q,B]$). Since $[q,B] \in K_{[w,A]} \setminus K'$, the operation PUSH($[p,A]$) is also performed before POP($[p,B]$). Hence, POP($[p,B]$) can enlarge neither $L_{[w,A]}$ nor $L_{[u,A]}$.

It remains to consider $[p,B] \in K'$. Let $[q,B] \in K'$ be the node nearest to the top of K' for which PUSH($[q,A]$) has not been performed at the moment when MDFS performs PUSH($[u,A]$). Since $[p,B]$ has this property, $[q,B]$ exists. By consideration of the backpath of the path from $[q,B]$ to $[u,B]$, it is easy to prove that MDFS finds a path from $[u,A]$ to $[q,A]$ not containing $[q,B]$. Hence, $L_{[u,A]} = [q,A]$ after the performance of POP($[q,B]$), and hence, $[q,B] = [p,B]$. Since MDFS has found a path from $[w,A]$ to $[u,A]$ which does not contain $[q,B]$, there holds $L_{[w,A]} = [q,A] = [p,A]$. ∎

Now, the correctness of the algorithm MDFS can easily be derived from Lemma 2 and Lemma 4.

Theorem 3

a) MDFS constructs a path from s to t, if a strongly simple path from s to t exists.

b) MDFS constructs only strongly simple paths.

Proof: a) Let $P = s, [v'_0,B], [v_1,A], [v'_1,B], \ldots, [v'_{r-1},B], [v_r,A], t$ be a strongly simple path from s to t. It is clear that MDFS considers the edge $(s,[v'_0,B])$, and performs the operation PUSH($[v'_0,B]$). (Note that v'_0 is M-free.) Hence, $[v'_0,B]$, $[v_r,A]$ fulfill the assumptions of Lemma 2 with respect to the path $[v'_0,B], [v_1,A], \ldots, [v'_{r-1},B], [v_r,A]$. Hence, by Lemma 2, MDFS performs PUSH($[v_r,A]$), and hence, PUSH(t). Hence, MDFS constructs a path from s to t.

b) is a direct consequence of Invariant 1 of Lemma 4. ∎

6 An implementation of MDFS

Now we will describe how to implement MDFS efficiently. Only two parts of the algorithm are nontrivial to implement.

1. The manipulation of $L_{[w,A]}$, $[w, A] \in V'$.
2. The reconstruction of a strongly simple path P from s to t which is constructed by the algorithm.

For the solution of both subproblems it is useful not to perform the POP-operations explicitly, and to maintain the whole MDFS-tree T. This can be done as follows:

The data structure is a tree T. A pointer TOP always points to TOP(K) in T. The current MDFS-stack K is represented by the unique path from the root s of T to TOP(K) in T. For performing the operation POP, pointer TOP is changed such that it points to the unique direct predecessor in T. When we perform a PUSH-operation, T obtains a new leaf to which TOP points.

Invariant 2 and Invariant 3 are the key for the efficient implementation of our method. Now we will describe the update of $L_{[w,A]}$. By the definition of $L_{[w,A]}$, we have only to change $L_{[w,A]}$ after a PUSH- or after a POP-operation. More exactly, we have to perform:

After PUSH($[u, A]$)**:** $L_{[w,A]} := \emptyset$ if $L_{[w,A]} = [u, A]$.

After POP($[u, B]$)**:** $L_{[w,A]} := [u, A]$ if

1. PUSH($[u, A]$) has never been performed, and
2. MDFS has found a path $P = [w, A], Q, [u, A]$ with $[u, B] \notin Q$.

After the performance of POP($[u, B]$), eventually, MDFS has to find all nodes $[w, A]$ which fulfill Property 2. This can easily be done by any graph search method like depth-first search, starting in node $[u, A]$ and running the considered edges backwards. When the node $[u, B]$ is reached, a backtrack is performed. But with respect to the efficiency, it is useful to investigate the properties of MDFS and to refine the graph search.

First, we will characterize the paths $P = [w, A], Q, [u, A]$ with $[u, B] \notin Q$, found by MDFS. Let $P = e_1, e_2, \ldots, e_t$. Then, the following properties are fulfilled:

1. e_t is a weak back edge.

2. If we start in edge e_t and consider P backwards, then we see some tree edges followed by a single cross, forward or back edge, followed by a sequence of tree edges, and so on.

Hence, we need after the performance of POP($[u, B]$) the following sets of edges:

$$R_{[u,A]} = \{[v, B] \in V' \mid ([v, B], [u, A]) \text{ is a weak back edge}\}$$

and for some $[q, A] \in V'$

$$E_{[q,A]} = \{[v, B] \in V' \mid ([v, B], [q, A]) \text{ is a cross, forward, or back edge}\}.$$

According to Invariant 3, during the backward search some subpaths can be skipped over. Therefore, we need the following set of nodes

$$D_{[q,A]} = \{[p, A] \in V' \mid L_{[p,A]} = [q, A] \text{ previously}\}.$$

By Invariant 3, $D_{[q,A]} \subseteq D_{[q',A]}$ implies $L_{[q,A]} = L_{[q',A]}$. Hence, the knowledge of $L_{[q',A]}$ and the fact $D_{[q,A]} \subseteq D_{[q',A]}$ implies the knowledge of $L_{[q,A]}$.

We say that $D_{[q,A]}$ is *current* if $D_{[q,A]} \not\subseteq D_{[q',A]}$, for all $[q', A] \in V' \setminus \{[q, A]\}$. According to Invariant 3, we can compute $L_{[p,A]}$ in the following way.

1. Compute $[q, A]$ such that $[p, A] \in D_{[q,A]}$, and $D_{[q,A]}$ is current.

2. If $[q, A]$ does not exist, then $L_{[p,A]} = \emptyset$. Otherwise,

$$L_{[p,A]} = \begin{cases} [q, A] & \text{if PUSH}([q, A]) \text{ has never been performed} \\ \emptyset & \text{otherwise} \end{cases}$$

As described above, a correct manipulation of the current sets $D_{[q,A]}$ allows the solution of the first subproblem. Note that every $[p, A] \in V'$ is contained in at most one current set $D_{[q,A]}$.

