# Implementation of $O(nm \log n)$ Weighted Matchings in General Graphs The Power of Data Structures

Kurt Mehlhorn and Guido Schäfer

Max–Planck–Institut für Informatik,
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
{mehlhorn|schaefer}@mpi-sb.mpg.de

**Abstract.** We describe the implementation of an $O(nm \log n)$ algorithm for weighted matchings in general graphs. The algorithm is a variant of the algorithm of Galil, Micali, and Gabow [GMG86] and requires the use of concatenable priority queues. No previous implementation had a worst–case guarantee of $O(nm \log n)$. We compare our implementation to the experimentally fastest implementation (called Blossom IV) due to Cook and Rohe [CR97]; Blossom IV is an implementation of Edmonds' algorithm and has a running time no better than $\Omega(n^3)$. Blossom IV requires only very simple data structures. Our experiments show that our new implementation is competitive to Blossom IV.

## 1 Introduction

The *weighted matching problem* asks for the computation of a *maximum–weight matching* in a graph. A *matching* $M$ in a graph $G = (V, E)$ is a set of edges no two of which share an endpoint. The edges of $G$ have weights associated with them and the weight of a matching is simply the sum of the weights of the edges in the matching. There are two variants of the matching problem: one can either ask for a *maximum–weight perfect matching* (a matching is *perfect* if all vertices in the graph are matched) or for a *maximum–weight matching*. Our implementation can be asked to compute either a maximum–weight perfect matching or a maximum–weight matching. Previous implementations were restricted to compute optimal perfect matchings.[1]

Edmonds [Edm65b] invented the famous blossom–shrinking algorithm and showed that weighted perfect matchings can be computed in polynomial time. A straightforward implementation of his algorithm runs in time $O(n^2m)$, where $n$ and $m$ are the number of vertices and edges of $G$, respectively. Lawler [Law76] and Gabow [Gab74] improved the running time to $O(n^3)$. The currently most efficient codes implement variants of these algorithms, Blossom IV of Cook and Rohe [CR97] being the most efficient implementation available.

Galil, Micali, and Gabow [GMG86] improved the running time to $O(nm \log n)$ and Gabow [Gab90] achieved $O(n(m + n \log n))$. Somewhat better asymptotic running

[1] The maximum–weight matching problem can be reduced to the maximum–weight perfect matching problem: the original graph is doubled and zero weight edges are inserted from each original vertex to the corresponding doubled vertex. For an original graph with $n$ vertices and $m$ edges, the reduction doubles $n$ and increases $m$ by $m + n$; it soon becomes impractical for large instances.

times are known for integral edge weights. The improved asymptotics comes mainly through the use of sophisticated data structures; for example, the algorithm of Galil, Micali, and Gabow requires the use of concatenable priority queues in which the priorities of certain subgroups of vertices can be changed by a single operation. It was open and explicitly asked in [AC93] and [CR97], whether the use of sophisticated data structures helps in practice. We answer this question in the affirmative.

The preflow–push method for maximum network flow is another example of an algorithm whose asymptotic running time improves dramatically through the use of sophisticated data structures. With the highest level selection rule and only simple data structures the worst–case running time is $O(n^2\sqrt{m})$. It improves to $\tilde{O}(nm)$ with the use of dynamic trees. None of the known efficient implementations [CG, MN99] uses sophisticated data structures and attempts to use them produced far inferior implementations [Hum96].

This paper is organized as follows: in Section 2 we briefly review Edmonds' blossom–shrinking algorithm, in Section 3 we describe our implementation, in Section 4 we report our experimental findings, and in Section 5 we offer a short conclusion.

## 2 Edmonds' Algorithm

Edmonds' blossom–shrinking algorithm is a primal–dual method based on a linear programming formulation of the maximum–weight perfect matching problem. The details of the algorithm depend on the underlying formulation. We will first give the formulation we use and then present all details of the resulting blossom–shrinking approach.

*LP Formulations:* Let $G = (V, E)$ be a general graph. We need to introduce some notions first. An incidence vector $x$ is associated with the edges of $G$: $x_e$ is set to 1, when $e$ belongs to the matching $M$; otherwise, $x_e$ is set to 0. For any subset $S \subseteq E$, we define $x(S) = \sum_{e \in S} x_e$. The edges of $G$ having both endpoints in $S \subseteq V$ are denoted by $\gamma(S) = \{uv \in E : u \in S \text{ and } v \in S\}$, and the set of all edges having exactly one endpoint in $S$ is referred to by $\delta(S) = \{uv \in E : u \in S \text{ and } v \notin S\}$. Moreover, let $\mathcal{O}$ consist of all non–singleton odd cardinality subsets of $V$: $\mathcal{O} = \{\mathcal{B} \subseteq V : |\mathcal{B}| \text{ is odd and } |\mathcal{B}| \geq 3\}$.

The maximum–weight perfect matching problem for $G$ with weight function $w$ can then be formulated as a linear program:

$$
\begin{array}{llll}
(\text{WPM}) & \text{maximize} & w^T x & \\
& \text{subject to} & x(\delta(u)) = 1 & \text{for all } u \in V, \quad (1)\\
& & x(\gamma(\mathcal{B})) \leq \lfloor |\mathcal{B}|/2 \rfloor & \text{for all } \mathcal{B} \in \mathcal{O}, \quad (2)\\
& & x_e \geq 0 & \text{for all } e \in E. \quad (3)
\end{array}
$$

(WPM)(1) states that each vertex $u$ of $G$ must be matched and (WPM)(2)–(3) assure each component $x_e$ to be either 0 or 1.

An alternative formulation exists. In the alternative formulation, the second constraint (WPM)(2) is replaced by $x(\delta(\mathcal{B})) = 1$ for all $\mathcal{B} \in \mathcal{O}$. Both the implementation of Applegate and Cook [AC93] and the implementation of Cook and Rohe [CR97] use the alternative formulation.

The formulation above is used by Galil, Micali and Gabow [GMG86] and seems to be more suitable to achieve $O(nm \log n)$ running time. It has the additional advantage that changing the constraint (WPM)(1) to $x(\delta(u)) \leq 1$ for all $u \in V$ gives a formulation of the non–perfect maximum–weight matching problem. Our implementation handles both variants of the problem.

