Weighted Matchings in General Graphs

Diplomarbeit von Guido Schäfer

Diese Arbeit wurde unter der Betreuung von Prof. Dr. Kurt Mehlhorn am Max–Planck–Institut für Informatik in Saarbrücken innerhalb der Fachrichtung 6.2, Informatik, der Universität des Saarlandes angefertigt.

Hiermit erkläre ich an Eides Statt, daß ich diese Diplomarbeit selbständig verfaßt und nur die im Literaturverzeichnis angegebenen Quellen benutzt habe. Ferner habe ich die Arbeit noch keinem anderen Prüfungsamt vorgelegt.

Saarbrücken, 31. Mai 2000

An dieser Stelle möchte ich mich bei Prof. Dr. Kurt Mehlhorn für die Vergabe des sehr interessanten Themas und die ausgezeichnete Betreuung sowie für die Bereitstellung der hervorragenden Arbeitsbedingungen am Max–Planck–Institut für Informatik bedanken. Ferner möchte ich all jenen danken, die durch ihre Hilfe maßgeblich zum Gelingen dieser Arbeit beigetragen haben, insbesondere Jochen Könemann und Robert Spence für das Korrekturlesen. Ein großer Dank gilt meinen Freunden, die mich die ganze Zeit über und besonders in der Endphase immer wieder unterstützten. Letztlich möchte ich meinen Eltern danken, die mir das Studium ermöglichten und besonders meiner Freundin Sabine, die immer für mich da war.

Contents

Introduction

Combinatorial optimization is a field of applied mathematics and theoretical computer science. A major topic in combinatorial optimization are *linear optimization problems.* Said simply, a linear optimization problem requires the optimization of a linear function over a discrete set of solutions. An intensively studied and well–known problem in combinatorial optimization is the *weighted matching problem*: it requires the computation of a matching having maximum or minimum weight. A *matching* M in an undirected graph G is a set of edges no two of which share an endpoint. The edges of G are associated with weights and the total weight of a matching M is the sum of all the weights of the edges in M. M may further be restricted to being *perfect*, which constitutes the *weighted perfect matching problem*; a matching M is *perfect*, if every vertex in G has exactly one incident edge in M.
Many variants and extensions of the weighted matching problem exist. As an example of a variant, G might be restricted to being bipartite; this is called the *bipartite weighted matching problem.* An example of an extension, on the other hand, is the *b–matching problem*, where each vertex may have up to b incident matching edges.

There are (at least) three types of applications that motivate the investigation of weighted matching problems. (1) Direct applications of the weighted matching problem exist. (2) Many other problems can be reduced to the weighted matching problem. (3) Several algorithms (repeatedly) solve the weighted matching problem in order to progress. We will give examples of each of the three application types stated. Some of these are widely known. Additionally, we wish to present two new applications (of type (2) and (3)) that were encountered during the writing of the thesis and thus have been, for us, a major source of motivation.
A classic example of an application of type (1) is to optimize, i.e. in this case to minimize, the time spent by a plotter pen in *pen–up motion*, i.e. moving from one point to another without drawing. Reingold and Tarjan [RT81] showed this to be a weighted perfect matching problem. We briefly summarize their reasoning. Assume we wish to plot a connected figure, and assume further that the time spent by the plotter moving from one point to another is proportional to the Euclidean distance. We classify the starting and crossing points of the figure (i.e. the points where a line starts or several lines cross) to be either *odd* or *even.* A point is odd when an odd number of lines emerge, otherwise it is even. A fundamental theorem in graph theory is that there exists always an even number of odd points. Moreover, Euler proved that a figure can be traced (starting and ending in the same point) with no pen–up motion iff it is connected and no odd points exist. Thus, we need to find a new set of lines such that each odd point becomes even and, moreover, the total time of pen–up motion along

these lines is minimized. We thus define a complete graph G whose vertices correspond to the odd points of the figure and whose edge weights correspond to the Euclidean distance of these points. Minimizing the time of pen–up motion then means finding a minimum–weight perfect matching in G.
An example of type (2), which we would like to present as a motivating application for the weighted matching problem, is the so–called *dominance problem*. Its application stems from the field of computational linguistics. A dominance problem is given by a collection of vertex disjoint rooted trees and a set of *dominance wishes*. A dominance wish is a directed edge from a leaf of some tree to the root of some other tree — the leaf wishes to dominate the root. The task is to assemble the trees into a forest such that every dominance wish is satisfied, i.e. each directed edge reduces to an ancestor–predecessor relationship. Althaus et al. [ADK$^+$00] recently showed that deciding the satisfiability of a dominance problem can be reduced to a weighted matching problem.
As an example of type (3), we consider a fundamental communication problem known as *gossiping*: n processing units are required to interchange their data with each other. The underlying communication network is modeled by a graph G. A processing unit (i.e. vertex) is permitted to communicate with only one of its neighbours (i.e. adjacent vertices) at a time. The task of stating an optimal gossiping schedule, such that in the end every processing unit knows the data of all other processing units, is NP–hard. Beier and Sibeyn [BS00] use a *matching heuristic* to compute a good, sub–optimal gossiping schedule. The heuristic can be regarded as working in rounds. In each round, weights are assigned (on the basis of different criteria) to the connections (i.e. edges) of the communication network. Then, a maximum–weight matching is computed with respect to these weights. The pairs of matched processing units communicate with each other. Another well–known example of this type is Christofides' approximation algorithm for the *traveling salesman problem* (see [Chr76]). The problem is defined by a complete graph G consisting of n vertices (which represent cities), where the edge weights correspond to the Euclidean distances. The task is to find a tour of minimum length. Christofides' algorithm computes a tour whose length is at most $3/2$ as long as the length of an optimum tour; it is still the currently best known approximation algorithm for the traveling salesman problem. In a first step, the algorithm constructs a minimum spanning tree T of G, and afterwards a minimum–weight perfect matching M on the odd degree vertices of T is computed. The graph $T \cup M$ then reduces to a tour with the desired property.
Various other examples of the above–mentioned application types exist and can be found, for example, in Ball, Bodin and Dial [BBD83], Derigs and Metz [DM92], Bell [Bel94] and Ahuja, Magnanti and Orlin [AMO93].

Matching problems have been the subject of intensive research over several decades. The earliest result in matching theory we came across, widely known as König's Theorem, dates back to 1916 (see [Kön16]). One of the cornerstones in matching theory is due to Edmonds [Edm65b, Edm65a]. In 1965, he invented the famous *blossom–shrinking algorithm*, which enables a solution for the weighted matching problem to be computed in polynomial–time. A straightforward implementation, as originally proposed by Edmonds himself, requires time $O(n^2m)$, where n and m denote the number of vertices and edges in G, respectively. Since then, the theoretical running–time of the blossom–shrinking approach has been successively improved. Both Lawler [Law76] and Gabow [Gab74] improved the asymptotic running–time to $O(n^3)$. Later, Galil, Micali and Gabow [GMG86] achieved $O(nm \log n)$ and finally Gabow [Gab90] stated that Edmonds' blossom–shrinking algorithm can be implemented to run in time $O(n(m+n \log n))$. Somewhat better asymptotic time bounds can be achieved for integer edge weights using scaling algorithms (see Gabow and Tarjan [GT91]).

The currently most efficient codes implement variants of Edmonds' blossom–shrinking algorithm and are based on either the $O(n^2m)$ or $O(n^3)$ approach. For the time being, the best known implementation, named *Blossom IV*, is due to Cook and Rohe [CR97]. Their implementation is based on earlier work by Applegate and Cook [App93]. They do not claim a theoretical time bound, but, as we shall see, it cannot be better than $\Omega(n^3)$. Blossom IV is known to be highly efficient in practice; the data structures it uses are simple.
The algorithms suggested by Galil, Micali and Gabow [GMG86] and by Gabow [Gab90] mainly achieve a better asymptotic time bound by using sophisticated data structures. For example, the algorithm of Galil, Micali and Gabow requires a data structure *concatenable priority queue*, in which the priorities of certain subgroups of vertices can be uniformly changed by a single operation. Up to now, it has been an open question (and one explicitly posed in [App93] and [CR97]), whether or not the use of sophisticated data structures will help in practice. We will answer this question in the affirmative: the implementation we shall present in this thesis is based on the ideas of Galil, Micali and Gabow and turned out to be competitive — if not even superior — to Blossom IV.

The structure of the thesis is as follows. In **Chapter 1**, we will develop all details of the blossom–shrinking algorithm. We will start with the definition of some variants of the weighted matching problem and introduce important concepts, such as *augmenting paths*, that are crucial to almost all matching algorithms. The blossom–shrinking approach will first be considered for the cardinality matching case. Linear programming formulations for both the weighted matching problem and the weighted perfect matching problem will then be investigated. Duality theory will lead us towards a primal–dual method for the weighted matching problem based on Edmonds' blossom–shrinking approach. Finally, we will conclude the chapter with a brief survey of the four different realizations mentioned above.
In **Chapter 2**, we will illustrate the ideas underlying our implementation. Most of these are based on or have been developed from the ideas put forward by Galil, Micali and Gabow [GMG86]. We will outline how the blossom–shrinking approach can be implemented using priority queues. The difficulty of handling varying priorities within these priority queues will be overcome by taking advantage of the fact that these values change uniformly. Moreover, we will demonstrate in detail the need for concatenable priority queues.
In **Chapter 3** we will describe our implementation and discuss some experimental results. We implemented two versions of the algorithm: a *single search tree approach* and a *multiple search tree approach*. First, the results from Chapter 2 will be incorporated into a single search tree algorithm. Then, all necessary extensions and modifications for the multiple search tree approach will be presented. The efficiency of both algorithms is considerably improved by using a heuristic to create a better initial solution. We will discuss two heuristics: a *greedy heuristic* and a *fractional matching heuristic*. Finally, some running–time experiments will reveal the efficiency of our algorithms in practice.

Chapter 1

Matching Theory

In this chapter we will establish essential concepts that are fundamental for later discussion. We begin with the definition of the matching problem and outline some of its variants. Some useful notations such as the concept of augmenting paths will follow and lead to a first generic algorithm solving matching problems. Starting with the cardinality matching problem, we will present the main ideas of Edmonds' well–known blossom–shrinking approach. Results from the field of combinatorial optimization will guide us towards an extension of the blossom–shrinking approach for weighted matching problems.

1.1 The Matching Problem and its Variants

Let $G = (V, E)$ be an undirected graph, where V and E denote the set of vertices and edges, respectively. The number of vertices and edges are referred to by $n = |V|$ and $m = |E|$. Since G is undirected, we will denote an edge e between two vertices u and v as an unordered pair $\{u, v\}$, or uv for short. G is *bipartite* when a partition $V = A \dot{\cup} B$ of the vertices of G exists and each edge $uv \in E$ has exactly one vertex in A and one in B.