For the organisation of the backward search, we also need the knowledge if $L_{[p,A]} \neq \emptyset$ previously. This will be realized by the correct update of the following set.

$$L = \{[p, A] \in V' \mid L_{[p,A]} \neq \emptyset \text{ previously}\}.$$

Now we can give a detailed description of the backward search which will be performed after POP($[u, B]$).

The consideration of those paths $P = [w, A], Q, [u, A]$ with $[u, B] \notin Q$ is done in several rounds. In the first round, we construct backwards all paths *without any* cross, forward, or back edge. In the second round, all paths with *exactly one* such edge are constructed implicitly, and so on. In the ith round, we consider the weak back edges $([v, B], [u, A])$ if $i = 1$, and we consider those edges $([v, B], [q, A]) \in E_{[q,A]}$ for which $L_{[q,A]} = [u, A]$ is computed in the $(i-1)$th round, if $i > 1$. Starting in node $[v, B]$, we follow backwards the tree edges as long as node $[u, B]$ is reached. If we reach a node $[p, A] \in L$, then we compute the current $D_{[r,A]}$ such that $[p, A] \in D_{[r,A]}$, and we jump to $[r, A]$ for the continuation of the backward search. According to Invariant 3, $L_{[x,A]} = L_{[r,A]}$ and hence, $L_{[x,A]} = [u, A]$ for all $[x, A] \in D_{[r,A]}$.

For the reconstruction of a strongly simple path from s to t constructed by the algorithm, we store in variable $P_{[r,A]}$ that edge in $E_{[q,A]}$ which concludes that block of tree edges containing the tree edge with end node $[r, A]$, for all $[r, A] \in V'$ with $L_{[r,A]} \neq \emptyset$ for the first time. As soon as $P_{[r,A]}$ is defined, the node $[r, A]$ is inserted into L. Hence, node $[q, A]$ is determined unambiguously by the algorithm.

The implementation of MDFS must be done with attention to the correct manipulation of the sets $D_{[q,A]}, R_{[q,A]}$, and $E_{[q,A]}$. The following table describes in terms of the case of MDFS, and in terms of the operation which is performed, how MDFS has to update these sets.

case, operation	*set updating*
Case 1	no update
Case 2.1	$E_{[w,A]} := E_{[w,A]} \cup \{[v, B]\}$
Case 2.2.i	$E_{[w,A]} := E_{[w,A]} \cup \{[v, B]\}$
Case 2.2.ii	$R_{[w,A]} := R_{[w,A]} \cup \{[v, B]\}$
Case 2.3.i	
$L_{[w,A]} \neq \emptyset$	no update
$L_{[w,A]} = \emptyset$	$E_{[w,A]} := E_{[w,A]} \cup \{[v, B]\}$ if $[w, A] \notin L$
Case 2.3.ii	no update
PUSH($[u, A]$)	no update
POP($[v, B]$)	$D_{[v,A]} := \{[p, A] \mid$ MDFS has found a path from $[p, A]$ to $[v, A]$, not containing $[v, B]\}$

In Case 2.1, it is clear that $[w, A] \notin L$ since POP($[w, A]$) is not performed. In Case 2.2.i, $[w, A] \notin L$ follows directly from $[w, B] \in K$ and Lemma 1. Note that in Case 2.3.i, subcase $L_{[w,A]} \neq \emptyset$, we have to store the information that edge $([v, B], [w, A])$ is used. In the implementation, we accomplish this by adding the edge $([v, B], [w, A])$ to node $[v, B]$ in K. Then we obtain an *expanded node* $\langle([v, B], [w, A]); [v, B]\rangle$. The considerations above lead to the following implementation of the procedure SEARCH.

```
            procedure SEARCH;
            if TOP(K) = t then
                reconstruct a strongly simple path P from s to t
                which has been constructed by the algorithm
            else
                mark TOP(K) "pushed";
                for all nodes [w, Y] ∈ N[TOP(K)]
                do
(Case 1)            if Y = B then
                        PUSH([w, B]);
                        SEARCH
(Case 2)            else
(Case 2.1)              if [w, A] ∈ K then
                            E_[w,A] := E_[w,A] ∪ {TOP(K)}
                        else
(Case 2.2)                  if [w, B] ∈ K then
(Case 2.2.i)                    if [w, A] is marked "pushed" then
                                    E_[w,A] := E_[w,A] ∪ {TOP(K)}
(Case 2.2.ii)                   else
                                    R_[w,A] := R_[w,A] ∪ {TOP(K)}
                                fi
(Case 2.3)                  else
(Case 2.3.i)                    if [w, A] is marked "pushed" then
                                    if L_[w,A] ≠ ∅ then
                                        expand TOP(K) in K to
                                        ⟨(TOP(K), [w, A]); TOP(K)⟩;
                                        PUSH(L_[w,A]); L_[w,A] := ∅;
                                        SEARCH
                                    else
```

```
                                    if [w, A] ∉ L then
                                        E_[w,A] := E_[w,A] ∪ {TOP(K)}
                                    fi
                                fi
(Case 2.3.ii)                   else
                                    PUSH([w, A]);
                                    SEARCH
                                fi
                            fi
                        fi
                    fi
                od;
                (* let TOP(K) = [v, X]*)
                if TOP(K) = [v, B] and [v, A] is not marked "pushed" then
                    L_act := [v, A];
                    D_{L_act} := ∅;
                    L_def := ∅;
                    (* L_def will contain the start nodes for the next round. *)
                    for all [q, B] ∈ R_[v,A]
                    do
                        CONSTRL(([q, B], [v, A]), [v, B]);
                    od;
                    while L_def ≠ ∅
                    do
                        choose any [k, A] ∈ L_def;
                        L_def := L_def \ {[k, A]};
                        for all [q, B] ∈ E_[k,A]
                        do
                            CONSTRL(([q, B], [k, A]), [v, B])
                        od
                    od
                fi;
                POP;
            fi.
```

CONSTRL is a call of the following procedure.

procedure CONSTRL(([q, B], [u, A]), [x, B]);
P_{act} := ([q, B], [u, A]);
[z, B] := [q, B];
while [x, B] is not reached
do
    for all [y, A] on the backpath from [z, B] to $L \cup \{[x, B]\}$
    do
        $D_{L_{act}} := D_{L_{act}} \cup \{[y, A]\}$;
        $L := L \cup \{[y, A]\}$;
        $P_{[y,A]} := P_{act}$;
        $L_{def} := L_{def} \cup \{[y, A]\}$;
    od;
    if the last considered [y, A] $\in L$ **then**
        ($*$ Let $D_{[r,A]}$ be the current set containing [y, A] $*$)
        $D_{L_{act}} := D_{L_{act}} \cup D_{[r,A]}$;
        [z, B] := [r, B]
    fi
od.