Consider the dual linear program of (WPM). A *potential* $y_u$ and $z_\mathcal{B}$ is assigned to each vertex $u$ and non–singleton odd cardinality set $\mathcal{B}$, respectively.

$$
(\overline{\text{WPM}}) \quad \begin{array}{lrll}
\text{minimize} & \displaystyle\sum_{u \in V} y_u + \sum_{\mathcal{B} \in \mathcal{O}} \lfloor |\mathcal{B}|/2 \rfloor \, z_\mathcal{B} & & \\
\text{subject to} & z_\mathcal{B} \geq 0 & \text{for all } \mathcal{B} \in \mathcal{O}, & (1) \\
& y_u + y_v + \displaystyle\sum_{\substack{\mathcal{B} \in \mathcal{O} \\ uv \in \gamma(\mathcal{B})}} z_\mathcal{B} \geq w_{uv} & \text{for all } uv \in E. & (2)
\end{array}
$$

The *reduced cost* $\pi_{uv}$ of an edge $uv$ with respect to a dual solution $(y, z)$ of ($\overline{\text{WPM}}$) is defined as given below. We will say an edge $uv$ is *tight*, when its reduced cost $\pi_{uv}$ equals 0.

$$\pi_{uv} = y_u + y_v - w_{uv} + \sum_{\substack{\mathcal{B} \in \mathcal{O} \\ uv \in \gamma(\mathcal{B})}} z_\mathcal{B}.$$

*Blossom–Shrinking Approach:* Edmonds' blossom–shrinking algorithm is a primal–dual method that keeps a primal (not necessarily feasible) solution $x$ to (WPM) and also a dual feasible solution $(y, z)$ to ($\overline{\text{WPM}}$); the primal solution may violate (WPM)(1). The solutions are adjusted successively until they are recognized to be optimal. The optimality of $x$ and $(y, z)$ will be assured by the feasibility of $x$ and $(y, z)$ and the validity of the complementary slackness conditions (CS)(1)–(2):

$$
(\text{CS}) \quad \begin{array}{rcll}
x_{uv} > 0 & \Longrightarrow & \pi_{uv} = 0 & \text{for all edges } uv \in E, \quad (1) \\
z_\mathcal{B} > 0 & \Longrightarrow & x(\gamma(\mathcal{B})) = \lfloor |\mathcal{B}|/2 \rfloor & \text{for all } \mathcal{B} \in \mathcal{O}. \quad (2)
\end{array}
$$

(CS)(1) states that matching edges must be tight and (CS)(2) states that non–singleton sets $\mathcal{B}$ with positive potential are *full*, i.e., a maximum number of edges in $\mathcal{B}$ are matched.

The algorithm starts with an arbitrary matching $M$.[2] $x$ satisfies (WPM)(2)–(3). Each vertex $u$ has a potential $y_u$ associated with it and all $z_\mathcal{B}$'s are 0. The potentials are chosen such that $(y, z)$ is dual feasible to ($\overline{\text{WPM}}$) and, moreover, satisfies (CS)(1)–(2) with respect to $x$. The algorithm operates in phases.

In each phase, two additional vertices get matched and hence do no longer violate (WPM)(1). After $O(n)$ phases, either all vertices will satisfy (WPM)(1) and thus the computed matching is optimal, or it has been discovered that no perfect matching exists.

The algorithm attempts to match free vertices by growing so–called *alternating trees* rooted at free vertices.[3] Each alternating tree $T_r$ is rooted at a free vertex $r$. The edges in $T_r$ are tight and alternately matched and unmatched with respect to the current matching $M$. We further assume a labeling for the vertices of $T_r$: a vertex $u \in T_r$ is labeled *even* or *odd*, when the path from $u$ to $r$ is of even or odd length,

[2] We will often use the concept of a matching $M$ and its incidence vector $x$ interchangeably.

[3] We concentrate on a *multiple search tree* approach, where alternating trees are grown from all free vertices simultaneously. The *single search tree* approach, where just one alternating tree is grown at a time, is slightly simpler to describe, gives the same asymptotic running time, but leads to an inferior implementation.

respectively. Vertices that are not part of any alternating tree are matched and said to be *unlabeled*. We will use $u^+$, $u^-$ or $u^\varnothing$ to denote an even, odd or unlabeled vertex.

An alternating tree is extended, or *grown*, from even labeled tree vertices $u^+ \in T_r$: when a tight edge $uv$ from $u$ to a non–tree vertex $v^\varnothing$ exists, the edge $uv$ and also the matching edge $vw$ of $v$ (which must exist, since $v$ is unlabeled) is added to $T_r$. Here, $v$ and $w$ get labeled odd and even, respectively.

When a tight edge $uv$ with $u^+ \in T_r$ and $v^+ \in T_{r'}$ exists, with $T_r \neq T_{r'}$, an *augmenting path* from $r$ to $r'$ has been discovered. A path $p$ from a free vertex $r$ to another free vertex $r'$ is called *augmenting*, when the edges along $p$ are alternately in $M$ and not in $M$ (the first and last edge are unmatched). Let $p_r$ denote the tree path in $T_r$ from $u$ to $r$ and, correspondingly, $p_{r'}$ the tree path in $T_{r'}$ from $v$ to $r'$. By $\overline{p_r}$ we denote the path $p_r$ in reversed order. The current matching $M$ is augmented by $(\overline{p_r}, uv, p_{r'})$, i.e., all non–matching edges along that path become matching edges and vice versa. After that, all vertices in $T_r$ and $T_{r'}$ will be matched; therefore, we can destroy $T_r$ and $T_{r'}$ and unlabel all their vertices.

Assume a tight edge $uv$ connecting two even tree vertices $u^+ \in T_r$ and $v^+ \in T_r$ exists (in the same tree $T_r$). We follow the tree paths from $u$ and $v$ towards the root until the lowest common ancestor vertex *lca* has been found. *lca* must be even by construction of $T_r$ and the simple cycle $C = (lca, \ldots, u, v, \ldots, lca)$ is full. The subset $\mathcal{B} \subseteq V$ of vertices on $C$ are said to form a *blossom*, as introduced by Edmonds [Edm65b]. A key observation is that one can *shrink* blossoms into an even labeled pseudo–vertex, say *lca*, and continue the growth of the alternating trees in the resulting graph.[4] To shrink a cycle $C$ means to collapse all vertices of $\mathcal{B}$ into a single pseudo–vertex *lca*. All edges $uv$ between vertices of $\mathcal{B}$, i.e., $uv \in \gamma(\mathcal{B})$, become non–existent and all edges $uv$ having exactly one endpoint $v$ in $\mathcal{B}$, i.e., $uv \in \delta(\mathcal{B})$, are replaced by an edge from $u$ to *lca*.