An ordered sequence $p = (e_1, e_2, \ldots, e_k)$ of edges, with $e_i = u_i u_{i+1} \in E$, $1 \leq i \leq k$, is called a *path* from u_1 to u_{k+1} in G. Alternatively, we will represent p by the sequence $p = (u_0, u_1, \ldots, u_k)$ of vertices traversed. A path p is called *simple*, when all vertices on p are distinct. Let C be a path starting and ending with the same vertex. C is then called a *cycle*. Moreover, C is said to be a *simple cycle*, when no other cycle is contained in C.

A *matching* M of G is a subset of edges such that no two edges of M share a common vertex (see Figure 1.1 for an example). All edges in M are said to be *matched* and edges in the difference $E \setminus M$ are *unmatched.* Analogously, a vertex u is said to be *matched* if there exists an incident matched edge $uv \in M$; otherwise u is *unmatched* or *free.* The adjacent vertex v of u with respect to a matched edge $e = uv$ is the *mate* of u. M is a *perfect matching* when all vertices of G are matched and hence $|M| = n/2$.

The *matching problem* is to find a matching in a graph G that meets certain require-

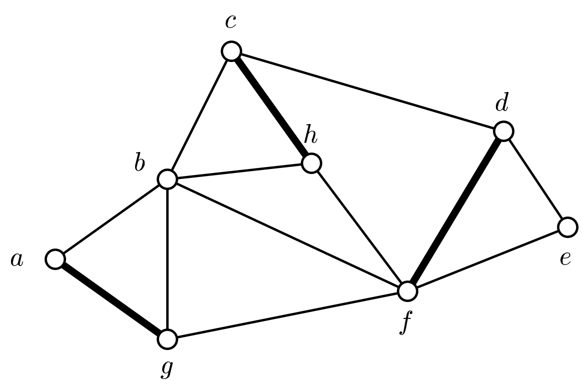

Figure 1.1: Let G be the graph depicted above. $M = \{ag, ch, df\}$ is a matching of G. $p = (e, f, d)$ is an example of an alternating path. $p' = (b, h, c, d, f, e)$ is an augmenting path. $M' = M \oplus p' = \{ag, bh, cd, fe\}$ is a matching in G with $|M'| = |M| + 1$. M' is perfect and hence a maximum–cardinality matching of G.

ments. We will distinguish between two kinds of matching problems: the unweighted and the weighted matching problem. In the weighted matching problem a weight function $w : E \longmapsto \mathbb{R}$ on the edges of G is additionally given. The distinction is further refined on the basis of whether or not G is bipartite. Altogether we classify four variants of the matching problem, which are defined below.

Maximum–Cardinality Bipartite Matching Let $G = (A \dot\cup B, E)$ be a bipartite graph. The *maximum–cardinality bipartite matching problem* is to find a matching M in G of maximum cardinality, i.e. $|M| \geq |M'|$ for any other matching M' of G.

Maximum–Cardinality Matching Consider a general graph $G = (V, E)$. In the *maximum–cardinality matching problem* a matching M of maximum cardinality has to be determined.

In both cardinality cases, M need not necessarily be perfect. However, every perfect matching of G forms a maximum–cardinality matching.

Maximum–Weight Bipartite Matching Let $G = (A \dot\cup B, E, w)$ be a bipartite graph with weight function w. Finding a matching M with total weight $w(M) = \sum_{e \in M} w(e)$ and $w(M) \geq w(M')$ for all other matchings M' of G constitutes the *maximum–weight bipartite matching problem.*

In the *maximum–weight bipartite perfect matching problem* M is further restricted to being perfect. This problem is also known as the *maximum–weight assignment problem.*

Maximum–Weight Matching The most general case of all matching problems is the *maximum–weight matching problem.* Given a general graph $G = (V, E, w)$ with weight function w, the task is to find a matching M having maximum weight $w(M)$ among all possible matchings of G.

As above, one might wish to obtain a perfect matching of maximum weight. This constitutes the *maximum–weight perfect matching problem.*

Let $G = (V, E, w)$ be an instance of a weighted matching problem. One might wish to obtain a matching of minimum instead of maximum weight in G. However, each minimum–weight matching problem can be reduced to an appropriate maximum–weight matching problem by negating the signs of all weights. That is, a maximum–weight matching M of $G' = (V, E, -w)$ will be a mimimum–weight matching in G.

Many other variants and extensions of the matching problem exist; for example f–factors, b–matchings, T–joins, etc. However, in the context of this thesis, we will only focus on the four variants defined above. For extensive sources concerning all aspects of matching problems, see, for example, Lovász and Plummer [LP86] and Pulleyblank [Pul95].

1.2 Matching Concepts

Two concepts are crucial to all matching algorithms: alternating paths and augmenting paths. The importance of both will become clear shortly. Throughout this section let $G = (V, E)$ be a graph that might or might not be bipartite. All results apply to both cases unless stated otherwise.

Definition 1.2.1 (Alternating Path) Let $p = (e_1, e_2, \ldots, e_k)$ be a simple path from u to v and M a matching in G. p is an *alternating path* with respect to M, when the edges along p are alternately in M and not in M.

An alternating path $p = (e_1, \ldots, e_k)$ with respect to M, where both endpoints u and v are free, can be used to augment the current matching M. To see this, consider the symmetric difference M' of M and p: $M' = M \oplus p = (M \setminus p) \cup (p \setminus M)$. M' equals M except that all matching edges with respect to M on p are unmatched in M' and all non–matching edges with respect to M on p are matched in M'. It can easily be seen that M' itself forms a matching.[1] Moreover, $|M'| = |M| + 1$ and thus M has indeed been augmented. We will say *M has been augmented by p to M'* and p is called an *augmenting path.* See Figure 1.1 for an example.

Definition 1.2.2 (Augmenting Path) An alternating path $p = (e_1, \ldots, e_k)$ with respect to a matching M is called *augmenting* when both endpoints of p are free.

The discussion above gives rise to the idea that we can compute a maximum–cardinality matching by repeatedly seeking an augmenting path p to a current matching M. When p exists, M is augmented by p and we proceed with the augmented matching $M \oplus p$.

[1]Each vertex that is matched in M is also matched in M'. Only u and v are additionally matched in M'. But u and v were free in M and thus M' is a matching.

Otherwise, M is claimed to be maximum.
The following lemma states that the latter conclusion does in fact hold. It is due to Berge [Ber57].

Lemma 1.2.1 M is a matching of maximum cardinality iff there does not exist an augmenting path with respect to M in G.

Proof:
Clearly, if there exists an augmenting path p with respect to M, then $M' = M \oplus p$ is a matching having cardinality $|M'| = |M| + 1$. Thus, M is not a maximum–cardinality matching.

Assume that M is not a maximum–cardinality matching, i.e. there exists a matching M' with $|M'| > |M|$. We show that an augmenting path p with respect to M must exist.
Consider the graph $\widetilde{G}$ containing the edges $M \oplus M'$ only. Each vertex in $\widetilde{G}$ has either degree zero, one or two. Therefore, $\widetilde{G}$ consists of isolated vertices, paths and cycles. Since M and M' are matchings, the edges on every path and cycle are alternately in M and in M'. All cycles must be of even length having as many edges in M as in M'. Since $|M'| > |M|$, there must be at least one path, say p, in $\widetilde{G}$ having more edges out of M' than of M. The first and last edge of p must be in M' and hence p is an augmenting path with respect to M. $\square$

Using Lemma 1.2.1 we state a first generic algorithm to compute a maximum–cardinality matching:

Algorithm 1.2.1 Generic algorithm for maximum–cardinality matching problems.

```
let $M$ be any matching
while there exists an augmenting path $p$ with respect to $M$
    replace $M$ by the augmented matching $M \oplus p$
```

Observe that Algorithm 1.2.1 can be refined to search for an augmenting path from each free vertex exactly once.

We show that if no augmenting path starting in a free vertex r with respect to a matching M exists, then there will never exist an augmenting path starting in r with respect to any other matching M' obtained from M by a series of augmentations: $M' = ((M \oplus p_0) \oplus p_1) \oplus \ldots$. Suppose p' is an augmenting path starting in r with respect to a matching M' and no augmenting path starting in r with respect to M exists. Let $e = uv$ denote the first edge in p' with $e \in M'$ but $e \notin M$. One endpoint, say u, is reachable from r by an alternating path with respect to M. The non–existence of any augmenting path from r with respect to M implies, that no alternating path from u with respect to M starting with a matched edge to any other free vertex exists. However, this is a contradiction, since e can in this case never be matched.

In the rest of this section, a search strategy for finding an augmenting path in a bipartite graph G will be considered closely. The difficulties arising for the general case are then indicated; they will be solved in Section 1.3.

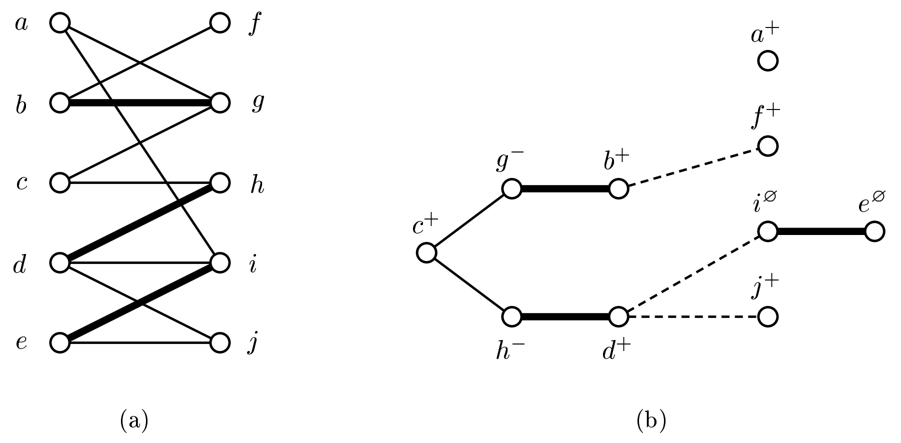

Figure 1.2: Let $G = (A\dot{\cup}B, E)$ be the graph given in (a). Edges in M are drawn bold. A possible alternating tree T rooted at the free vertex c is depicted in (b). In the next step, T can either be enlarged by taking the edges di and ie to T, or one of the two augmenting paths $p = (f, b, g, c)$ and $p' = (j, d, h, c)$ will be found.