The reconstruction of a strongly simple path P from s to t constructed by the algorithm remains to be explained. Beginning at the end of P, such a path P can be reconstructed by traversing the MDFS-tree T backwards. Note that TOP points to the end of P, and that the father of each node in T is always unique. As long as we traverse tree edges of the algorithm MDFS, we have no difficulty. But every time we meet a node [u, A] which was added to P by an application of Case 2.3.i, we have to reconstruct a subpath [w, A], Q, [u, A] which was joined to P. In this situation, the considered portion of T is the expanded node $\langle([v, B], [w, A]); [v, B]\rangle$; i.e., the structure of T tells us that MDFS has applied Case 2.3.i. It remains to reconstruct Q. Note that $P_{[w,A]}$ contains the non-tree edge of MDFS, which finishes the block containing the tree edge with end node [w, A]. Let $P_{[w,A]} = ([v', B], [v'', A])$. Then $P^1_{[w,A]}$ denotes [v', B], and $P^2_{[w,A]}$ denotes [v'', A]. As long as [u, A] is met, we reconstruct Q block by block, beginning at node [w, A]. Each block can be reconstructed as the path P itself. These considerations lead to the following procedure for the reconstruction of an augmenting path, constructed by the algorithm.

procedure RECONSTRPATH(t, s);
$ACTNODE := t$;
while $ACTNODE \neq s$
do
  if $father(ACTNODE)$ is not expanded **then**
    $ACTNODE := father(ACTNODE)$
  else $(*$ let $father(ACTNODE) = \langle([v, B], [w, A]); [v, B]\rangle *)$
    RECONSTRQ$(ACTNODE, [w, A])$;
    $ACTNODE := [v, B]$
  fi
od.

RECONSTRQ is a call of the following procedure.

procedure RECONSTRQ$([u, A], [w, A])$;
$ANF := [w, A]$;
RECONSTRPATH(P^1_{ANF}, ANF);
while $P^2_{ANF} \neq [u, A]$
do
  $ANF := P^2_{ANF}$;
  RECONSTRPATH(P^1_{ANF}, ANF)
od.

Note that the correctness of the manipulation of $L_{[w,A]}$, $[w, A] \in V'$, and the correctness of the reconstruction of the M-augmenting path P follow from Lemma 4, and are straightforward to prove. The procedure RECONSTRPATH resembles standard recursive methods used for the reconstruction of augmenting paths (i.e., see [18]).

The time and space complexity of our implementation of MDFS remain to be considered. It is easy to see that the time used by the algorithm MDFS is bounded by $O(n + m)$ plus the total time needed for the manipulation of the sets $D_{[q,A]}$, $[q, A] \in V'$. If we use linear lists for the realization of the sets $D_{[q,A]}$ with a pointer to the node $[q, A]$ for each element of $D_{[q,A]}$, the execution time for each union operation is bounded by $O(n)$. Following the pointer corresponding to $[p, A]$, we can find the set containing $[p, A]$

in constant time. At most n union operations are performed by MDFS. Hence the total time used for the manipulation of the sets $D_{[q,A]}$ is bounded by $O(n^2)$. The time needed for the n union operations can be reduced to $O(n \log n)$ if we use the following standard trick, the socalled *weighted union heuristic*:

We store with each set the number of elements of the set. A union operation is performed by changing the pointer of the smaller of the two sets which are involved and updating the number of elements. Everytime when the pointer with respect to an element is changed, the size of the set containing this element is at least twice of the size of its previous set. Hence, for each element, its pointer is changed at most $\log n$-times. Hence, the total time used for all union operations is $O(n \log n)$. Altogether, the total time used for the augmentation of one augmenting path is $O(m + n \log n)$.

If we use for the update of the sets $D_{[u,A]}$ disjoint set union [18], the total time can be bounded $O((m + n)\alpha(m, n))$. Note that for each node $[p, A]$ one find operation suffices for the decision of $L_{[p,A]}$. Further, we can reduce these bounds to $O(m + n)$ using incremental tree set union [12]. The space complexity of MDFS is bounded by $O(m + n)$. The considerations above lead to the following theorem.

Theorem 4 *MDFS can be implemented such that it uses only $O(m+n)$ time and $O(m + n)$ space.*

7 Definitions and the primal-dual method

Let $G = (V, E)$ be an undirected graph. If we associate with each edge $(i, j) \in E$ a weight $w_{ij} > 0$ then we obtain a *weighted undirected graph* $G = (V, E, w)$. The weight $w(M)$ of a matching M is the sum of the weights of the edges in M. A matching $M \subseteq E$ has *maximum weight* if $\sum_{(i,j)\in M'} w_{ij} \leq \sum_{(i,j)\in M} w_{ij}$ for all matchings $M' \subseteq E$. Given a weighted undirected graph $G = (V, E, w)$, the *maximum weighted matching problem* is finding a matching $M \subseteq E$ of maximum weight.

First, we will describe the primal-dual method for the computation of a maximum weighted matching. Let $G = (V, E)$ be a weighted undirected graph. Let $\mathcal{F} = \{E_1, E_2, \ldots, E_r\}$, $E_i \subseteq E$ be a family of pairwise distinct

subsets of E. With each node $i \in V$ we associate a *node weight* $\pi(i) \geq 0$. Furthermore, with each edge set $E_l \in \mathcal{F}$, we associate a *set weight* $\mu(E_l) \geq 0$. These new variables are called *dual variables*.

Note that the primal-dual method for bipartite graphs only uses dual variables with respect to the nodes of the graph. This suffices since every node $v \in A \cup B$ has the property that all M-alternating paths from an M-free node in B to v have even length if $v \in B$ and odd length if $v \in A$. But in general graphs, with respect to a node $v \in V$ simultaneously, there can exist M-alternating paths from M-free nodes in B to v of odd and of even length. Moreover, both end nodes of an edge can have even or odd distances from the M-free nodes in B with respect to M-alternating paths. Hence, the dual variables with respect to the edge sets are needed. The exact reasons for this will be clearer during the development of the method.