However, we will regard these vertices to be conceptually shrunk into a new pseudo–vertex only. Since pseudo–vertices might get shrunk into other pseudo–vertices, the following view is appropriate: the current graph is partitioned into a *nested family* of odd cardinality subsets of $V$. Each odd cardinality subset is called a blossom. A blossom might contain other blossoms, called *subblossoms*. A *trivial* blossom corresponds to a single vertex of $G$. A blossom $\mathcal{B}$ is said to be a *surface blossom*, if $\mathcal{B}$ is not contained in another blossom. All edges lying completely in any blossom $\mathcal{B}$ are *dead* and will not be considered by the algorithm; all other edges are *alive*.

*Dual Adjustment:* The algorithm might come to a halt due to the lack of further tight edges. Then, a *dual adjustment* is performed: the dual solution $(y, z)$ is adjusted such that the objective value of $(\overline{\text{WPM}})$ decreases. However, the adjustment will be of the kind such that $(y, z)$ stays dual feasible and, moreover, preserves (CS)(1)–(2). One way to accomplish the desired result is to update the potentials $y_u$ of all vertices $u \in V$ and the potential $z_{\mathcal{B}}$ of each non–trivial surface blossom $\mathcal{B}$ by some $\delta > 0$ as stated below:

$$y_u = y_u + \sigma\delta, \quad \text{and} \quad z_{\mathcal{B}} = z_{\mathcal{B}} - 2\sigma\delta.$$

The *status indicator* $\sigma$ is defined as follows: $\sigma$ equals $-1$ or $1$ for even or odd tree blossoms (trivial or non–trivial), respectively, and equals 0, otherwise. The status of

[4] The crucial point is that any augmenting path in the resulting graph can be lifted to an augmenting path in the original graph.

a vertex is the status of the surface blossom containing it. Let $T_{r_1}, \ldots, T_{r_k}$ denote the current alternating trees. The value of $\delta$ must be chosen as $\delta = \min\{\delta_2, \delta_3, \delta_4\}$, with

$$\begin{aligned}
\delta_2 &= \min_{uv \in E} \{\pi_{uv} \ : \ u^+ \in T_{r_i} \text{ and } v^{\varnothing} \text{ not in any tree}\}, \\
\delta_3 &= \min_{uv \in E} \{\pi_{uv}/2 \ : \ u^+ \in T_{r_i} \text{ and } v^+ \in T_{r_j}\}, \\
\delta_4 &= \min_{\mathcal{B} \in \mathcal{O}} \{z_{\mathcal{B}}/2 \ : \ \mathcal{B}^- \in T_{r_i}\},
\end{aligned}$$

where $T_{r_i}$ and $T_{r_j}$ denote any alternating tree, with $1 \leq i, j \leq k$.[5] The minimum of an empty set is defined to be $\infty$. When $\delta = \infty$, the dual linear program ($\overline{\text{WPM}}$) is unbounded and thus no optimal solution to (WPM) exists (by weak duality).

When $\delta$ is chosen as $\delta_4$, the potential of an odd tree blossom, say $\mathcal{B}^- \in T_r$, will drop to 0 after the dual adjustment and therefore is not allowed to participate in further dual adjustments. The action to be taken is to *expand* $\mathcal{B}$, i.e., the defining subblossoms $\mathcal{B}_1, \ldots, \mathcal{B}_{2k+1}$ of $\mathcal{B}$ are lifted to the surface and $\mathcal{B}$ is abandoned. Since $\mathcal{B}$ is odd, there exists a matching tree edge $ub$ and a non–matching tree edge $dv$. The vertices $b$ and $d$ in $\mathcal{B}$ are called the *base* and *discovery* vertex of $\mathcal{B}$. Assume $\mathcal{B}_i$ and $\mathcal{B}_j$ correspond to the subblossoms containing $b$ and $d$, respectively. Let $p$ denote the even length alternating (alive) path from $\mathcal{B}_j$ to $\mathcal{B}_i$ lying exclusively in $\gamma(\mathcal{B})$. The path $p$ and therewith all subblossoms on $p$ are added to $T_r$; the subblossoms are labeled accordingly. All remaining subblossoms get unlabeled and leave $T_r$.

*Realizations:* We argued before that the algorithm terminates after $O(n)$ phases. The number of dual adjustments is bounded by $O(n)$ per phase.[6] A *union–find* data structure supporting a *split* operation, additionally, is sufficient to maintain the surface graph in time $O(m + n \log n)$ per phase.[7] The existing realizations of the blossom–shrinking algorithm differ in the way they determine the value of $\delta$ and perform a dual adjustment.

The most trivial realization inspects all edges and explicitly updates the vertex and blossom potentials and thus needs $O(n + m)$ time per dual adjustment. The resulting $O(n^2 m)$, or $O(n^4)$, approach was suggested first by Edmonds [Edm65a].

Lawler [Law76] and Gabow [Gab74] improved the asymptotic running time to $O(n^3)$. The idea is to keep the *best edge*, i.e., the edge having minimum reduced cost, for each non–tree vertex $v^{\varnothing}$ and for each odd tree vertex $v^- \in T_r$ to an even labeled tree vertex (the necessity of the latter is due to the expansion of blossoms). Moreover, each even tree blossom knows its best edges to other even tree blossoms. The time required for the determination of $\delta$ and to perform a dual adjustment is therewith reduced to $O(n)$. The overall running time of $O(n(m + n^2))$ ensues.

---

[5] Notice that $T_{r_i}$ and $T_{r_j}$ need not necessarily to be different in the definition of $\delta_3$.

[6] Whenever $\delta = \delta_2, \delta_3$, at least one vertex becomes an even labeled tree vertex, or a phase terminates. An even tree vertex stays even and resides in its tree until that tree gets destroyed. Thus, $\delta = \delta_2, \delta_3$ may happen $O(n)$ times per phase. The maximum cardinality of a blossom is $n$ and thence $\delta = \delta_4$ occurs $O(n)$ times per phase.

[7] Each vertex knows the *name* of its surface blossom. On a *shrink* or an *expand* step all vertices of the smaller group are renamed.

## 3 Implementation

*Blossom IV:* The implementation (called Blossom IV) of Cook and Rohe [CR97] is the most efficient code for weighted perfect matchings in general graphs currently available. The algorithm is implemented in C. The comparison to other implementations is made in two papers: (1) In [CR97] Blossom IV is compared to the implementation of Applegate and Cook [AC93]. It is shown that Blossom IV is substantially faster. (2) In [AC93] the implementation of Applegate and Cook is compared to other implementations. The authors show that their code is superior to all other codes.