Let $G = (A\dot{\cup}B, E)$ be a bipartite graph and M an arbitrary matching. The search starts from a free vertex r of G and terminates either when an augmenting path p to another free vertex has been found, or there does not exist an augmenting path starting in r.

A tree T is grown from r such that each path from a vertex u in T to the root r is alternating with respect to M. The vertices of T are labeled either *even* or *odd*, stating that the alternating path to the root is of even or odd length. T is called the *alternating tree*. Matched vertices that do not belong to T are said to be *unlabeled*. All free vertices are initially labeled even. For short, we denote an even, odd or unlabeled vertex v by v^+, v^- or $v^\varnothing$, respectively. In cases where a vertex label is, for example, either unlabeled or labeled even we use notions like $v^{\{\varnothing|+\}}$ etc.

Initially, T consists of the even vertex r^+ only. The alternating tree is grown from even vertices $u^+ \in T$.
Let $v^\varnothing \notin T$ be adjacent to any vertex $u^+ \in T$. T is extended by taking the unmatched edge uv and also the matching edge of v to T, i.e. the edge vw, where $w^\varnothing \notin T$ is the mate of v. Here, v and w get labeled odd and even, respectively.
When an even vertex $v^+ \notin T$ is adjacent to any vertex $u^+ \in T$, an augmenting path $p = (v, u, \dots, r)$ with respect to M has been found.
If at some stage the tree cannot be grown and no adjacent free vertex exists, the search terminates due to the non–existence of an augmenting path beginning in r.

A possible example scenario for an alternating tree T in a bipartite graph can be seen in Figure 1.2.

Let us try to apply the described search to the general graph G illustrated in Figure 1.3(a). Clearly, the path $p = (g, c, d, e, f, b, a, r)$ is augmenting. However, when an

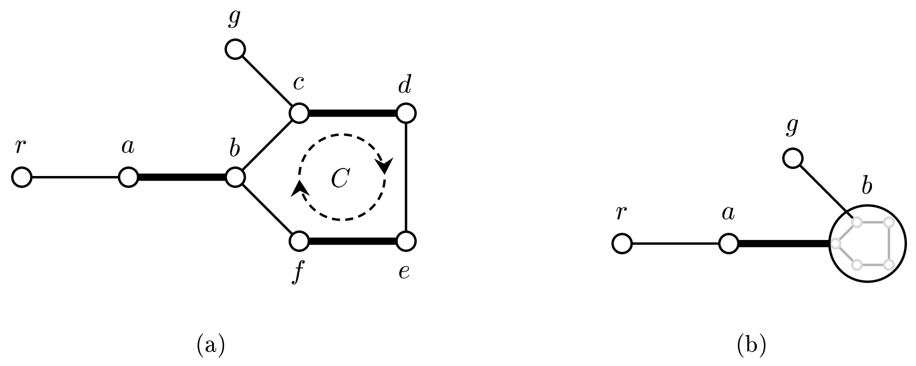

Figure 1.3: Let G and M be as given in (a). $C = (b, c, d, e, f, b)$ is an odd length cycle. By definition, $\mathcal{B} = \{b, c, d, e, f\}$ forms a blossom. b is the base of $\mathcal{B}$. For every vertex $u \in \mathcal{B}$ an even length alternating path to the base exists. For example, $p = (c, d, e, f, b)$ is the corresponding path for c. The graph $G' = (V', E')$ obtained from G by shrinking the blossom $\mathcal{B}$ is shown in (b). It is $V' = \{r, a, b, g\}$ and $E' = \{ra, ab, gb\}$.

alternating tree is grown from r, p could be missed when c is labeled odd. It is due to the existence of odd length cycles that augmenting paths are missed. Since odd length cycles cannot occur in a bipartite graph it becomes also perspicuous why the current search strategy operates correctly in the bipartite case only.

Edmonds was the first to circumvent this problem; he did so by using the concept of blossoms, which will be the subject of the next section.

1.3 Edmonds' Blossom–Shrinking Approach

In 1965, Edmonds extended the search described in the preceding section to the general case (see [Edm65b]). The resulting algorithm is widely known as the *blossom–shrinking approach* and will be the subject of this section.

We first establish a general basis by introducing the blossom concept and the idea of shrinking. Thereafter, a different interpretation of those concepts, which will be more appropriate for the weighted matching case, is shown to be equivalent. Based on that alternative interpretation, the search for an augmenting path in a general graph is revised at the end of this section.

Let $G = (V, E)$ be a general graph. The following two notations will be helpful. For any subset $S \subseteq V$ we denote the edges of G having both endpoints in S by $\gamma(S)$:

$$\gamma(S) = \{uv \in E \ : \ u \in S \text{ and } v \in S\}.$$

Conversely, we define $\delta(S)$ as the set of all edges having exactly one endpoint in S:

$$\delta(S) = \{uv \in E \ : \ u \in S \text{ and } v \notin S\}.$$

Note that $\delta(\{v\})$ denotes all edges incident to a vertex v. In that case, we will write $\delta(v)$ for short.

As mentioned above, it is due to the existence of an odd length cycle that our current search might miss an augmenting path. Assume C denotes such an odd length cycle and, moreover, let C contain a maximum number of matching edges. This concept is what we call a *blossom.*

Definition 1.3.1 (Blossom) Let M be a matching in G and $\mathcal{B} \subseteq V$ an odd cardinality subset of vertices. $\mathcal{B}$ is a blossom, when $\gamma(\mathcal{B})$ contains a simple cycle C that traverses all vertices of $\mathcal{B}$, and, moreover, a maximum number of edges along C are matched, i.e. $|M \cap C| = \lfloor |\mathcal{B}|/2 \rfloor$.

Figure 1.3(a) shows an example of a blossom. The only vertex in a blossom $\mathcal{B}$ that is either free, or whose matching edge is not contained in $\gamma(\mathcal{B})$, is called the *base* of $\mathcal{B}$. $\mathcal{B}$ is *free*, when its base is free; otherwise, $\mathcal{B}$ is *matched.*

Our interest in the blossom concept stems from the following fact. Consider a blossom $\mathcal{B}$ with base b. For any arbitrary vertex u of $\mathcal{B}$ an even length alternating path p from u to the base b must exist. Moreover, the first edge of p is a matching edge and p lies exclusively in $\mathcal{B}$, i.e. $e \in \gamma(\mathcal{B})$ for each edge e in p. Edmonds observed that one can benefit from that property by *shrinking* the blossom $\mathcal{B}$ into a single vertex, for example into b. Informally, this means that all vertices of $\mathcal{B}$ are collapsed into b and all edges in $\gamma(\mathcal{B})$ become non–existent. Let G' denote the graph obtained from G by shrinking the blossom $\mathcal{B}$ (see Figure 1.3(b)). Formally, $G' = (V', E')$ can be defined as follows.

$$V' = (V \setminus \mathcal{B}) \cup \{b\}$$

and

$$E' = \gamma(V \setminus \mathcal{B}) \cup \{ub \; : \; uv \in \delta(\mathcal{B}) \text{ and } u \notin \mathcal{B}\}.$$

Let M' denote the matching in G' that corresponds to M, i.e. $M' = M \setminus \gamma(\mathcal{B})$. The intention behind shrinking is that any augmenting path p' with respect to M' in G' can be *lifted* (as described in the proof below) to an augmenting path p with respect to M in G.

Lemma 1.3.1 Let G' be a graph obtained from G by shrinking a blossom $\mathcal{B}$ as described above. If an augmenting path p' with respect to M' in G' exists, then there also exists an augmenting path p with respect to M in G.

Proof:
Let p' be an augmenting path in G'. We consider only the case where p' traverses b, since otherwise p' reduces to an augmenting path in G. We can break p' at b into p_1 and p_2: $p' = (p_1, b, p_2)$. Let p_2 be the path that starts with the non–matching edge bv. When b is an endpoint of p' and hence must be free, p_1 is empty. Otherwise, p_1 ends with the matched edge ub. Due to the construction of G', there must be a vertex $w \in \mathcal{B}$ such that wv is an edge in G. Moreover, we know there must exist a possibly empty even length alternating path in $\gamma(\mathcal{B})$ from w to b. Let $p_{\mathcal{B}}$ denote that path in reversed order, i.e. leading from b to w in G. The augmenting path p in G then consists simply of the concatenation p_1, $p_{\mathcal{B}}$ and p_2, where the first edge bv of p_2 is replaced by wv. $\square$

We will soon refine the search strategy of Section 1.2 such that it will work for general graphs. But first, we wish to argue that each graph $G^{(i)}$ obtained from G by a series of shrinkings can be viewed as a *nested family of odd cardinality subsets of* V. Let us introduce that notion next:

$\mathcal{N}(V)$ is a nested family of odd cardinality subsets of V, when

(1) each element S of $\mathcal{N}(V)$ is a subset of V having odd cardinality, and

(2) for two elements $S_i, S_j \in \mathcal{N}(V)$ with $S_i \neq S_j$, either $S_i \subset S_j$, or $S_j \subset S_i$, or $S_j \cap S_i = \emptyset$ holds.

Assume $G^{(i)}$ is obtained from G as given below.

$$G = G^{(0)} \xrightarrow{\textit{shrink } \mathcal{B}_0} G^{(1)} \xrightarrow{\textit{shrink } \mathcal{B}_1} \cdots \xrightarrow{\textit{shrink } \mathcal{B}_{i-1}} G^{(i)}$$

Let $V^{(i)}$ denote the set of vertices in $G^{(i)}$. Each vertex $v \in V^{(i)}$ corresponds to an odd cardinality set $S_v^{(i)} \subseteq V$ which can be defined recursively. We have $S_v^{(0)} = \{v\}$ and for $i > 0$:

$$S_v^{(i)} = \begin{cases} S_v^{(i-1)} & \text{when } v \notin \mathcal{B}_{i-1}, \\ \bigcup\limits_{u \in \mathcal{B}_{i-1}} S_u^{(i-1)} & \text{otherwise.} \end{cases}$$

Note that uniting an odd number of odd cardinality sets will result in an odd cardinality set. Therefore, each $S_v^{(i)}$ is indeed of odd cardinality. Moreover, observe that a maximum number $\lfloor |S_v^{(i)}|/2 \rfloor$ of edges in $\gamma(S_v^{(i)})$ are matched; this can easily be shown by induction on i.

From the definition of $S_v^{(i)}$ it follows that

$$\mathcal{N}(V) = \bigcup_{j=0}^{i} \left(\bigcup_{v \in V^{(j)}} S_v^{(j)} \right)$$

is a nested family of odd cardinality subsets of V.

$\mathcal{N}(V)$ provides sufficient structural information about the nesting of blossoms. The nesting of blossoms will be of major importance in the weighted matching case later on. Therefore, we redefine — or better, reinterpret — the concept of blossoms and introduce some additional terms based on the view we are about to develop.