The values of the dual variables are treated such that the following invariant is always fulfilled:

- For all $(i,j) \in E$ there hold $w(i,j) \leq \pi(i) + \pi(j) + \sum_{(i,j) \in E_l} \mu(E_l)$.

For each edge $(i,j) \in E$, its dual weight $d(i,j)$ is defined by

$$d(i,j) = \pi(i) + \pi(j) + \sum_{(i,j) \in E_l} \mu(E_l).$$

We define the dual weight $d(M)$ of a matching M by

$$d(M) = \sum_{(i,j) \in M} d(i,j).$$

Note that always $w(M) \leq d(M)$ for all matchings $M \subseteq E$.

With respect to an arbitrary matching $M \subset E$, the maximum contribution of the node weight $\pi(i)$ to its dual weight can be $\pi(i)$ since i is adjacent to at most one edge in M. Note that $|E_l \cap M| \leq c(E_l)$ where $c(E_l)$ is the size of a maximum cardinality matching with respect to E_l. Hence, the maximum contribution of the set weight $\mu(E_l)$ can be $c(E_l)\mu(E_l)$. Hence,

$$\sum_{i \in V} \pi(i) + \sum_{E_l \in \mathcal{F}} c(E_l)\mu(E_l)$$

will be always an upper bound for the dual weight of any matching of G. Therefore, with respect to a matching M,

$$w(M) = \sum_{i \in V} \pi(i) + \sum_{E_l \in \mathcal{F}} c(E_l)\mu(E_l)$$

implies that the matching M is a maximum weighted matching.

The question is now, when with respect to a matching M, this equality holds. Since the dual weight of an edge is at least as large as its weight, we obtain the necessary condition $d(i,j) = w(i,j)$ for all edges $(i,j) \in M$. Since all summands in both sums are nonnegative, the node weight $\pi(i)$ has to be 0 for all M-free nodes $i \in V$. Furthermore, for all E_l such that $|M \cap E_l| < c(E_l)$ the set weight $\mu(E_l)$ has to be 0. Altogether, we obtain the following necessary and sufficient conditions:

1. $r(i,j) = d(i,j) - w(i,j) = 0$ for all $(i,j) \in M$,
2. $\pi(i) = 0$ for all M-free nodes $i \in V$, and
3. $\mu(E_l) = 0$ for all $E_l \in \mathcal{F}$ with $|E_l \cap M| < c(E_l)$.

The value $r(i,j)$ is called the *reduced cost* of the edge (i,j).

The primal-dual method for the weighted matching problem can be separated into rounds. The input of every round will be a matching M and values for the dual variables which fulfill the Conditions 1 and 3 with respect to the matching M. Our goal within the round is to modify M and the values of the dual variables such that Conditions 1 and 3 remain valid and the number of nodes violating Condition 2 is strictly decreased.

One round devides into two steps, the *search step* and the *extension step*. The search step try to improve the current matching by finding an augmenting path P such that the number of free nodes with node weight larger than 0 can be decreased by the augmentation of P. If this is not possible then the extension step decreases the values of some dual variables by an appropriate value δ. The extension step can decrease the reduced cost of some edges to 0. Hence, the next search step possibly finds an augmenting path.

During the search step, we will use MDFS. Hence, we define with respect to the current matching M the directed bipartite graph $G_M = (V', E_M, w)$

as follows:

$$\begin{aligned} V' &= \{[v,A],[v,B] \mid v \in V\} \cup \{s,t\} \qquad s,t \notin V,\ s \neq t \\ E_M &= \{([v,A],[w,B]),([w,A],[v,B]) \mid (v,w) \in M\} \\ &\quad \cup \{([x,B],[y,A]),([y,B],[x,A]) \mid (x,y) \in E \setminus M\} \\ &\quad \cup \{(s,[v,B]),([v,A],t) \mid v \in V \text{ is } M\text{-free}\}\,. \end{aligned}$$

Both copies $([i,X],[j,\overline{X}])$ and $([j,\overline{X}],[i,X])$, $X \in \{A,B\}$ obtain weight $w(i,j)$ and reduced cost $r(i,j)$. We arrange that edges with tail s or head t have always reduced cost 0. According to Condition 1, it is only allowed to consider augmenting paths where all edges on these paths have reduced cost 0. Hence, the input graph $G_M^* = (V', E_M^*, w)$ will be the subgraph of G_M containing exactly those edges in E_M having reduced cost 0. I.e., $E_M^* = \{([i,X],[j,\overline{X}]) \in E_M \mid r(i,j) = 0\}$. Note that $M \subseteq E_M^*$.

We start with the empty matching $\emptyset$ and define the graph $G_\emptyset = (V', E_\emptyset, w)$ as described above. Let $W = \max_{(i,j)\in E} w_{ij}$. We initialize all node weights $\pi(i)$ by $W/2$. At the beginning, the family $\mathcal{F}$ of subsets of E will be the empty set such that no set weight has to be defined. In dependence to the algorithm, the needed elements of $\mathcal{F}$ and the corresponding set weights will be defined. As soon as $\mu(E_l)$ becomes zero for an edge set $E_l \in \mathcal{F}$, we will delete E_l from $\mathcal{F}$.

Altogether, we obtain the input graph $G_\emptyset^* = (V', E_\emptyset^*, w)$ for the first search step, where

$$\begin{aligned} E_\emptyset^* = \ & \{([i,B],[j,A]),([j,B],[i,A]) \mid ([i,B],[j,A]) \in E_\emptyset,\ w(i,j) = W\} \\ & \cup \{(s,[i,B]),([i,A],t) \mid i \in V\}. \end{aligned}$$

A search step terminates with a matching M, a weighted directed graph $G_M = (V', E_M, w)$ and a current subgraph $G_M^* = (V', E_M^*, w)$ such that G_M^* contains no M-augmenting path P. It is not hard to see that no augmenting path P with the property that after the augmentation of P, Condition 3 would be not fulfilled, exists. Otherwise, another M-augmenting path would also exist.