A user of Blossom IV can choose between three modes: a single search tree approach, a multiple search tree approach, and a refinement of the multiple search tree approach, called the *variable $\delta$ approach.* In the variable $\delta$ approach, each alternating tree $T_{r_i}$ chooses its own dual adjustment value $\delta_{r_i}$ so as to maximize the decrease in the dual objective value. A heuristic is used to make the choices (an exact computation would be too costly). The variable $\delta$ approach does not only lead to a faster decrease of the dual objective value, it, typically, also creates tight edges faster (at least, when the number of distinct edge weights is small). The experiments in [CR97] show that the variable $\delta$ approach is superior to the other approaches in practice. We make all our comparisons to Blossom IV with the variable $\delta$ approach.

Blossom IV uses a heuristic to find a good initial solution (jump start) and a price–and–repair heuristic for sparsening the input graph. We discuss both heuristics in Section 4.

*Our Implementation:* We come to our implementation. It has a worst–case running time of $O(nm\log n)$, uses (concatenable) priority queues extensively and is able to compute a non–perfect or a perfect maximum–weight matching. It is based on LEDA [MN99] and the implementation language is C++. The implementation can run either a single search tree approach or a multiple search tree approach; it turned out that the additional programming expenditure for the multiple search tree approach is well worth the effort regarding the efficiency in practice. Comparisons of our multiple search tree algorithm to the variable $\delta$ approach of Blossom IV will be given in Section 4.

The underlying strategies are similar to or have been evolved from the ideas of Galil, Micali and Gabow [GMG86]. However, our approach differs with regard to the maintenance of the varying potentials and reduced costs. Galil et al. handle these varying values within the priority queues, i.e., by means of an operation that changes all priorities in a priority queue by the same amount, whereas we establish a series of formulae that enable us to compute the values on demand. The time required to perform a dual adjustment is considerably improved to $O(1)$. Next, the key ideas of our implementation will be sketched briefly. As above, we concentrate on the description of the multiple search tree approach.

The definitions of $\delta_2, \delta_3$ and $\delta_4$ suggest to keep a priority queue for each of those; which we will denote by *delta2*, *delta3* and *delta4*, respectively. The priorities stored in each priority queue correspond to the value of interest, i.e., to (one half of) the reduced cost of edges for *delta2* and *delta3* and to blossom potentials for *delta4*. The minor difficulty that these priorities decrease by $\delta$ with each dual adjustment is simply overcome as follows. We keep track of the amount $\Delta = \sum \delta_i$ of all dual adjustments and compute the *actual* priority $\widetilde{p}$ of any element in the priority queues taking $\Delta$ into consideration: $\widetilde{p} = p - \Delta$. A dual adjustment by $\delta$ then easily reduces to an increase of $\Delta$ by $\delta$; a dual adjustment takes time $O(1)$ (additional details to affirm that will be given subsequently).

Some details for the maintenance of *delta2* are given next. We associate a *concatenable priority queue* $P_{\mathcal{B}}$ with each surface blossom $\mathcal{B}$. A concatenable priority queue supports all standard priority queue operations and, in addition, a *concat* and *split* operation. Moreover, the elements are regarded to form a sequence. *concat* concatenates the sequences of two priority queues and, conversely, *split* splits the sequence of a priority queue at a given item into two priority queues. Both operations can be achieved to run in time $O(\log n)$, see, for example, [Meh84, Section III.5.3] and [AHU74, Section 4.12]. Our implementation is based on $(2, 16)$–trees.

$P_{\mathcal{B}}$ contains exactly one element $\langle p, u\rangle$ for each vertex $u$ contained in $\mathcal{B}$. Generally, $p$ represents the reduced cost of the best edge of $u$ to an even labeled tree vertex. More precise, let $T_{r_1}, \ldots, T_{r_k}$ denote the alternating trees at any stage of the blossom–shrinking approach. Every vertex $u$ is associated with a series of (at most $k$) incident edges $uv_1, \ldots, uv_k$ and their reduced costs $\pi_{uv_1}, \ldots, \pi_{uv_k}$ (maintained by a standard priority queue). Each edge $uv_i$ represents the best edge from $u$ to an even tree vertex $v_i^+ \in T_{r_i}$. The reduced cost $\pi_{uv_{i^*}}$ of $u$'s best edge $uv_{i^*}$ (along all best edges $uv_i$ associated with $u$), is the priority stored with the element $\langle p, u\rangle$ in $P_{\mathcal{B}}$.

When a new blossom $\mathcal{B}$ is formed by $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{2k+1}$ the priority queues $P_{\mathcal{B}_1}, P_{\mathcal{B}_2}, \ldots, P_{\mathcal{B}_{2k+1}}$ are concatenated one after another and the resulting priority queue $P_{\mathcal{B}}$ is assigned to $\mathcal{B}$. Thereby, we keep track of each $t_i$, $1 \leq i \leq 2k+1$, the last item in $\mathcal{B}_i$. Later, when $\mathcal{B}$ gets expanded the priority queues to $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{2k+1}$ can easily be recovered by splitting the priority queue of $\mathcal{B}$ at each item $t_i$, $1 \leq i \leq 2k+1$.

Whenever a blossom $\mathcal{B}$ becomes unlabeled, it sends its best edge and the reduced cost of that edge to *delta2*. Moreover, when the best edge of $\mathcal{B}$ changes, the appropriate element in *delta2* is adjusted. When a non–tree blossom $\mathcal{B}^{\varnothing}$ becomes an odd tree blossom its element in *delta2* is deleted. These insertions, adjustments and deletions on *delta2* contribute $O(n \log n)$ time per phase.

We come to *delta3*. Each tree $T_{r_i}$ maintains its own priority queue *delta3*$_{r_i}$ containing all *blossom forming* edges, i.e., the edges connecting two even vertices in $T_{r_i}$. The priority of each element corresponds to one half of the reduced cost; note, however, that the actual reduced cost of each element is computed as stated above. The edges inserted into *delta3*$_{r_i}$ are assured to be alive. However, during the course of the algorithm some edges in *delta3*$_{r_i}$ might become dead. We use a *lazy–deletion* strategy for these edges: dead edges are simply discarded when they occur as the minimum element of *delta3*$_{r_i}$. The minimum element of each *delta3*$_{r_i}$ is sent to *delta3*. Moreover, each tree $T_{r_i}$ sends its best edge $uv$ with $u^+ \in T_{r_i}$ and $v^+ \in T_{r_j}$, $T_{r_i} \neq T_{r_j}$, to *delta3*. When a tree $T_{r_i}$ gets destroyed, its (two) representatives are deleted from *delta3* and *delta3*$_{r_i}$ is freed. Since $m \leq n^2$, the time needed for the maintenance of all priority queues responsible for *delta3* is $O(m \log n)$ per phase.