Each element $\mathcal{B} \in \mathcal{N}(V)$ is called a blossom of G.[2] Moreover, we distinguish between *trivial* and *non–trivial* blossoms. A trivial blossom $\mathcal{B} = \{v\}$ corresponds to the vertex v in G. All non–singleton sets $\mathcal{B} \in \mathcal{N}(V)$ are non–trivial blossoms; they contain other blossoms which we call *subblossoms*: $\mathcal{B}_i$ is a subblossom of $\mathcal{B}$ if $\mathcal{B}_i \subset \mathcal{B}$.
A maximum superset $\mathcal{B} \in \mathcal{N}(V)$, i.e. $\mathcal{B} \not\subset S$ for all sets $S \in \mathcal{N}(V)$, is what we call a *surface blossom*. Obviously, surface blossoms are not contained in other blossoms. Notice, that each vertex in $G^{(i)}$ corresponds to a surface blossom in $\mathcal{N}(V)$.

[2] We wish to emphasize that $\mathcal{B}$ does not form a blossom in the sense of Definition 1.3.1: the simple cycle C containing all vertices of $\mathcal{B}$ does not necessarily have to exist. But it is assured, however, that an even length path from each vertex $v \in \mathcal{B}$ to the base vertex exists.

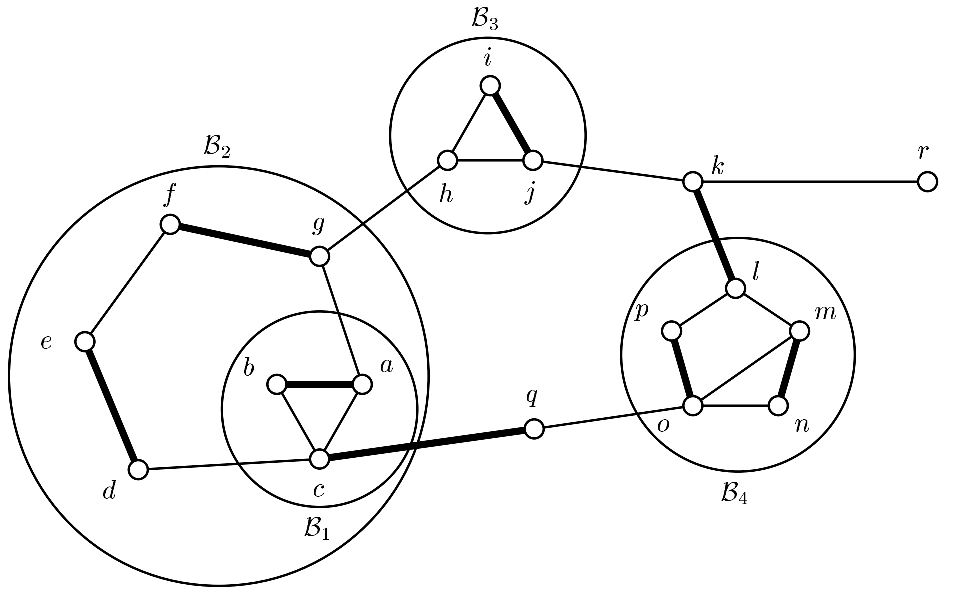

Figure 1.4: Example of a graph G after a series of shrinkings. There are four non–trivial blossoms: $\mathcal{B}_1 = \{a, b, c\}$, $\mathcal{B}_2 = \{\mathcal{B}_1, d, e, f, g\}$, $\mathcal{B}_3 = \{h, i, j\}$ and $\mathcal{B}_4 = \{l, m, n, o, p\}$. The nested family of odd cardinality subsets of V equals $\mathcal{N}(V) = \{\{a\}, \{b\}, \ldots, \{r\}, \mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$. $\mathcal{B}_1$ is an immediate subblossom of $\mathcal{B}_2$; the trivial blossom $\{a\}$ is a subblossom, but not an immediate subblossom of $\mathcal{B}_2$. The base of $\mathcal{B}_2$ and $\mathcal{B}_1$ is c. Current surface blossoms are $\mathcal{B}_2, \mathcal{B}_3, \{k\}, \mathcal{B}_4, \{q\}$ and $\{r\}$, of which the first five form a new free blossom with base h.

All edges e in G are classified as either *dead* or *alive.* An edge e is dead, when it lies in a blossom $\mathcal{B}$, i.e. $e \in \gamma(\mathcal{B})$; all other edges are alive. Thus, after a series of shrinkings the current graph G is viewed as being partitioned into surface blossoms which are connected by alive edges only. Therefore, G will also be called the *surface graph.*
Let $p = (e_1, e_2, \ldots, e_k)$ be an ordered sequence of alive edges of G. We say p is a *(surface) path* from $\mathcal{B}_1$ to $\mathcal{B}_{k+1}$ in G, when $e_i \in \delta(\mathcal{B}_i) \cap \delta(\mathcal{B}_{i+1})$ for $1 \le i \le k$. p is *simple*, when additionally all blossoms $\mathcal{B}_i$, $1 \le i \le k+1$, on p are distinct. The definitions for alternating and augmenting paths extend to surface paths in the obvious way. A *(surface) cycle* $C = (e_1, e_2, \ldots, e_k)$ in G is a path from a blossom $\mathcal{B}_1$ to itself. C is *simple*, when no other cycle is contained in C.

Suppose $C = (e_1, e_2, \ldots, e_{2k+1})$ is a simple surface cycle of odd length in G. Let $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{2k+1}$ denote the odd number of surface blossoms that lie on C. Moreover, let C contain k matching edges with respect to a matching M in G. Then, a new blossom

$$\mathcal{B} = \bigcup_{i=1}^{2k+1} \mathcal{B}_i$$

has been found. We can shrink $\mathcal{B}$ by adding it to $\mathcal{N}(V)$. Consequently, all blossoms $\mathcal{B}_i$, $1 \le i \le 2k+1$, stop being surface blossoms and become subblossoms of $\mathcal{B}$. $\mathcal{B}$ is a new surface blossom of G. The defining blossoms $\mathcal{B}_i$, $1 \le i \le 2k+1$, of $\mathcal{B}$ will be called *immediate subblossoms* of $\mathcal{B}$. Figure 1.4 shows an example scenario.

Algorithm 1.3.1 Generic algorithm to search for an augmenting path p from a free vertex r. Let G be the underlying graph and M a matching in G such that r is free.

```
let $r$ be the only even vertex of $T$
while there does not exist an alive edge $e = uv$ with $u^+ \notin T$ and $v^+ \in T$ {
    if an alive edge $uv$ with $u^+ \in T$ and $v^\varnothing \notin T$ exists {
        let $b$ be the base of $\mathcal{B}_v$ and $w$ denote the mate of $b$, with $w \in \mathcal{B}_w$
        make $\mathcal{B}_v$ an odd and $\mathcal{B}_w$ an even labeled blossom of $T$
        add the edges $uv$ and $bw$ to $T$
    }
    else if an alive edge $uv$ with $u^+ \in T$ and $v^+ \in T$ exists {
        determine the lowest common ancestor $\mathcal{B}_{lca}$ of $\mathcal{B}_u$ and $\mathcal{B}_v$ in $T$
        let $p_1 = (e_1, \ldots, e_{2j})$ be the alternating path from $\mathcal{B}_{lca}$ to $\mathcal{B}_u$ in $T$, and
        let $p_2 = (e_{2j+2}, \ldots, e_{2k+1})$ be the alternating path from $\mathcal{B}_v$ to $\mathcal{B}_{lca}$ in $T$
        all surface blossoms on $C = (p_1, e_{2j+1} = uv, p_2)$ define a new blossom $\mathcal{B}$
        shrink $\mathcal{B}$ by making all surface blossoms on $C$ to subblossoms of $\mathcal{B}$
        $\mathcal{B}$ gets labeled even and all edges in $\gamma(\mathcal{B})$ are considered to be dead
    }
    else terminate, $T$ is abandoned since no augmenting path for $r$ exists
}
there must exist an even length alternating surface path $p''$ from $\mathcal{B}_v$ to $\mathcal{B}_r$ in $T$
$p' = (e, p'')$ is an augmenting surface path from $\mathcal{B}_u$ to $\mathcal{B}_r$
raise $p'$ to an augmenting path $p$ in the original graph $G$ using Lemma 1.3.1
```

By now we are well prepared to revise our search for an augmenting path. At the end we give a generic algorithm that seeks an augmenting path in a general graph G. The algorithm is based on the nested view of G developed above and will be fundamental for the weighted matching problem.

Let M be a matching in G and r a free vertex with respect to M. As in the bipartite case, an alternating tree T is grown from r. However, T forms a tree with respect to the surface blossoms of G only, and the edges used by the search are restricted to being alive. For the sake of conciseness, we denote the surface blossom to a vertex u of G by $\mathcal{B}_u$. Moreover, we stipulate that each vertex u retains the label of its surface blossom $\mathcal{B}_u$, and u is said to be in T, when $\mathcal{B}_u$ is contained in T.

Shortly, it will become apparent that non–trivial blossoms can occur only as even tree blossoms in the unweighted matching case. However, in the weighted matching case later on, non–trivial blossoms will also occur outside of T and can be even or odd tree blossoms. Therefore, we do some preparatory work by assuming non–trivial blossoms to be of any kind.

Initially, T consists of the even labeled vertex r^+ only. The search assumes the following labeling for all surface blossoms outside of T: each free surface blossom is labeled even and each matched surface blossom is unlabeled. Four cases have to be distinguished.

Let uv be an alive edge with $u^+ \in T$ and $v^\varnothing \notin T$. The base b of $\mathcal{B}_v$ must be matched, since $\mathcal{B}_v$ is unlabeled. Let w denote the mate of b in $\mathcal{B}_w$. T is extended by making $\mathcal{B}_v$

Algorithm 1.3.2 Generic algorithm to compute a maximum–cardinality matching in a general graph G.

```
let $M$ be an arbitrary matching in $G$
label all free vertices even and unlabel all matched vertices
for each vertex $r$ in $G$ {
    if $r$ is matched continue with another vertex
    grow an alternating tree $T$ rooted in $r$ as described in Algorithm 1.3.1
    if an augmenting path $p$ with respect to $M$ in $G$ has been found {
        replace $M$ by the augmented matching $M \oplus p$
        unlabel all vertices contained in $T$
        delete all non–trivial surface blossoms of $T$
        destroy $T$
    }
    else $T$ has been abandoned
        continue with another vertex
}
$M$ is a maximum–cardinality matching
```

an odd and $\mathcal{B}_w$ an even labeled tree blossom and taking uv and bw to T. This is what we will call a *grow step* henceforth.