For the treatment of the extension step, consider the expanded MDFS-tree T_{exp}, computed by the last modified depth-first search on G_M^*. Note that this MDFS was unsucessful; i.e., no path from the start node s to the target node t was found. The goal of the extension step is to add edges to

T_{exp} such that possibly an augmenting path is found. Therefore, we have to decrease the reduced cost of edges with positive reduced cost. Such edges (i,j) have to be in $E \setminus M$. Moreover, $[i,B]$ has to be in T_{exp}. But according to the conditions which we have to maintain, some nodes in $V_B \cap T_{exp}$ are not allowed. Let B_f denote the set of these nodes. The exact definition of B_f will be given during the development of the method. Let

$$B_T = V_B \cap T_{exp} \setminus B_f \text{ and } A_T = V_A \cap T_{exp}.$$

The idea is to decrease the reduced cost $r(i,j)$ of all edges (i,j) with positive reduced cost and $[i,B] \in B_T$ by the appropriate value δ. With respect to the other end node j of edge (i,j), the following four cases can arise:

1. $[j,B] \notin B_T$ and $[j,A] \notin A_T$,
2. $[j,B] \notin B_T$ and $[j,A] \in A_T$,
3. $[j,B] \in B_T$ and $[j,A] \notin A_T$, and
4. $[j,B] \in B_T$ and $[j,A] \in A_T$.

We decrease $r(i,j)$ by decreasing $\pi(i)$ by the appropriate value δ; i.e., we decrease $\pi(i)$ by δ for all nodes i with $[i,B] \in B_T$. As a consequence of the decrease of the node weights $\pi(i)$, the reduced cost of edges e in G_M^* with end node $[i,A]$ or $[i,B]$ becomes negative. According to Condition 1, we have to increase such reduced cost. We distinguish two cases:

1. The other end node of e corresponds to A_T but not to B_T.
2. The other end node of e corresponds to B_T.

If Case 1 is fulfilled then we can increase the reduced cost of edge e by increasing the node weight of the other end node of e by δ; i.e., we increase $\pi(j)$ by δ for all nodes j such that $[j,A] \in A_T$ and $[j,B] \notin B_T$.

If Case 2 is fulfilled then the reduced cost $r(i,j)$ is decreased by 2δ. This can be corrected by increasing the set weight $\mu(E_l)$ of exactly one set E_l containing the edge (i,j) by 2δ. E_l will have the property that all edges in E_l are in E_M^* and both end nodes of these edges are contained in B_T. Two questions have to be answered:

1. What is the accurate edge set E_l for increasing its set weight?

2. What is the appropriate value δ?

To answer the first question let use consider MDFS which is used as subroutine during the search step. Review the definitions and the properties of the sets $L_{[w,A]}$ and $D_{[q,A]}$ as given at Pages 8 and 17, respectively. First, it is useful to investigate the structure of a set $D_{[q,A]}$. Let

$$D'_{[q,A]} = \{p \in V \mid [p,A] \in D_{[q,A]} \cup \{[q,A]\}\}.$$

Furthermore, let

$$\tilde{D}_{[q,A]} = \{[p,A],[p,B] \mid p \in D'_{[q,A]}\}.$$

The unique node $p \in D'_{[q,A]}$ such that p is end node of an edge $(r,p) \in M$ with $r \notin D'_{[q,A]}$ is the node q. Let $(r,q) \in M$ be the unique matched edge with end node q. We say that a path P *enters* or *leaves* $\tilde{D}_{[q,A]}$ *via an edge* $([x,B],[y,A])$ *in* $E \setminus M$ if $(x,y) \in E \setminus M$. During the performance of MDFS, for an M-augmenting path P there are three possibilities to run through a set $\tilde{D}_{[q,A]}$.

1. P enters and leaves $\tilde{D}_{[q,A]}$ via an edge in $E \setminus M$.

2. P enters $\tilde{D}_{[q,A]}$ via the matched edge $([r,A],[q,B])$ and leaves $\tilde{D}_{[q,A]}$ via an edge in $E \setminus M$.

3. P enters $\tilde{D}_{[q,A]}$ via an edge in $E \setminus M$ and leaves $\tilde{D}_{[q,A]}$ via the matched edge $([q,A],[r,B])$.

If an M-alternating path R enters $\tilde{D}_{[q,A]}$ via the edge $([r,A],[q,B])$ then, by Lemma 3, for all $v \in D'_{[q,A]}$, $[v,B] \in B_T$. Then, with respect to each edge in

$$\hat{E}^*_M = \{(i,j) \mid ([i,A],[j,B]) \in E^*_M \text{ or } ([i,B],[j,A]) \in E^*_M\}$$

with both end nodes in $D'_{[q,A]}$, we have to increase exactly one edge set containing this edge. Note that for all $v \in V$ there exists at most one current $D_{[q,A]}$ such that $v \in D'_{[q,A]}$. Hence, we define the edge set E_q corresponding to $D'_{[q,A]}$ by

$$E_q = (D'_{[q,A]} \times D'_{[q,A]}) \cap \hat{E}^*_M.$$

Note that E_q changes when $\hat{E}^*_M$ changes. If we have to increase the set weight with respect to an edge (i,j), then we choose the edge set E_q where $D_{[q,A]}$ is the current set with the property that $i,j \in D'_{[q,A]}$. Note that $[i,B],[j,B] \in B_T$ implies that $D_{[q,A]}$ exists.

Let us examine the effect of the augmentation of P to the number of edges in the current matching with both end nodes in $D'_{[q,A]}$. If the crossing of P through $D'_{[q,A]}$ is of Type 1, then this number decreases by 1. In the other cases, this number does not change. Hence, the augmentation of an augmenting path of Type 2 or 3 is always allowed but the augmentation of an augmenting path of Type 1 is only allowed if $\mu(E_q) = 0$.

Next, we will determine the accurate value for δ.

Since all node weights have to be nonnegative, δ cannot be larger than the node weight of an M-free node i. Note that with respect to an M-free node i, always $[i,B] \in B_T$ is fulfilled. Hence, all free nodes have the same node weight and $\delta \geq \pi(i)$, i M-free implies that after the change of the dual values, all node weights are nonnegative.