Handling *delta4* is trivial. Each non–trivial odd surface blossom $\mathcal{B}^- \in T_{r_i}$ sends an element to *delta4*. The priority corresponds to one half of the potential $z_{\mathcal{B}}$.

What remains to be shown is how to treat the varying blossom and vertex potentials as well as the reduced cost of all edges associated with the vertices. The crux is that with each dual adjustment all these values uniformly change by some amount of $\delta$.

For example, consider an even surface blossom $\mathcal{B}^+ \in T_{r_i}$. The potential of $\mathcal{B}$ changes by $+2\delta$, the potential of each vertex $u \in \mathcal{B}$ by $-\delta$ and the reduced costs of all edges assosiated with each $u \in \mathcal{B}$ change by $-2\delta$ with a dual adjustment by $\delta$. Taking advantage of that fact, the actual value of interest can again be computed by taking $\Delta$ and some additional information into consideration. The idea is as follows.

Each surface blossom $\mathcal{B}$ has an offset $\mathit{offset}_{\mathcal{B}}$ assigned to it. This offset is initially set to 0 and will be adjusted whenever $\mathcal{B}$ changes its status. The formulae to compute the actual potential $\widetilde{z}_{\mathcal{B}}$ for a (non–trivial) surface blossom $\mathcal{B}$, the actual potential $\widetilde{y}_u$ for a vertex $u$ (with surface blossom $\mathcal{B}$) and the actual reduced cost $\widetilde{\pi}_{uv_i}$ of an edge $uv_i$ assosicated with $u$ (and surface blossom $\mathcal{B}$) are given below:

$$\widetilde{z}_{\mathcal{B}} = z_{\mathcal{B}} - 2\mathit{offset}_{\mathcal{B}} - 2\sigma\Delta, \tag{1}$$
$$\widetilde{y}_u = y_u + \mathit{offset}_{\mathcal{B}} + \sigma\Delta, \tag{2}$$
$$\widetilde{\pi}_{uv_i} = \pi_{uv_i} + \mathit{offset}_{\mathcal{B}} + (\sigma - 1)\Delta. \tag{3}$$

Here, $\sigma$ and $\Delta$ are defined as above. It is not difficult to affirm the offset update of a surface blossom $\mathcal{B}$:

$$\mathit{offset}_{\mathcal{B}} = \mathit{offset}_{\mathcal{B}} + (\sigma - \sigma')\Delta, \tag{4}$$

which is necessary at the point of time, when $\mathcal{B}$ changes its status indicator from $\sigma$ to $\sigma'$. To update the blossom offset takes time $O(1)$. We conclude, that each value of interest can be computed (if required) in time $O(1)$ by the formulae (1)–(3).

It is an easy matter to handle the blossom offsets in an expand step for $\mathcal{B}$: $\mathit{offset}_{\mathcal{B}}$ is assigned to each offset of the defining subblossoms of $\mathcal{B}$. However, it is not obvious in which way one can cope with different blossom offsets in a shrink step. Let $\mathcal{B}$ denote the blossom that is going to be formed by the defining subblossoms $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{2k+1}$. Each odd labeled subblossom $\mathcal{B}_i$ is made even by adjusting its offset as in (4). The corresponding offsets $\mathit{offset}_{\mathcal{B}_1}, \mathit{offset}_{\mathcal{B}_2}, \ldots, \mathit{offset}_{\mathcal{B}_{2k+1}}$ may differ in value. However, we want to determine a common offset value $\mathit{offset}_{\mathcal{B}}$ such that the actual potential and the actual reduced costs associated with each vertex $u \in \mathcal{B}_i$ can be computed with respect to $\mathit{offset}_{\mathcal{B}}$.

The following strategy assures that the offsets of all defining subblossoms $\mathcal{B}_i$, $1 \le i \le 2k+1$, are set to zero and thus the desired result is achieved by the common offset $\mathit{offset}_{\mathcal{B}} = 0$.

Whenever a surface blossom $\mathcal{B}'$ (trivial or non–trivial) becomes an even tree blossom, its offset $\mathit{offset}_{\mathcal{B}'}$ is set to zero. Consequently, in order to preserve the validity of (2) and (1) for the computation of the actual potential $\widetilde{y}_u$ of each vertex $u \in \mathcal{B}'$ and of the actual potential $\widetilde{z}_{\mathcal{B}'}$ of $\mathcal{B}'$ itself (when $\mathcal{B}'$ is non–trivial only), the following adjustments have to be performed:

$$y_u = y_u + \mathit{offset}_{\mathcal{B}'},$$
$$z_{\mathcal{B}'} = z_{\mathcal{B}'} - 2\mathit{offset}_{\mathcal{B}'}.$$

Moreover, the stored reduced cost $\pi_{uv_i}$ of each edge $uv_i$ associated with each vertex $u \in \mathcal{B}'$ is subject to correction:

$$\pi_{uv_i} = \pi_{uv_i} + \mathit{offset}_{\mathcal{B}'}.$$

Observe that the adjustments are performed at most once per phase for a fixed vertex. Thus, the time required for the potential adjustments is $O(n)$ per phase. On the other hand, the corrections of the reduced costs contributes total time $O(m \log n)$ per phase.

In summary, we have established a convenient way to handle the varying potentials as well as the reduced costs of edges associated with a vertex. The values of interest can be computed on demand by the formulae developed. The additional overhead produced by the offset maintenance has been proved to consume $O(m \log n)$ time per phase. This concludes the description of our approach (for a more extensive discussion the reader is referred to [Sch00]).

*Correctness:* Blossom IV and our program does not only compute an optimal matching $M$ but also an optimal dual solution. This makes it easy to verify the correctness of a solution. One only has to check that $M$ is a (perfect) matching, that the dual solution is feasible (in particular, all edges must have non-negative reduced costs), and that the complementary slackness conditions are satisfied.

## 4 Experimental Results

We experimented with three kinds of instances: Delaunay instances, (sparse and dense) random instances having a perfect matching, and complete geometric instances.

For the Delaunay instances we chose $n$ random points in the unit square and computed their Delaunay triangulation using the LEDA Delaunay implementation. The edge weights correspond to the Euclidean distances scaled to integers in the range $[1, \ldots, 2^{16})$. Delaunay graphs are known to contain perfect matchings [Dil90].