Let us assume there exists an alive edge uv with $u^+ \in T$ and $v^+ \in T$. We determine the *lowest common ancestor surface blossom* $\mathcal{B}_{lca}$ of $\mathcal{B}_u$ and $\mathcal{B}_v$. That is, $\mathcal{B}_{lca}$ is the first blossom that is both on the surface tree path from $\mathcal{B}_u$ to $\mathcal{B}_r$ and on the surface tree path from $\mathcal{B}_v$ to $\mathcal{B}_r$. Notice that from the way we built T, $\mathcal{B}_{lca}$ must be labeled even. Let $p_1 = (e_1, \ldots, e_{2j})$ denote the even length surface path from $\mathcal{B}_{lca}$ to $\mathcal{B}_u$ and $p_2 = (e_{2j+2}, \ldots, e_{2k+1})$ the even length surface path from $\mathcal{B}_v$ to $\mathcal{B}_{lca}$ in T. Obviously, $C = (p_1, e_{2j+1} = uv, p_2)$ is an odd length surface cycle and moreover, a maximum number k of edges on that cycle are matched, i.e. we have detected a blossom $\mathcal{B}$. $\mathcal{B}$ is defined as the union of all surface blossoms $\mathcal{B}_i$ on C, with $1 \le i \le 2k+1$. Since for every vertex v of $\mathcal{B}$ an even length alternating path to the base of $\mathcal{B}$ (this will actually be the base of $\mathcal{B}_{lca}$) exists, and therefore also an even length alternating path from v to the root r of T, $\mathcal{B}$ gets labeled even.[3] All blossoms $\mathcal{B}_i$, $1 \le i \le 2k+1$, become subblossoms of $\mathcal{B}$ and each edge in $\gamma(\mathcal{B})$ is no longer used by the search. That completes the description of a so–called *shrink step*.

When an alive edge uv with $u^+ \in T$ and $v^+ \notin T$ is encountered, an augmenting surface path $p' = (vu, p'')$ from $\mathcal{B}_v$ to $\mathcal{B}_r$ is directly available. Here, p'' denotes the even length alternating surface path from $\mathcal{B}_u$ to $\mathcal{B}_r$ in T. p' can be lifted to an augmenting path p in the original graph G by repeatedly applying Lemma 1.3.1.

Last, when none of the above cases applies T is *abandoned*, since no augmenting path starting in r exists. T retains its identity, i.e. all surface blossoms in T stay in T and retain their label. T will never be looked at again.
When an alternating tree T is abandoned, there are no edges from any vertex $u^+ \in T$

[3] Actually, that is the justification for the label of a vertex being determined by its surface blossom.

to any other vertex $v^{\{\varnothing|+\}} \notin T$. Moreover, each edge uv connecting two even vertices $u^+ \in T$ and $v^+ \in T$ is dead, i.e. lies in a surface blossom $\mathcal{B}^+ \in T$. Each odd surface blossom $\mathcal{B}_i^- \in T$ (which is trivial in the unweighted matching case) is matched by an alive edge $ij \in M$ with an even surface blossom $\mathcal{B}_j^+ \in T$, and $\mathcal{B}_r^+ \in T$ is the only surface blossom that is free in T.

The complete search for an augmenting path in a general graph G is summarized in Algorithm 1.3.1.

Combining the idea of Algorithm 1.2.1 with the search just described yields a generic algorithm for computing a maximum–cardinality matching in a general graph G as given in Algorithm 1.3.2.

In the rest of this section, we will prove optimality of M^*, the matching obtained by Algorithm 1.3.2, and thus establish correctness. The results to come are interesting from a theoretical point of view. However, the optimality criteria for the weighted matching case will be of another kind and only Algorithm 1.3.1 will be used. Therefore, the reader may also skip directly to the next section.

Different optimality criteria have evolved over several decades. Two of them will be considered more closely. The first is due to Edmonds [Edm65b] and is based on the notion of an *odd set cover*. The second is known as the Tutte–Berge Formula.

Assume M^* leaves t vertices unmatched. The cardinality of M is thus $\lfloor (n-t)/2 \rfloor$, where n denotes the number of vertices in G. For each free vertex r_i, $1 \le i \le t$, an alternating tree T_i, which has been abandoned by the search, is rooted in $\mathcal{B}_{r_i}$. As we outlined above, each vertex $u^- \in T_i$, $1 \le i \le t$, is matched with a surface blossom $\mathcal{B}^+ \in T_i$ and only the root blossom $\mathcal{B}_{r_i}$ is free. Remember that all edges uv connecting two even vertices must lie in the same blossom $\mathcal{B}^+ \in T_i$ for some $1 \le i \le t$. All unlabeled vertices $u^\varnothing$ are matched with a vertex $v^\varnothing$ and for each tree T_i, there exists no edge uv with $u^\varnothing$ and $v^+ \in T_i$.

Let $\mathcal{C}(V)$ be a family of pairwise disjoint odd cardinality subsets of V. $\mathcal{C}(V)$ is called an *odd set cover* of G when for every edge $e \in E$: $e \in \delta(v)$ for a singleton set $\{v\} \in \mathcal{C}(V)$, or otherwise $e \in \gamma(S)$ for a non–singleton set $S \in \mathcal{C}(V)$.

The *capacity cap*(S) of a set $S \in \mathcal{C}(V)$ is defined as

$$cap(S) = \begin{cases} 1 & \text{when } S \text{ is a singleton set,} \\ \lfloor |S|/2 \rfloor & \text{otherwise.} \end{cases}$$

As can easily be verified, the total capacity $cap(\mathcal{C}(V)) = \sum_{S \in \mathcal{C}(V)} cap(S)$ of an odd set cover gives an upper bound for the cardinality of any matching in G, i.e. $|M| \le cap(\mathcal{C}(V))$.[4]

Edmonds constructed an odd set cover $\mathcal{C}(V)$ of G having capacity equal to the cardinality of M^* and thus proved M^* to be maximum.

$$\mathcal{C}(V) = \{v^- \in T_i \;:\; 1 \le i \le k\} \;\cup\; \{\mathcal{B}^+ \in T_i \;:\; 1 \le i \le k, \text{ and } \mathcal{B} \text{ is non–trivial}\}.$$

When $U \neq \emptyset$, we choose some $\hat{u} \in U$ and add $\{\hat{u}\}$ to $\mathcal{C}(V)$. Additionally, $U \setminus \hat{u}$ is added to $\mathcal{C}(V)$, when $|U| > 2$.[5]

[4]Let M be a matching in G. Each edge $e \in M$ must be covered by some set $S \in \mathcal{C}(V)$ and the number of matching edges covered by some $S \in \mathcal{C}(V)$ is clearly bounded above by $cap(S)$.

[5]Let us see why $\mathcal{C}(V)$ does indeed form an odd set cover. Each odd vertex $v^- \in T_i$, $1 \le i \le t$,

Each odd vertex v covers exactly $1 = cap(v)$ matching edge of M^*. We argued above that the number of matched edges in an even surface blossom $\mathcal{B}$ equals $\lfloor |\mathcal{B}|/2 \rfloor = cap(\mathcal{B})$. Finally, $\hat{u}$ covers exactly $1 = cap(\hat{u})$ matching edge. If $|U| > 2$, all other $\lfloor (|U|-1)/2 \rfloor = cap(U \setminus \hat{u})$ matching edges are covered by $U \setminus \hat{u}$. Thus, we have $|M^*| = cap(\mathcal{C}(V))$ as desired. We can now state the optimality criteria which is due to Edmonds [Edm65b].

Lemma 1.3.2 Let $G = (V, E)$ be a graph and M a matching in G. Moreover, let $\mathcal{C}(V)$ be an odd set cover of G having capacity $cap(\mathcal{C}(V))$. Then, M is a maximum–cardinality matching and $\mathcal{C}(V)$ is an odd set cover having minimum capacity, iff $|M| = cap(\mathcal{C}(V))$.

Another interesting possibility to obtain an upper bound on the cardinality of a matching M in G is as follows.
Let $A \subseteq V$ be an arbitrary subset of vertices of G. Removing each vertex $u \in A$ and all its incident edges from G, results in a new graph denoted by $G \setminus A$. Let $C_1, C_2, \ldots, C_k$ be the connected components in $G \setminus A$ having an odd number of vertices. Each C_i contains either a free vertex, or there exists a matching edge $uv \in M$ with $u \in C_i$ and $v \in A$. Since M is a matching, the endpoints in A of those edges must be distinct. Therefore, at most $|A|$ such matching edges exist. Consequently, we can conclude that at least $k - |A|$ vertices must be free with respect to M. To put it differently, no more than $n - (k - |A|)$ vertices can be matched by M.

Let $occ(G)$ denote the number of connected components in G having an odd number of vertices. The cardinality of a matching M is thus bounded by $|M| \leq \lfloor (n - occ(G \setminus A) + |A|)/2 \rfloor$, for any $A \subseteq V$.

Again, we show optimality of M^*. Choose $A = \{v^- \in T_i : 1 \leq i \leq k\}$. Obviously, $occ(G \setminus A)$ must be $|A| + t$, since that is the total number of even surface blossoms in all abandoned trees T_i, $1 \leq i \leq k$. Thus, the bound stated above becomes tight, i.e. $|M^*| = \lfloor (n-t)/2 \rfloor = \lfloor (n - occ(G \setminus A) + |A|)/2 \rfloor$, and M^* is maximum. The following optimality criterion for a maximum–cardinality matching has just been proved. It is due to Berge [Ber58].

Lemma 1.3.3 Let $G = (V, E)$ be a graph having n vertices and M a matching in G. M is a maximum–cardinality matching, iff a set $A \subseteq V$ exists with $|M| = \lfloor (n - occ(G \setminus A) + |A|)/2 \rfloor$.

The discussion above and Lemma 1.3.3 immediately imply the following corollary which states a condition for the existence of a perfect matching. It was originally proved by Tutte [Tut47].

Corollary 1.3.1 A graph $G = (V, E)$ has a perfect matching iff for every set $A \subseteq V$ of vertices $occ(G \setminus A) \leq |A|$.

As an aside, observe that Algorithm 1.3.2 will find a perfect matching, if there exists any. But it can even prove the non–existence of a perfect matching using Corrollary 1.3.1. To see this, consider any abandoned tree T_i. Let A denote the set of odd vertices in T_i. Since the number of even labeled surface blossoms in T_i equals $|A| + 1$, it is $occ(G \setminus A) = |A| + 1 > |A|$ and we have thus proved that no perfect matching exists. In conclusion, we can state that Algorithm 1.3.2 can solve maximum–cardinality perfect matching problems as well.

covers all its incident edges. All edges lying in an even labeled surface blossom $\mathcal{B}^+ \in T_i$ are covered by $\mathcal{B} \in \mathcal{C}(V)$. Edges connecting two vertices of U are covered by $\hat{u}$ or lie in $\gamma(U \setminus \hat{u})$ and are hence covered by $U \setminus \hat{u}$. Finally, no other edges exist as stated before.