If edge $([i,B],[j,A])$ is of Type 1, we have to choose $\delta = r(i,j)$ for decreasing $r(i,j)$ to 0. If $([i,B],[j,A])$ is of Type 2, independently from the choice of δ, $r(i,j)$ doesn't change.

If $([i,B],[j,A])$ is of Type 3 or 4, we have to choose $\delta = 1/2r(i,j)$, since both node weights $\pi(i)$ and $\pi(j)$ will be decreased. Note that δ has to be chosen in such a way that after the extension step $r(i,j) \geq 0$ for all edges $(i,j) \in E$. Hence, δ should not be larger than the minimal reduced cost with respect to edges (i,j) with $[i,B] \in B_T$ and $[j,A] \notin A_T$, and also not larger than the half of the minimal reduced cost with respect to edges (i,j) with $[i,B],[j,B] \in B_T$ and $r(i,j) > 0$.

Since Invariant 3 has to be maintained, with respect to current $D_{[q,A]}$ the following holds: Let E'_q be the latest created edge set with respect to $D_{[q,A]}$ with $\mu(E'_q) > 0$. We call this edge set E'_q *current* with respect to $D_{[q,A]}$. If during the MDFS, no path enters $D'_{[q,A]}$ via $([r,A],[q,B])$ but there is a path R entering $D'_{[q,A]}$ via an edge in $E \setminus M$, R has to leave $D'_{[q,A]}$ via $([q,A],[r,B])$, independently if $[q,A]$ is already pushed or not. Since $(q,r) \in M$ and hence, $([q,A],[r,B]) \in E^*_M$, this is always possible. Note that R enters $D'_{[q,A]}$ through a node in A_T and leaves $D'_{[q,A]}$ through a node in A_T.

In dependence which nodes in $D_{[q,A]}$ are entering nodes of such paths R, with respect to a node $v \in D'_{[q,A]}$ the following can happen:

a) $[v, B] \in B_T$,

b) $[v, B] \notin B_T$ but $[v, A] \in A_T$, or

c) $[v, B] \notin B_T$ and $[v, A] \notin A_T$.

The problem to solve is the following: How to change the node weights of the nodes in $D'_{[q,A]}$?

It is clear that according to Condition 1, we have to increase $\pi(q)$ and also $\pi(v)$ for all entering nodes $[v, A]$ by δ. Possibly there are edges in E'_q with exactly one end node is an entering node, with both end nodes are entering nodes or with no end node is an entering nodes. With respect to all these cases, the node weights and $\mu(E'_q)$ have to be changed in such a manner that the invariants remain valid. For doing this, we increase $\pi(v)$ by δ for all $v \in D'_{[q,A]}$. Since we have increased the reduced cost of every edge in E'_q by 2δ, we decrease $\mu(E'_q)$ by 2δ. Since $\mu(E'_q)$ has to be nonnegative, δ has to be chosen such that before the change of the dual variables, $\mu(E'_q) \geq 2\delta$.

Note that there can exist nodes $[i, B] \in \tilde{D}_{[q,A]}$ which are also in T_{exp}. This are exactly those nodes in V_B which are not allowed to be in B_T. Hence, we can give the exact definition of the node set B_f as follows:

$$B_f = \{[i, B] \mid [i, B] \in \tilde{D}_{[q,A]}, D_{[q,A]} \text{ current and } [q, B] \notin T_{exp}\}.$$

Altogether, we can define δ in the following way:

$$\begin{aligned}
\delta_0 &= \pi(i), \text{ where } i \text{ is } M\text{-free}, \\
\delta_1 &= \min\{r(i,j) \mid [i, B] \in B_T \text{ and } [j, A] \notin A_T\}, \\
\delta_2 &= \min\{r(i,j) \mid [i, B], [j, B] \in B_T \text{ and } r(i,j) > 0\}, \text{ and} \\
\delta_3 &= \min\{\mu(E'_q) \mid D_{[q,A]} \text{ current}, E'_q \text{ current}, [q, B] \notin B_T \text{ and } [q, A] \in A_T\}.
\end{aligned}$$

Then we define

$$\delta = \min\{\delta_0, \delta_1, \delta_2/2, \delta_3/2\}.$$

Altogether, we have obtained the following extension step:

$\delta_0 := \pi(i)$ for an M-free node i;
$\delta_1 := \min\{r(i,j) \mid [i,B] \in B_T \text{ and } [j,A] \notin A_T\}$;
$\delta_2 := \min\{r(i,j) \mid [i,B],[j,B] \in B_T \text{ and } r(i,j) > 0\}$;
$\delta_3 := \min\{\mu(E'_q) \mid D_{[q,A]} \text{ current}, E'_q \text{ current}, [q,B] \notin B_T \text{ and } [q,A] \in A_T\}$;
$\delta := \min\{\delta_0, \delta_1, \delta_2/2, \delta_3/2\}$;
for all $[i,B] \in B_T$
do
$\pi(i) := \pi(i) - \delta$
od;
for all $[i,B] \notin B_T$, $[i,A] \in A_T$ **and** $i \notin D'_{[q,A]}$ for any current $D_{[q,A]}$
do
$\pi(i) := \pi(i) + \delta$
od;
for all $D_{[q,A]}$ current with $[q,B] \notin B_T$ but $[q,A] \in A_T$
do
$\pi(i) := \pi(i) + \delta$
od;
for all $D_{[q,A]}$ current and $[q,B] \in B_T$
do
$\mu(E_q) := \mu(E_q) + 2\delta$
od;
for all $D_{[q,A]}$ current, $[q,B] \notin B_T$ and $[q,A] \in A_T$
do
$\mu(E'_q) := \mu(E'_q) - 2\delta$, where E'_q is current with respect to $D_{[q,A]}$
od.

The correctness of the described primal-dual method follows directly from the discussion done during the development of the method.

8 An implementation of the primal-dual method

First, we will determine the number of dual changes which can occur between two augmentations in the worst case. We distinguish four cases.

Case 1: $\delta = \delta_0$

After the change of the dual variables, $\pi(i) = 0$ for all M-free nodes $i \in V$. Hence, the current matching M is of maximum weight and the algorithm terminates. Hence, Case 1 occurs at most once.