For the random instances we chose random graphs with $n$ vertices. The number of edges for sparse graphs was chosen as $m = \alpha n$ for small values of $\alpha$, $\alpha \leq 10$. For dense graphs the density is approximately 20%, 40% and 60% of the density of a complete graph. Random weights out of the range $[1, \ldots, 2^{16})$ were assigned to the edges. We checked for perfect matchings with the LEDA cardinality matching implementation.

Complete geometric instances were induced by $n$ random points in an $n \times n$ square and their Euclidean distances.

*Experimental Setting:* All our running times are in seconds and are the average of $t = 5$ runs, unless stated otherwise. All experiments were performed on a Sun Ultra Sparc, 333 Mhz.

*Initial Solution, Single and Multiple Search Tree Strategies:* We implemented two strategies for finding initial solutions, both of them well known and also used in previous codes. The *greedy heuristics* first sets the vertex potentials: the potential $y_v$ of a vertex $v$ is set to one–half the weight of the heaviest incident edge. This guarantees that all edges have non–negative reduced cost. It then chooses a matching within the tight edges in a greedy fashion. The fractional matching heuristic [DM86] first solves the fractional matching problem (constraints (WPM)(1) and (WPM)(3)). In the solution all variables are half–integral and the edges with value 1/2 form odd length cycles. The initial matching consists of the edges with value 1 and of $\lfloor |C|/2 \rfloor$ edges from every odd cycle. Applegate and Cook [AC93] describe how to solve the fractional matching problem. Our fractional matching algorithm uses priority queues and similar strategies to the one evolved above. The priority queue based approach appears to be highly efficient.[8] Table 1 compares the usage of different heuristics in

[8] Experimental results (not presented in this paper) showed that the priority queue approach is substantially faster than the specialized algorithm of LEDA to compute a maximum–weight (perfect) matching in a bipartite graph.

combination with the single search tree (SST) and the multiple search tree (MST) approach.

**Table 1.** SST vs. MST on Delaunay graphs. $^-$, $^+$ or $^*$ indicates usage of no, the greedy or the fractional matching heuristic. The time needed by the greedy or the fractional matching heuristic is shown in the columns GY and FM.

| $n$ | SST$^-$ | MST$^-$ | SST$^+$ | MST$^+$ | GY | SST$^*$ | MST$^*$ | FM | $t$ |
|---|---|---|---|---|---|---|---|---|---|
| 10000 | 37.01 | 6.27 | 24.05 | 4.91 | 0.13 | 5.79 | 3.20 | 0.40 | 5 |
| 20000 | 142.93 | 14.81 | 89.55 | 11.67 | 0.24 | 18.54 | 8.00 | 0.83 | 5 |
| 40000 | 593.58 | 31.53 | 367.37 | 25.51 | 0.64 | 76.73 | 17.41 | 1.78 | 5 |

The fractional matching heuristic is computational more intensive than the greedy heuristic, but leads to overall improvements of the running time. The multiple search tree strategy is superior to the single search tree strategy with both heuristics. We therefore take the multiple search tree strategy with the fractional matching heuristic as our canonical implementation; we refer to this implementation as MST$^*$. Blossom IV also uses the fractional matching heuristic for constructing an initial solution. We remark that the difference between the two heuristics is more pronounced for the single search tree approach.

Table 2 compares Blossom IV with the fractional matching heuristic, multiple search trees without (B4$^*$) and with (B4$^*_{\mathrm{var}}$) variable $\delta$'s with MST$^*$. We used Delaunay graphs.

**Table 2.** B4$^*$, B4$^*_{\mathrm{var}}$ vs. MST$^*$ on Delaunay graphs.

| $n$ | B4$^*$ | B4$^*_{\mathrm{var}}$ | MST$^*$ | $t$ |
|---|---|---|---|---|
| 10000 | 73.57 | 4.11 | 3.37 | 5 |
| 20000 | 282.20 | 12.34 | 7.36 | 5 |
| 40000 | 1176.58 | 29.76 | 15.84 | 5 |

The table shows that the variable $\delta$ approach B4$^*_{\mathrm{var}}$ makes a tremendous difference for Blossom IV and that MST$^*$ is competitive with B4$^*_{\mathrm{var}}$.

*Influence of Edge Weights:* Table 3 shows the influence of the edge weights on the running time. We took random graphs with $m = 4n$ edges and random edge weights in the range $[1, \ldots, b]$ for different values of $b$. For $b = 1$, the problem is unweighted.

**Table 3.** $B4^*$, $B4^*_{var}$ vs. $MST^*$ on random graphs with variable weight range.

| $n$ | $\alpha$ | $b$ | $B4^*$ | $B4^*_{var}$ | $MST^*$ | $t$ |
|---|---|---|---|---|---|---|
| 10000 | 4 | 1 | 3.98 | 3.99 | 0.85 | 1 |
| 10000 | 4 | 10 | 2.49 | 3.03 | 2.31 | 1 |
| 10000 | 4 | 100 | 3.09 | 3.10 | 2.58 | 1 |
| 10000 | 4 | 1000 | 17.41 | 8.40 | 2.91 | 1 |
| 10000 | 4 | 10000 | 13.69 | 11.91 | 2.78 | 1 |
| 10000 | 4 | 100000 | 12.06 | 11.20 | 2.69 | 1 |

The running time of $B4^*$ and $B4^*_{var}$ depends significantly on the size of the range, the running time of $MST^*$ depends only weakly (except for the unweighted case which is simpler). $MST^*$ is superior to $B4^*_{var}$.

We try an explanation. When the range of edge weights is small, a dual adjustment is more likely to make more than one edge tight. Also it seems to take fewer dual adjustments until an augmenting path is found. Since dual adjustments are cheaper in our implementation ($O(m \log n)$ for all adjustments in a phase of our implementation versus $O(n)$ for a single adjustment in Blossom IV), our implementation is less harmed by large numbers of adjustments.

*Asymptotics:* Tables 4 and 5 give some information about the asymptotics. For Tab. 4 we have fixed the graph density at $m = 6n$ and varied $n$.