1.4 LP Formulations for Weighted Matching Problems

In the preceding sections, important matching concepts such as augmenting paths have been introduced. Further, we acquired a generic algorithm that can solve both variants of the maximum–cardinality matching problem. The stated results serve as a good basis for the weighted case considered in this and the subsequent sections.

Fundamental findings in the area of combinatorial optimization will guide us to a generic algorithm for the weighted matching problem. We assume familiarity with terms such as linear programming formulations, relaxation, duality theory (weak and strong duality, complementary slackness) as well as the concepts behind primal–dual methods. For extensive sources concerning these subjects, see Bertsimas and Tsitsiklis [BT97], Papadimitriou and Steiglitz [PS82] and Chvátal [Chv83].

We start with the discussion of linear programming formulations for the weighted matching problem.

1.4.1 LP Formulation for the Weighted Matching Problem

Let $G = (V, E, w)$ be an instance of the maximum–weight matching problem. The maximum–weight matching problem can be formulated as a zero–one integer linear programming problem. An incidence vector x is associated with the edges of G. Each component x_e is a decision variable having value 0 or 1. The relation between the incidence vector x and a matching M is as follows:

$$x_e = \begin{cases} 0 & \text{if } e \text{ does not belong to the matching } M, \\ 1 & \text{if } e \text{ does belong to the matching } M. \end{cases}$$

An incidence vector x corresponding to a given matching M is called the *characteristic vector* of M.

Let $S \subseteq E$ be a subset of edges and x an incidence vector associated with the edges E of G. $x(S)$ is defined as the sum over all components x_e with $e \in S$, i.e. $x(S) = \sum_{e \in S} x_e$.

We are now able to formulate the maximum–weight matching problem as a zero–one integer linear program (IWM):

$$\begin{array}{llrcll} \text{(IWM)} & \text{maximize} & w^T x & & & \\ & \text{subject to} & x(\delta(u)) & \leq & 1 & \text{for all } u \in V, \quad (1) \\ & & x_e & \in & \{0,1\} & \text{for all } e \in E. \quad (2) \end{array}$$

(IWM)(1) assures that each vertex has at most one incident edge that is matched. Note that each optimal solution x of (IWM) corresponds to a maximum–weight matching M. And conversely, every characteristic vector x to a maximum–weight matching M is an optimal solution to (IWM). Therefore, (IWM) does in fact formulate the maximum–weight matching problem.

A standard technique in combinatorial optimization is to relax the zero–one constraint (IWM)(2) which yields the linear programing relaxation (WM').

$$\begin{array}{llrcll} (\text{WM'}) & \text{maximize} & w^T x & & & \\ & \text{subject to} & x(\delta(u)) & \leq & 1 & \text{for all } u \in V, \quad (1) \\ & & x_e & \geq & 0 & \text{for all } e \in E. \quad (2) \end{array}$$

Unfortunately, (WM') does not have zero–one solutions only.[6] To see this, consider a graph $G = (V, E)$ having three vertices $V = \{a, b, c\}$ that lie on a odd length cycle, i.e. $E = \{ab, bc, ca\}$. Assume further that $w_e = 1$ for all edges $e \in E$. Then, $\hat{x}_e = 1/2$ for each edge e of G is an optimal solution to (WM') having objective value $3/2$. However, $\hat{x}$ is not a solution to (IWM) (the objective value of an optimal solution to (IWM) is 1).

Consequently, the two formulations (IWM) and (WM') are not equal, or to put it differently, (WM') is said to be *not as strong as* (IWM). A measure for the strength of a linear programming relaxation is the closeness of its *feasible set* to the *convex hull* defined by the feasible incidence vectors of the original integer program.

In general, the *feasible set* $\mathcal{F}^{(\text{LP})}$ to a linear programming formulation (LP) consists of all feasible incidence vectors to (LP). For example,

$$\mathcal{F}^{(\text{WM'})} = \{x \; : \; x \text{ satisfies (WM')(1) and (WM')(2)}\}.$$

The convex hull $\mathcal{P}^{(\text{LP})}$ of a feasible set $\mathcal{F}^{(\text{LP})}$ can be seen as a polyhedron spanned by $\mathcal{F}^{(\text{LP})}$.[7]

For an integer linear programming formulation (ILP) and its relaxation (LP') the relation $\mathcal{P}^{(\text{ILP})} \subseteq \mathcal{P}^{(\text{LP'})}$ always holds, whereas one cannot expect that the opposite does too. The relation between $\mathcal{P}^{(\text{IWM})}$ and $\mathcal{P}^{(\text{WM'})}$ is a perfect example.

Theorem 1.4.1 Two linear programming formulations (LP) and (LP') are *equally strong*, iff $\mathcal{P}^{(\text{LP})} = \mathcal{P}^{(\text{LP'})}$.

The question is, whether there exists a linear programming formulation similar to (WM') that is moreover as strong as (IWM).

Let $\mathcal{O}$ denote the set of all non–singleton odd cardinality subsets of V:

$$\mathcal{O} = \{\mathcal{B} \subseteq V \; : \; |\mathcal{B}| \text{ is odd and } |\mathcal{B}| \geq 3\}.$$

Consider the linear programming formulation (WM) below.

---

[6] However, the two linear programing formulations (WM') and (IWM) have been proved to be equivalent for the bipartite weighted matching problem. The proof is due to Birkhoff [Bir46].

[7] The convex hull $\mathcal{P}$ of a finite set $S = \{x_1, x_2, \ldots, x_k\} \in \mathbb{R}^n$ is defined as the set of all convex combinations of S:

$$\mathcal{P} = \{x = \textstyle\sum_{i=1}^k \lambda_i x_i \; : \; \sum_{i=1}^k \lambda_i = 1, \; x_i \in S \text{ and } \lambda_i \geq 0, \; 1 \leq i \leq k\}.$$

More precisely, we would have to distinguish between a polyhedron $\mathcal{P}^{(\text{LP})}$ which is defined by (i.e. is equal to) its feasible set $\mathcal{F}^{(\text{LP})}$ and a polyhedron $\mathcal{P}^{(\text{LP})}$ which is defined by the convex hull of its feasible set $\mathcal{F}^{(\text{LP})}$ (e.g. in cases where (LP) is an integer linear program). However, we do not wish to go into the details of polyhedral combinatorics at this point. Instead, for a more extensive discussion concerning these aspects, the interested reader is referred to Cook et al. [CCPS98] and Bertsimas and Tsitsiklis [BT97].

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

(WM) equals (WM') except that a new series of constraints (WM)(2) has been added. (WM)(2) states, that the number of matched edges in $\gamma(\mathcal{B})$, where $\mathcal{B} \subseteq V$ is a non–singleton odd cardinality set, is bounded above by $\lfloor |\mathcal{B}|/2 \rfloor$. Note that (WM)(2) coincides with one's intuition. It can easily be observed that each characteristic vector x to a given matching M must satisfy (WM)(1)–(3) and therefore: $\mathcal{P}^{(\text{IWM})} \subseteq \mathcal{P}^{(\text{WM})}$.

What consequences does the additional constraint (WM)(2) entail? As before, let us regard the graph G consisting of an odd cycle only. Setting $x_e = 1/2$ for all edges of G is not a feasible solution to (WM), since $x(\gamma(\{a, b, c\})) = 3/2 \nleq 1$.

The idea arises that (WM) is a stronger formulation than (WM'). And indeed, as the following lemma shows, the linear programming formulation (WM) is not only stronger than (WM'), but as strong as (IWM).

Lemma 1.4.1 Let $\mathcal{P}^{(\text{IWM})}$ and $\mathcal{P}^{(\text{WM})}$ represent the polyhedron of (IWM) and (WM), respectively. Then $\mathcal{P}^{(\text{IWM})} = \mathcal{P}^{(\text{WM})}$.

Lemma 1.4.1 is one of the cornerstones of the weighted matching theory. It is due to Edmonds. Generally, one can prove Lemma 1.4.1 either directly, or by an algorithmic proof.

We will do so by the latter method, i.e. we develop an algorithm that computes a matching M and moreover, the characteristic vector x to M will be an optimal solution to (WM). Further details are deferred to Section 1.6. Similar algorithmic proofs can be found in Pulleyblank [Pul95] and Cook et al. [CCPS98].

The direct proof is complex and not given here. Details can be found in the original work of Edmonds [Edm65a]. Cook et al. [CCPS98, Chapter 6] and Lovász and Plummer [LP86] are also excellent sources.

1.4.2 LP Formulation for the Weighted Perfect Matching Problem

The linear programming formulation for the maximum–weight perfect matching problem slightly differs from (WM) and will be sketched next. In Section 1.5 we will see that under certain conditions, each maximum–weight perfect matching problem can be reduced to the maximum–weight matching problem and contrariwise. Taking that fact into consideration, one may wonder if it is worth the effort to inspect the weighted perfect matching case separately. However, the differences between those two problems regarding linear programming formulation aspects are interesting to see and, moreover, both problems can be incorporated into one generic algorithm easily as, will be exploited in Section 1.6.

Again, we start with the integer linear program. Since every vertex has to be matched in the maximum–weight perfect matching problem, the primal condition (IWM)(1) becomes an equality constraint:

$$\begin{array}{llrcll}
\text{(IWPM)} & \text{maximize} & w^T x & & & \\
& \text{subject to} & x(\delta(u)) & = & 1 & \text{for all } u \in V, \quad (1)\\
& & x_e & \in & \{0,1\} & \text{for all } e \in E. \quad (2)
\end{array}$$

In the perfect case, too, the linear programming relaxation of (IWPM) is not as strong as (IWPM) itself. But as in the non–perfect case, adding a new series of constraints helps. The corresponding linear program is (WPM).

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

At this point one observes that the formulation of (WM) is a generalization of (WPM), since $\mathcal{P}^{(\text{WPM})}$ is a face of $\mathcal{P}^{(\text{WM})}$. The following lemma states that (IWPM) is as strong as (WPM).

Lemma 1.4.2 Let $\mathcal{P}^{(\text{IWPM})}$ and $\mathcal{P}^{(\text{WPM})}$ represent the polyhedron of (IWPM) and (WPM), respectively. Then $\mathcal{P}^{(\text{IWPM})} = \mathcal{P}^{(\text{WPM})}$.