Case 2: $\delta = \delta_1$

Then, during the next search step, at least one new node $[j, A]$ enters A_T. Hence, Case 2 occurs at most n-times.

Case 3: $\delta = \delta_2$

Then, during the next search step, at least one new edge enters $\hat{E}^*_M$. Furthermore, $[i, B], [j, B] \in B_T$ as long as no augmentation is performed. Hence, this edge can leave $\hat{E}^*_M$ for the first time after the next augmentation. Hence, Case 3 occurs at most m-times.

Case 4: $\delta = \delta_3$

Then at least one current edge set E_q disappears. As long as $[q, B] \notin B_T$, the reduced cost of edges in E_q cannot be decreased. Hence, such an edge cannot produce a new current edge set before the node $[q, B]$ is put into B_T by the algorithm. But $[q, B]$ stays in B_T at least until the next augmentation. Hence, such an edge set cannot contribute to the definition of δ_3 before the next augmentation. Hence, Case 4 occurs at most m-times.

First, we will discuss the implementation of the search steps between two augmentations. Note that after an extension step, the last MDFS can be continued instead of to start a new MDFS. With respect to the primal-dual method, the following special situation has to be treated by the search step:

If according to an extension step, $\mu(E_q)$ becomes 0, the corresponding edge set leaves the family $\mathcal{F}$. We distinguish two cases.

Case 1: $D_{[q,A]}$ remains current.

Then another set $E_q \in \mathcal{F}$ corresponds to the current $D_{[q,A]}$ and it is not allowed that an augmenting path P enters and leaves $\tilde{D}_{[q,A]}$ via an edge in $E \setminus M$. Nothing is to do with respect to the data structure for the manipulation of the sets $D_{[p,A]}$, $[p, A] \in V'$.

Case 2: $D_{[q,A]}$ loss the property to be current.

If $\mu(E_q)$ becomes 0 for the last edge set E_q corresponding to $D_{[q,A]}$ then it is allowed that an augmenting path P enters and leaves $\tilde{D}_{[q,A]}$ via an edge in $E \setminus M$. Hence, $D_{[q,A]}$ loss its property to be current. Then, we have to undo the union operation done with respect to $D_{[q,A]}$. Note that this union operation has been performed before the last augmentation. Since $[q, A]$ is already pushed, no set $D_{[q,A]}$ can become current before the next augmentation.

Our goal is to extend the data structure which uses the weighted union heuristic such that the time used for the deunion operations will be, up to a small constant factor, the same as the time used for the union operations and each find operation uses only constant time. This can be done in the following way:

During the performance of an union operation, instead of changing a pointer, we add a new pointer. The current pointer of an element will be always the last created pointer. It is clear, that we use for each element at most $\log n$ extra pointers. The time used for the union operations remains essentially the same. A deunion can be performed by the deletion of the current pointers created during the corresponding union operation and the update of the set sizes and of the name of the larger subset. It is not difficult to see how to perform these changes such that the used time is, up to a small constant factor, the same as the time used for the union operations. Furthermore, it is clear that a find operation needs only constant time.

Altogether, we need for the search step between two augmentations only $O(m+n \log n)$ time. Next we will give an implementation for the computation of the δ's and the update of the dual variables.

Note that all M-free nodes i have the same dual weight. Hence, δ_0 can be computed by the consideration of any M-free node.

For the computation of δ_1, we maintain a priority queue $\mathcal{P}_1$ which contains for all $[j, A] \notin A_T$ an edge (i, j) with $[i, B] \in B_T$ and the property that (i, j) has minimum reduced cost under all such edges, if such an edge exists. Furthermore, using an array of size n, we have direct access to the element of $\mathcal{P}_1$ corresponding to the node $[j, A]$. The weight of this element will be the reduced cost of the current edge $([i, j])$ such that $[i, B] \in B_T$ and $r(i, j)$ is minimum under all such edges. We can use a heap for the realization of the priority queue.

Note that each extension step decreases all weights of the elements in the priority queue by the current δ. It is useful to maintain the property that always the weight of all elements in $\mathcal{P}_1$ has to be decreased by the same amount. Hence, we maintain the sum Δ_1 of all dual changes done so far and modify the weights in the appropriate manner.

We update $\mathcal{P}_1$ with respect to $[i, B] \in B_T$ at the moment when $[i, B]$ is added to B_T in the following way:

For all edges (i, j) with $[j, A] \notin A_T$ perform the following update operations:

(1) If no element with respect to j is contained in the priority queue then insert the element (i, j) with weight $r(i, j) + \Delta_1$.

(2) If $\mathcal{P}_1$ contains an element with respect to j with larger weight than $r(i, j) + \Delta_1$ then replace the corresponding edge by (i, j) and decrease its weight such that its value becomes $r(i, j) + \Delta_1$.

(3) If neither Case 1 nor Case 2 is fulfilled then do nothing.

Note that Step 1 needs $O(\log n)$ time and is performed at most n times. Hence, the total time used for Step 1 is $O(n \log n)$. Step 2 is performed at most m times. After the decrease, interchanging father and son, we follow the path from the element to the root of the heap as long as the weight of the father is strictly larger than the weight of its son. This can be done in $O(\log n)$ time. Hence, the total time used for Step 2 is $O(m \log n)$.

If $\delta = \delta_1$, we have to delete at least one minimal element from $\mathcal{P}_1$. Each deletion can be performed in $O(\log n)$ time and the number of deletions is bounded by the number of nodes in V. Hence, the total time for such deletions is $O(n \log n)$.

Altogether, with respect to the computation of all δ_1's between two augmentations, the used time is $O(m \log n)$.

For the computation of all δ_2's, we maintain a priority queue $\mathcal{P}_2$ which contains all edges (i, j) such that $[i, B], [j, B] \in B_T$ and $r(i, j) > 0$. We can use a heap for the realization of the priority queue.

Similary to above, each extension step decreases all weights of the elements in $\mathcal{P}_2$. Now, the amount is two times the current δ. Hence, we maintain with respect to $\mathcal{P}_2$ the sum Δ_2 of all dual changes done so far with respect to edges in $\mathcal{P}_2$ and modify the weights in the appropriate manner.