**Table 4.** $B4^*$, $B4^*_{var}$ vs. $MST^*$ on sparse random graphs with $m = 6n$.

| $n$ | $\alpha$ | $B4^*$ | $B4^*_{var}$ | $MST^*$ | $t$ |
|---|---|---|---|---|---|
| 10000 | 6 | 20.94 | 18.03 | 3.51 | 5 |
| 20000 | 6 | 82.96 | 53.87 | 9.97 | 5 |
| 40000 | 6 | 194.48 | 177.28 | 29.05 | 5 |

The running times of $B4^*_{var}$ and $MST^*$ seem to grow less than quadratically (with $B4^*_{var}$ taking about six times as long as $MST^*$). Table 5 gives more detailed information. We varied $n$ and $\alpha$.

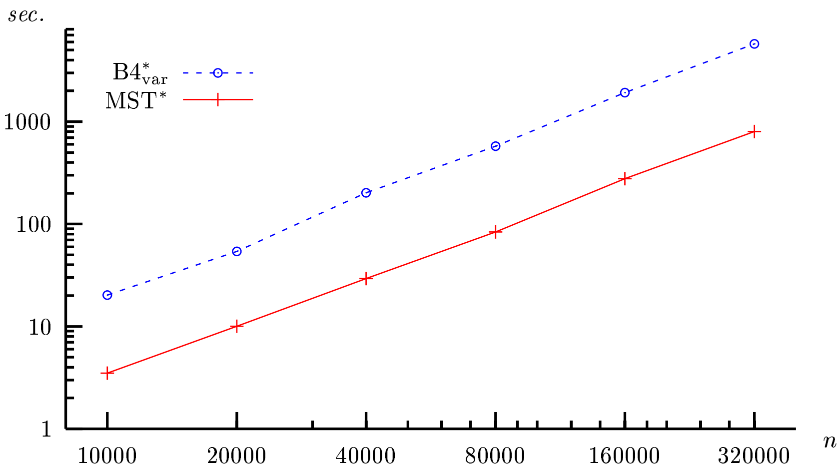


**Fig. 1.** Asymptotics of $B4^*_{var}$ and $MST^*$ algorithm on sparse random instances ($\alpha = 6$).

**Table 5.** $B4^*$, $B4^*_{var}$ vs. $MST^*$ on sparse random graphs with $m = \alpha n$.

| $n$ | $\alpha$ | $B4^*$ | $B4^*_{var}$ | $MST^*$ | $t$ |
|---|---|---|---|---|---|
| 10000 | 6 | 20.90 | 20.22 | 3.49 | 5 |
| 10000 | 8 | 48.50 | 22.83 | 5.18 | 5 |
| 10000 | 10 | 37.49 | 30.78 | 5.41 | 5 |
| 20000 | 6 | 96.34 | 54.08 | 10.04 | 5 |
| 20000 | 8 | 175.55 | 89.75 | 12.20 | 5 |
| 20000 | 10 | 264.80 | 102.53 | 15.06 | 5 |
| 40000 | 6 | 209.84 | 202.51 | 29.27 | 5 |
| 40000 | 8 | 250.51 | 249.83 | 36.18 | 5 |
| 40000 | 10 | 710.08 | 310.76 | 46.57 | 5 |

A log–log plot indicating the asymptotics of $B4^*_{var}$ and our $MST^*$ algorithm on random instances ($\alpha = 6$) is depicted in Fig. 1.

*Variance in Running Time:* Table 6 gives information about the variance in running time. We give the best, worst, and average time of five instances. The fluctuation is about the same for both implementations.

**Table 6.** $B4^*_{var}$ vs. MST* on sparse random graphs with $m = 6n$.

| $n$ | $\alpha$ | $B4^*_{var}$ | | | MST* | | | $t$ |
|---|---|---|---|---|---|---|---|---|
| | | best | worst | average | best | worst | average | |
| 10000 | 6 | 16.88 | 20.03 | 18.83 | 3.34 | 4.22 | 3.78 | 5 |
| 20000 | 6 | 49.02 | 60.74 | 55.15 | 9.93 | 11.09 | 10.30 | 5 |
| 40000 | 6 | 162.91 | 198.11 | 180.88 | 25.13 | 32.24 | 29.09 | 5 |

*Dense Graphs and Price and Repair:* Our experiments suggest that MST* is superior to $B4^*_{var}$ on sparse graphs. Table 7 shows the running time on dense graphs. Our algorithm is superior to Blossom IV even on these instances.

**Table 7.** B4*, $B4^*_{var}$ vs. MST* on dense random graphs. The density is approximately 20%, 40% and 60%.

| $n$ | $m$ | B4* | $B4^*_{var}$ | MST* | $t$ |
|---|---|---|---|---|---|
| 1000 | 100000 | 6.97 | 5.84 | 1.76 | 5 |
| 1000 | 200000 | 16.61 | 11.35 | 3.88 | 5 |
| 1000 | 300000 | 18.91 | 18.88 | 5.79 | 5 |
| 2000 | 200000 | 46.71 | 38.86 | 8.69 | 5 |
| 2000 | 400000 | 70.52 | 70.13 | 16.37 | 5 |
| 2000 | 600000 | 118.07 | 115.66 | 23.46 | 5 |
| 4000 | 400000 | 233.16 | 229.51 | 42.32 | 5 |
| 4000 | 800000 | 473.51 | 410.43 | 92.55 | 5 |
| 4000 | 1200000 | 523.40 | 522.52 | 157.00 | 5 |

Blossom IV provides a price–and–repair heuristic which allows it to run on implicitly defined complete geometric graphs (edge weight = Euclidean distance). The running time on these instances is significantly improved for $B4^*_{var}$ using the price–and–repair heuristic as can be seen in Tab. 8. We have not yet implemented such a heuristic for our algorithm.

**Table 8.** Blossom IV variable $\delta$ without ($B4^*_{var}$) and with ($B4^{**}_{var}$) price–and–repair heuristic vs. MST* on complete geometric instances. The Delaunay graph of the point set was chosen as the sparse subgraph for $B4^{**}_{var}$.

| $n$ | $B4^*_{var}$ | $B4^{**}_{var}$ | MST* | $t$ |
|---|---|---|---|---|
| 1000 | 37.01 | 0.43 | 24.05 | 5 |
| 2000 | 225.93 | 1.10 | 104.51 | 5 |
| 4000 | 1789.44 | 4.33 | 548.19 | 5 |

The idea is simple. A minimum–weight matching (it is now more natural to talk about minimum–weight matchings) has a natural tendency of avoiding large weight edges; this suggests to compute a minimum–weight matching iteratively. One starts with a sparse subgraph of light edges and computes an optimal matching. Once the optimal matching is computed, one checks optimality with respect to the full graph. Any edge of negative reduced cost is added to the graph and primal and dual solution are modified so as to satisfy the preconditions of the matching algorithm. Derigs and Metz [DM91, AC93, CR97] discuss the repair step in detail.