As for Lemma 1.4.1, the generic algorithm in Section 1.6 will prove correctness of the stated lemma. For alternative proofs all references given for Lemma 1.4.1 apply.

1.4.3 An Alternative LP Formulation for the Weighted Perfect Matching Problem

In Section 1.6 we will develop a primal–dual method that computes an optimal solution to the linear programming formulations given above. The details of that method depend on those fixed formulations. However, an alternative linear programming formulation for the maximum–weight perfect matching problem exists and will be the subject of this section. The pros and cons of that alternative formulation with respect to the resulting primal–dual method will be discussed in detail in Section 1.6.5.

In both cases, i.e. the perfect and non–perfect weighted matching problem, we added a series of constraints to the relaxation of the integer linear program in order to obtain a linear program that is as strong as its integer linear program. Those constraints have been of the form:

$$x(\gamma(\mathcal{B})) \le \lfloor |\mathcal{B}|/2 \rfloor \quad \text{for all } \mathcal{B} \in \mathcal{O}. \tag{1.1}$$

However, for the weighted perfect matching problem, the same effect can be achieved by a different type of constraint:

$$x(\delta(\mathcal{B})) \ge 1 \quad \text{for all } \mathcal{B} \in \mathcal{O}. \tag{1.2}$$

(1.2) means that at least one edge that leaves a non–singleton odd cardinality set $\mathcal{B}$, i.e. is part of $\delta(\mathcal{B})$, must be matched.

The alternative formulation for the maximum–weight perfect matching problem is given in (WPM*).

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

As mentioned above, it can be shown that (WPM*) is as strong as (IWPM). Thus, (WPM*) is indeed an alternative to (WPM).

1.5 Reductions

We intend to use this section to show that each instance of the maximum–weight matching problem can be reduced to an instance of the maximum–weight perfect matching problem. Moreover, assuming the availability of a technique to discover the non–existence of a perfect matching, the contrary can be achieved as well.

We will describe these reductions by means of a transformation τ such that

(I1) for each instance $G = (V, E, w)$ of the maximum–weight matching problem, a maximum–weight perfect matching M' in $G' = \tau(G)$ can be translated to a maximum–weight matching M in G, and

(I2) under the assumption that a perfect matching exists for an arbitrary instance $G' = (V', E', w')$ of the maximum–weight perfect matching problem, a maximum–weight matching M in $G = \tau^{-1}(G')$ corresponds to a maximum–weight perfect matching M' in G'.

First, τ will be constructed suiting (I1) and after that the inverse transformation τ^{-1} satisfying (I2) will be given.

1.5.1 Reducing the Weighted Matching Problem to the Weighted Perfect Matching Problem

Let $G = (V, E, w)$ be an instance of the maximum–weight matching problem. We give a transformation $\tau(G) = G'$, where $G' = (V', E', w')$, and then proceed to show that G' satisfies (I1).

Assume, $\widetilde{G} = (\widetilde{V}, \widetilde{E}, \widetilde{w})$ is a copy of G. For each vertex u, edge e and weight w_e of G, we denote the corresponding vertex, edge and weight in $\widetilde{G}$ by $\widetilde{u}$, $\widetilde{e}$ and $\widetilde{w}_{\widetilde{e}}$, respectively.

Consider the graph G' that consists of G and $\widetilde{G}$. Moreover, let G' have additional zero–cost edges from each vertex u of G to $\widetilde{u}$ of $\widetilde{G}$. More precisely, G' is given as $V' = V \mathbin{\dot\cup} \widetilde{V}$

and

$$E' = E \,\dot\cup\, \widetilde{E} \,\cup\, \{u\widetilde{u} \;:\; u \in V \text{ and } \widetilde{u} \in \widetilde{V}\}.$$

The weight function w' of G' is defined as:

$$w'_{e'} = \begin{cases} w_{e'} & \text{when } e' \in E, \\ \widetilde{w}_{e'} & \text{when } e' \in \widetilde{E}, \\ 0 & \text{when } e' = u\widetilde{u} \text{ with } u \in V \text{ and } \widetilde{u} \in \widetilde{V}. \end{cases}$$

Lemma 1.5.1 Let $G' = \tau(G)$ as given above. Each maximum–weight perfect matching M' in G' then corresponds to a maximum–weight matching M in G.

Proof:
Let M' be a maximum–weight perfect matching in G'. The difference

$$M' \setminus \{u\widetilde{u} \;:\; u \in V \text{ and } \widetilde{u} \in \widetilde{V}\} = M \,\dot\cup\, \widetilde{M}$$

decomposes into $M \subseteq E$ and $\widetilde{M} \subseteq \widetilde{E}$. Since M' is of maximum weight, M must be a maximum–weight matching in G.

Conversely, let M be a maximum–weight matching in G and $\widetilde{M}$ the corresponding matching in $\widetilde{G}$. Then

$$M' = M \,\cup\, \widetilde{M} \,\cup\, \{u\widetilde{u} \in E' \;:\; u \text{ free in } G \text{ and } \widetilde{u} \text{ free in } \widetilde{G}\}$$

is a perfect matching in G' with weight $w'(M') = 2w(M)$. □

The stated lemma is often used to reduce the proof of Lemma 1.4.1 to the proof of Lemma 1.4.2.

1.5.2 Reducing the Weighted Perfect Matching Problem to the Weighted Matching Problem

Consider an instance $G' = (V', E', w')$ of the maximum–weight perfect matching problem. We will construct a transformation τ^{-1} that gives us an instance $\tau^{-1}(G') = G$, with $G = (V, E, w)$, of the maximum–weight matching problem satisfying (I2). However, we wish to emphasize that the reduction to be stated is correct only when a perfect matching does indeed exist in G'.

In the discussion that follows, we assume that all edge weights of G' are non–negative. We may make this assumption, since the weighted perfect matching problem is not affected when all edge weights are modified by adding a constant $c = \max\{|w_e| : e \in E\}$.

Define $G = (V, E, w)$ with $V = V'$ and $E = E'$. The edge weights in G will be set such that each maximum–weight matching M in G is perfect. This can be achieved by adding a positive value L to the original edge weights of G': $w_e = w'_e + L$.

Choosing L such that the total weight $w(\widetilde{M})$ of each perfect matching $\widetilde{M}$ in G is larger than the total weight of any non–perfect matching M in G yields the desired result. Let $n = |V|$ denote the number of vertices of G; n is assumed to be even, since otherwise no perfect matching exists in G'. Moreover, let $C = \max\{w'_e : e \in E'\}$ be the maximum edge weight in G'. By the definition of w, we have $C + L \geq w_e \geq L$. The total weight $w(\widetilde{M})$ of each perfect matching $\widetilde{M}$ is thus bounded below by $|\widetilde{M}|\ L = (n/2)\ L$. Conversely, the total weight $w(M)$ of a non–perfect matching M cannot be more than $|M|\ (C + L)$. Hence, choosing L such that the relation

$$(n/2)\ L \quad > \quad |M|\ (C + L) \tag{1.3}$$

holds, assures that each maximum–weight matching M in G will be perfect. The right–hand side of (1.3) maximizes for $|M| = (n/2) - 1$, since that is the largest cardinality of a non–perfect matching possible. Therefore, choosing $L := (n/2)\ C > ((n/2) - 1)\ C$ has the desired effect.

Lemma 1.5.2 Let $G = \tau^{-1}(G')$ as given above and assume a perfect matching exists in G'. Each maximum–weight matching M in G then corresponds to a maximum–weight perfect matching M' in G'.

Proof:
Let M be a maximum–weight matching in G. From the construction above, it immediately follows that M must be perfect. The total weight of a maximum–weight perfect matching in G' is thus $w'(M) = w(M) - |M|\ L = w(M) - (n/2)\ L$.

Conversely, let M' be a maximum–weight perfect matching in G' having total weight $w'(M')$. M' is then a perfect matching in G of weight $w(M') = w'(M') + |M|\ L = w'(M') + (n/2)\ L$. Due to the construction of G, no non–perfect matching can have total weight larger than or equal to $w(M')$. Thus, M' is a maximum–weight matching in G. $\square$

Each maximum–weight perfect matching problem can thus be solved by an algorithm for the maximum–weight matching problem using Lemma 1.5.2 and a further technique to discover the non–existence of a perfect matching in G (for example Corrollary 1.3.1).

Mehlhorn and Näher [MN99] use a similar construction to force a maximum–weight bipartite matching algorithm to find a maximum–weight matching along all maximum–cardinality bipartite matchings.

1.6 Primal–Dual Method

In Section 1.4.1 a linear programing formulation for the maximum–weight matching problem was introduced. Based on that formulation, we will use duality theory to obtain a first high–level primal–dual method to compute a maximum–weight matching to a given instance. A primal–dual method based on the maximum–weight perfect matching problem formulation of Section 1.4.2 will then be outlined.

Edmonds' blossom–shrinking approach will be extended in Section 1.6.3 such that it becomes a concrete derivation of those primal–dual methods. The resulting generic algorithm establishes correctness of Lemma 1.4.1 and Lemma 1.4.2 and will serve as the fundamental approach for our implementations.

We will complete this section by showing a useful property of the dual solution to the maximum–weight matching and maximum–weight perfect matching problem and, moreover, discuss the pros and cons of a similar algorithm for the maximum–weight perfect matching problem using the alternative formulation of Section 1.4.3.

1.6.1 Primal–Dual Method for the Maximum–Weight Matching Problem

We repeat the linear programing formulation of the maximum–weight matching problem considered in Section 1.4.1:

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

We will use duality theory in order to derive a primal–dual method that computes an optimal solution to (WM). The main idea is to compute a matching M whose characteristic vector x is a feasible and moreover optimal solution to (WM). We will assure optimality of x by a feasible solution to the dual linear program of (WM) that satisfies all complementary slackness conditions with x.

The dual linear program $(\overline{\text{WM}})$ to (WM) is given next. Each vertex u and each non–singleton odd cardinality set $\mathcal{B}$ has an associated dual variable y_u and $z_{\mathcal{B}}$, respectively.

$$\begin{array}{llrcll}
(\overline{\text{WM}}) & \text{minimize} & \displaystyle\sum_{u \in V} y_u + \sum_{\mathcal{B} \in \mathcal{O}} \lfloor |\mathcal{B}|/2 \rfloor \, z_{\mathcal{B}} & & & \\
& \text{subject to} & y_u & \geq & 0 & \text{for all } u \in V, & (1)\\
& & z_{\mathcal{B}} & \geq & 0 & \text{for all } \mathcal{B} \in \mathcal{O}, & (2)\\
& & \displaystyle y_u + y_v + \sum_{\substack{\mathcal{B} \in \mathcal{O} \\ uv \in \gamma(\mathcal{B})}} z_{\mathcal{B}} & \geq & w_{uv} & \text{for all } uv \in E. & (3)
\end{array}$$

We will call y_u and $z_{\mathcal{B}}$ the *dual value*, or alternatively the *potential* of vertex u and blossom $\mathcal{B}$. $(\overline{\text{WM}})(3)$ states that the potentials of the endpoints of an edge $e = uv$ plus the sum of all potentials of non–trivial odd cardinality sets containing that edge must be greater or equal to the weight of e.
To simplify further notations, we introduce the notion of the *reduced cost* of an edge e.