We update $\mathcal{P}_2$ with respect to $[i,B] \in B_T$ at the moment when $[i,B]$ is added to B_T in the following way:

- For all $[j,B] \in B_T$ with $r(i,j) > 0$, we insert the edge (i,j) with weight $r(i,j) + \Delta_2$ to the priority queue.

Since at most m edges are inserted, the used time is $O(m \log n)$.

If $\delta = \delta_2$, we have to delete at least one minimal element from $\mathcal{P}_2$. Each deletion can be performed in $O(\log n)$ time and the number of deletions is bounded by the number of edges in E. Hence, the total time for such deletions is $O(m \log n)$.

Altogether, with respect to the computation of all δ_2's between two augmentations, the used time is $O(m \log n)$.

For the computation of all δ_3's, we maintain a priority queue $\mathcal{P}_3$ which contains for all current $D_{[q,A]}$ with $[q,B] \notin B_T$ and $[q,A] \in A_T$ the value $\mu(E'_q)$ where E'_q is current with respect to $D_{[q,A]}$. We use a heap for the realization of the priority queue $\mathcal{P}_3$.

Each extension step decreases all weights of the elements in P_3. The amount is two times the current δ. Hence, we can use the value Δ_2 defined above and modify the weights in the appropriate manner.

We update $\mathcal{P}_3$ before the computation of δ. We have to insert for all $[q,A] \in V_A$ such that $D_{[q,A]}$ is current and $[q,A]$ was inserted to T_{exp} after the last dual change and the last augmentation, respectively but $[q,B] \notin T_{exp}$ the value $\mu(E'_q) + \Delta_2$ where E'_q is current with respect to $D_{[q,A]}$. We have to delete for all $[q,B]$ which are inserted to T_{exp} after the last dual change and the last augmentation, respectively the corresponding value if in $\mathcal{P}_3$ such a value exists. Since between two augmentation, a value enters and leaves $\mathcal{P}_3$ with respect to the same edge set at most once, the total time used by the algorithm is $O(m \log n)$.

If $\delta = \delta_3$, we have to delete at least one minimal element from $\mathcal{P}_3$. If with respect to the deleted element there exists another current edge set E'_q with respect to the same node set, then we have to insert the value $\mu(E'_q) + \Delta_2$. Each deletion and each insertion can be performed in $O(\log n)$ time. The number of deletions and insertions is bounded by the number of edges in E. Hence, the total time for such deletions is $O(m \log n)$.

Altogether, with respect to the computation of all δ_3's between two augmentations, the used time is $O(m \log n)$.

We have proven the following theorem:

Theorem 5 *The primal-dual method can be implemented in time* $O(nm \log n)$.

We can refine the implementation in several ways. We will sketch some possibilities.

Note a new edge (i, j) with $[i, B], [j, B] \in B_T$ but $i, j \in D'_{[q,A]}$ where $D_{[q,A]}$ is a current set does not help for the construction of an augmenting path. If we add to the definition of δ_2 the condition that i and j correspond to different node sets then the number of insertions and deletions in $\mathcal{P}_2$ would be decreased to n. This would lead to the property that for all current $D_{[q,A]}$ there is exactly one corresponding edge set E_q. Hence, the computation of all δ_3's would be simplified, too.

In [11], Gabow descibes essentially, how to perform the "blossom steps" in $O(m + n \log n)$ time between two augmentations. We have solved the corresponding problem with respect to MDFS by the construction of a data structure supporting the operations find, union and deunion. This is much simpler than Gabow's solution. With respect to the dual adjustment steps, we can use the same method as Gabow obtaining the same time bounds.

Acknowledgment: I thank Henning Rochow and Marek Karpinski for helpful discussions and valuable hints.

References

[1] Aho A. V., Hopcroft J. E, and Ullman J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974), 187–189.

[2] Balinski M. L., Labelling to obtain a maximum matching, in *Combinatorial Mathematics and its Applications* (R. C. Bose and T. A. Dowling, eds.), University of North Carolina Press, Chapel Hill (1969), 585–602.

[3] Ball M. O., and Derigs U., An analysis of alternative strategies for implementing matching algorithms, *Networks* **13** (1983), 517–549.

[4] Berge C., Two theorems in graph theory, *Proc. Nat. Acad. Sci. U.S.A.*, **43** (1957), 842–844.

[5] Blum N., A new approach to maximum matching in general graphs, *17th ICALP* (1990), LNCS 443, 586–597.

[6] Blum N., A simplified realization of the Hopcroft-Karp approach to maximum matching in general graphs, Research report, Universität Bonn (1999) (available at www.cs.uni-bonn.de/IV/blum/).

[7] Edmonds J., Paths, trees, and flowers, *Canad. J. Math*, **17** (1965), 449–467.

[8] Edmonds J., Maximum matching and a polyhedron with 0,1-vertices, *J. Res. Nat. Bur. Standards* **69 B** (1965), 125–130.

[9] Gabow H. N., Implementations of algorithms for maximum matching on nonbipartite graphs, Doctoral thesis, Comp. Sci. Dept., Stanford Univ., Stanford, Calif. (1973).

[10] Gabow H. N., An efficient implementation of Edmonds algorithm for maximum matching on Graph, *J. ACM*, **23** (1976), 221–234.

[11] Gabow H. N., Data structures for weighted matching and nearest common ancestors with linking, *1st SODA* (1990), 434–443.

[12] Gabow H. N., and Tarjan R. E., A linear-time algorithm for a special case of disjoint set union, *J. Comput. Syst. Sci.*, bf 30 (1985), 209–221.

[13] Gabow H. N., and Tarjan R. E., Faster scaling algorithms for general graph-matching problems, *J. ACM* **38** (1991), 815–853.

[14] Gondran M., and Minoux M., *Graphs and Algorithms*, Wiley & Sons, (1984).

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

[16] Lawler E.: *Combinatorial Optimization, Networks and Matroids*, Holt, Rinehart and Winston, (1976).

[17] Lovász L., and Plummer M. D., *Matching Theory*, North-Holland Mathematics Studies 121, North-Holland, New York (1986).

[18] Tarjan J. E., *Data Structures and Network Algorithms*, SIAM (1983).

[19] Witzgall C., and Zahn C. T. Jr., Modification of Edmonds maximum matching algorithm, *J. Res. Nat. Bur. Standards*, **69 B** (1965), 91–98.