There are several natural strategies for selecting the sparse subgraph. One can, for example, take the lightest $d$ edges incident to any vertex. For complete graphs induced by a set of points in the plane and with edge weights equal to the Euclidean distance, the Delaunay diagram of the points is a good choice.

*'Worse–case' Instances for Blossom IV:* We wish to conclude the experiments with two 'worse–case' instances that demonstrate the superiority of our algorithm to Blossom IV.

The first 'worse–case' instance for Blossom IV is simply a chain. We constructed a chain having $2n$ vertices and $2n-1$ edges. The edge weights along the chain were alternately set to 0 and 2 (the edge weight of the first and last edge equal 0). $\mathrm{B4}^*_{\mathrm{var}}$ and our MST$^*$ algorithm were asked to compute a maximum–weight perfect matching. Note that the fractional matching heuristic will always compute an optimal solution on instances of this kind. Table 9 shows the results.

**Table 9.** $\mathrm{B4}^*_{\mathrm{var}}$ vs. MST$^*$ on chains.

| $2n$ | $\mathrm{B4}^*_{\mathrm{var}}$ | MST$^*$ | $t$ |
|---|---|---|---|
| 10000 | 94.75 | 0.25 | 1 |
| 20000 | 466.86 | 0.64 | 1 |
| 40000 | 2151.33 | 2.08 | 1 |

The running time of $\mathrm{B4}^*_{\mathrm{var}}$ grows more than quadratically (as a function of $n$), whereas the running time of our MST$^*$ algorithm grows about linearly with $n$. We present our argument as to why this is to be expected. First of all, the greedy heuristic will match all edges having weight 2; the two outer vertices remain unmatched. Each algorithm will then have to perform $O(n)$ dual adjustments so as to obtain the optimal matching. A dual adjustment takes time $O(n)$ for Blossom IV (each potential is explicitly updated), whereas it takes $O(1)$ for our MST$^*$ algorithm. Thus, Blossom IV will need time $O(n^2)$ for all these adjustments and, on the other hand, the time required by our MST$^*$ algorithm will be $O(n)$.

Another 'worse–case' instance for Blossom IV occurred in VLSI–Design having $n = 151780$ vertices and $m = 881317$ edges. Kindly, André Rohe made this instance available to us. We compared B4$^*$ and $\mathrm{B4}^*_{\mathrm{var}}$ to our MST algorithm. We ran our algorithm with the greedy heuristic (MST$^+$) as well as with the fractional matching heuristic (MST$^*$). The results are given in Tab. 10.

**Table 10.** B4$^*$, B4$^*_{\text{var}}$ vs. MST on `boese.edg` instance.

| $n$ | $m$ | B4$^*$ | B4$^*_{\text{var}}$ | MST$^+$ | MST$^*$ | $t$ |
|---|---|---|---|---|---|---|
| 151780 | 881317 | 200019.74 | 200810.35 | 3172.70 | 5993.61 | 1 |
| | | (332.01) | (350.18) | (5.66) | (3030.35) | |

The second row states the times that were needed by the heuristics. Observe that both Blossom IV algorithms need more than two days to compute an optimal matching, whereas our algorithm solves the same instance in less than an hour. For our MST algorithm the fractional matching heuristic did not help at all on this instance: to compute a fractional matching took almost as long as computing an optimum matching for the original graph (using the greedy heuristic).

## 5 Conclusion

We described the implementation of an $O(nm \log n)$ matching algorithm. Our implementation is competitive to the most efficient known implementation due to Cook and Rohe [CR97]. Our research rises several questions. (1) Is it possible to integrate the variable $\delta$ approach into an $O(nm \log n)$ algorithm? (2) A generator of instances forcing the implementation into their worst–case would be useful. (3) In order to handle complete graphs more efficiently the effect of a price–and–repair strategy is worth being considered; most likely, providing such a mechanism for MST$^*$ will improve its worst–case behaviour on those graphs tremendously. (4) Internally, the underlying graph is represented by the LEDA class *graph* which allows for dynamic operations like *new_node*, etc. However, dynamic operations are not required by our algorithm. The running time of some other graph algorithms in LEDA improved by a factor of about two using a static variant of the *graph* data structure (Stefan Näher: personal communication). Probably, a similar effect can be achieved for our maximum–weight matching algorithm as well.

## References


[AC93] D. Applegate and W. Cook. Solving large-scale matching problems. In D. Johnson and C.C. McGeoch, editors, *Network Flows and Matchings*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 557–576. American Mathematical Society, 1993.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullmann. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[CG] B. Cherkassky and A. Goldberg. PRF, a Maxflow Code. `www.intertrust.com/star/goldberg/index.html`.

[CR97] W. Cook and A. Rohe. Computing minimum–weight perfect matchings. Technical Report 97863, Forschungsinstitut für Diskrete Mathematik, Universität Bonn, 1997.

[Dil90] M.B. Dillencourt. Toughness and delaunay triangulations. *Discrete and Computational Geometry*, 5:575–601, 1990.

[DM86] U. Derigs and A. Metz. On the use of optimal fractional matchings for solving the (integer) matching problem. *Mathematical Programming*, 36:263–270, 1986.

[DM91] U. Derigs and A. Metz. Solving (large scale) matching problems combinatorially. *Mathematical Programming*, 50:113–122, 1991.

[Edm65a] J. Edmonds. Maximum matching and a polyhedron with (0,1) vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[Edm65b] J. Edmonds. Paths, trees, and flowers. *Canadian Journal on Mathematics*, pages 449–467, 1965.

[Gab74] H. N. Gabow. *Implementation of algorithms for maximum matching and nonbipartite graphs.* PhD thesis, Stanford University, 1974.

[Gab90] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In David Johnson, editor, *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 434–443, San Francisco, CA, USA, January 1990. SIAM.

[GMG86] Z. Galil, S. Micali, and H. N. Gabow. An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM J. Computing*, 15:120–130, 1986.

[Hum96] M. Humble. Implementierung von Flußalgorithmen mit dynamischen Bäumen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1996.

[Law76] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York, 1976.

[Meh84] K. Mehlhorn. *Data structures and algorithms. Volume 1: Sorting and searching*, volume 1 of *EATCS monographs on theoretical computer science*. Springer, 1984.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999. 1018 pages.

[Sch00] G. Schäfer. Weighted matchings in general graphs. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 2000.