Definition 1.6.1 (Reduced Cost) Let (y, z) be a solution to the dual linear program $(\overline{\text{WM}})$. The *reduced cost* π_{uv} of an edge $e = uv$ with respect to (y, z) is defined as:

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

An edge $e = uv$ is called *tight*, when its reduced cost π_{uv} equals zero. Note that $(\overline{\text{WM}})(3)$ can be replaced by $\pi_{uv} \geq 0$ for all edges uv of E. Thus, $(\overline{\text{WM}})(3)$ assures that the reduced cost of each edge is non–negative.

Let us deduce the complementary slackness conditions for (WM) and ($\overline{\text{WM}}$). Given a primal feasible solution x to (WM) and a dual feasible solution (y, z) to ($\overline{\text{WM}}$), then x and (y, z) are optimal iff the complementary slackness conditions (CS)(1)–(3) hold.

$$\text{(CS)} \qquad \begin{array}{rclcrcll} x_{uv} & > & 0 & \Longrightarrow & \pi_{uv} & = & 0 & \text{for all edges } uv \in E, \quad (1) \\ y_u & > & 0 & \Longrightarrow & x(\delta(u)) & = & 1 & \text{for all nodes } u \in V, \quad (2) \\ z_{\mathcal{B}} & > & 0 & \Longrightarrow & x(\gamma(\mathcal{B})) & = & \lfloor |\mathcal{B}|/2 \rfloor & \text{for all } \mathcal{B} \in \mathcal{O}. \quad (3) \end{array}$$

What do the above constraints mean? We now proceed to give an interpretation. (CS)(1) requires that matched edges must be tight. Because of (CS)(2), free vertices must have potential zero. Finally, due to (CS)(3), when a non–singleton odd cardinality set $\mathcal{B}$ has potential different from zero, then a maximum number of edges in $\mathcal{B}$ must be matched, i.e. $\gamma(\mathcal{B})$ contains $\lfloor |\mathcal{B}|/2 \rfloor$ matched edges. We will also say $\mathcal{B}$ must be *full*. Observe that each non–trivial blossom is a non–singleton odd cardinality set that is full.

Assume now that the following four invariants hold for x and (y, z):

(I1) x is a feasible solution to (WM),
(I2) (y, z) is a feasible solution to ($\overline{\text{WM}}$),
(I3) (CS)(1) holds, and
(I4) (CS)(3) holds.

Maintaining (I1) to (I4) we will alter the solutions x and (y, z) such that the violations of (CS)(2) are successively reduced. Eventually, (CS)(2) will hold too and we will thus have obtained optimal solutions x and (y, z) to (WM) and ($\overline{\text{WM}}$).

Let r be a vertex that violates (CS)(2), i.e. r is free and $y_r > 0$. Our purpose is either to match r (and thus alter the primal solution x), or to adjust the dual solution (y, z) such that the potential of r equals zero. Having achieved either of those, r will no longer violate (CS)(2). The following strategy realizes the outlined idea.

First, we try to match r. However, notice that by (CS)(1) tight edges are qualified to be matching edges only. The attempt to match r using all current tight edges might fail. In this case, a so–called *dual adjustment* by some $\delta > 0$ is performed. That is, the dual solution (y, z) gets adjusted to (y', z') such that

(I5) the objective value of ($\overline{\text{WM}}$) strictly decreases,
(I6) the invariants (I1) to (I4) remain true for (y', z'),
(I7) in general, new tight edges exist with respect to (y', z'), and
(I8) the potential of r strictly decreases.

(I5) assures that the dual solution converges with its optimum.[8] When new tight edges result from the dual adjustment, the attempt to match r is continued. Note that (I7) will hold in general only, i.e. not every dual adjustment will produce new tight edges.[9]

[8]Actually, if (I5) did not hold, the termination could not even be guaranteed for real weights (see Aráoz and Edmonds [AE85]).

[9]The reason for this will become clear shortly. For the time being, the reader is asked to accept that we cannot guarantee each dual adjustment to produce new tight edges, since we must preserve (I6).

Eventually, after a series of dual adjustments either sufficiently many tight edges will exist such that r can be matched, or the potential of r will drop to zero (due to (I8)). We summarize the discussed primal–dual method in Algorithm 1.6.1.

Algorithm 1.6.1 Generic primal–dual method for the maximum–weight matching problem.

```
let $x$ and $(y, z)$ satisfy (I1) to (I4)
while there exists a free vertex $r$ with $y_r > 0$ {
   repeat {
      try to match $r$ using tight edges only
      if $r$ is not matched yet
         perform dual adjustment by $\delta > 0$ such that (I5) to (I8) hold
   } until $y_r = 0$ or $r$ is matched
}
```

The only missing details that have to be filled in are how to find the initial feasible solutions x and (y, z) that satisfy (I1) to (I4), how to match free vertices using tight edges and how to perform a dual adjustment satisfying (I5) to (I8). We will come back to these details in Section 1.6.3.

1.6.2 Differences in Weighted Perfect Matching Case

Some minor changes in the primal–dual method ensue for the maximum–weight perfect matching problem. (WPM) introduced in Section 1.4.2 is used as the linear programing formulation for the maximum–weight perfect matching problem.

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

(WPM) equals (WM) except that (WPM)(1) is an equality constraint. Consequently, the non–negativity constraints for all vertices in ($\overline{\text{WM}}$) do not occur in the dual linear program ($\overline{\text{WPM}}$) of (WPM).

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

Thus, the complementary slackness conditions for a primal solution x of (WPM) and a dual solution (y, z) of ($\overline{\text{WPM}}$) are comprised of (CS)(1) and (CS)(3) only. We repeat them as (PCS)(1) and (PCS)(2) below:

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

The description and arguments given for the non–perfect case no longer make sense now. In the perfect case, we therefore maintain primal and dual solutions x and (y, z) that satisfy the invariants (J1) to (J4).

(J1) x satisfies all conditions of (WPM) except (WPM)(1),

(J2) (y, z) is a feasible solution to ($\overline{\text{WPM}}$),

(J3) (PCS)(1) holds, and

(J4) (PCS)(2) holds.

Gradually, the violations of (WPM)(1) are decreased such that in the end, x becomes a primal feasible solution and thus is optimal, or one discovers that the objective value of ($\overline{\text{WPM}}$) is unbounded and therefore, no perfect matching exists (by weak duality). As before, tight edges are used to match a free vertex r. If the current tight edges do not suffice to match r, a dual adjustment by $\delta > 0$ is performed. δ must be chosen such that

(J5) the objective value of ($\overline{\text{WPM}}$) strictly decreases,

(J6) the invariants (J1) to (J4) remain true for the adjusted dual solution (y', z'),

(J7) in general, new tight edges exist with respect to (y', z').

The objective value of ($\overline{\text{WPM}}$) is unbounded, when δ can be made arbitrarily large, i.e. $\delta = \infty$. The modified generic algorithm reduces to:

Algorithm 1.6.2 Generic primal–dual method for the maximum–weight perfect matching problem.

```
let $x$ and $(y, z)$ satisfy (J1) to (J4)
while there exists a free vertex $r$ {
   repeat {
      try to match $r$ using tight edges only
      if $r$ is not matched yet {
         choose $\delta > 0$ such that (J5) to (J7) hold
         if $\delta = \infty$ terminate, since no perfect matching exists
         else perform dual adjustment by $\delta$
      }
   } until $r$ is matched
}
```

1.6.3 The Blossom–Shrinking Approach Revisited

Based on the primal–dual methods discussed in the preceding sections we will extend Edmonds' blossom–shrinking approach (see Section 1.3) such that it can solve instances of the weighted matching problem (non–perfect and perfect). We will first focus on the maximum–weight matching problem and outline the differences for the perfect matching case thereafter.

The following three details are still open and will be filled in next:

1. constructing the initial solutions x and (y, z) to (WM) and ($\overline{\text{WM}}$) that satisfy (I1) to (I4),
2. matching a free vertex r with non–zero potential using tight edges only, and
3. performing a dual adjustment by $\delta > 0$ and assuring the validity of (I5) to (I8).

Throughout this section, let $G = (V, E, w)$ be an instance of the maximum–weight matching problem. x will denote the characteristic vector to a matching M of G. We will often not distinguish between a matching M and its characteristic vector x, and use one notion for the other.

Finding Initial Solutions

Clearly, the empty matching $M = \emptyset$, i.e. $x_e = 0$ for each edge $e \in E$, is a feasible solution to (WM). For each vertex u the potential is set to $y_u = \max\{w_e/2 : e \in \delta(u)\}$. The approach will use the potentials $z_{\mathcal{B}}$ of blossoms only. That is, the potential $z_{\mathcal{B}}$ of each non–singleton odd cardinality set is regarded as being set to $z_{\mathcal{B}} = 0$. Exceptions are the potentials that are associated with a non–trivial blossom $\mathcal{B}$; these can have value $z_{\mathcal{B}} > 0$. Initially, no non–trivial blossoms exist. We thus obtain a feasible solution (y, z) to the dual linear program ($\overline{\text{WM}}$).

Moreover, note that x and (y, z) satisfy both conditions (CS)(1) and (CS)(3). In summary, we can state that x and (y, z) meet the invariants (I1) to (I4).

Different possibilities to obtain better initial solutions will be the subject of Section 3.5. For now, assume we start with the solutions x and (y, z) above.

Reducing the Violations of (CS)(2)

Consider a free vertex r with non–zero potential $y_r > 0$. First, we will describe the attempt to match r using tight edges only. The *dual adjustment step*, which is triggered when the search does not succeed due to insufficiently many tight edges, will be considered more closely afterwards.

Matching a free vertex r using tight edges. From the discussion in Section 1.3 one immediately observes that the task of matching r reduces to a search for an augmenting path starting with r. Therefore, we grow an alternating tree T rooted at r as described in Algorithm 1.3.1. However, in the weighted matching case it is crucial that only tight edges are used by the search in order to preserve (CS)(1). All details of Algorithm 1.3.1 apply.

In the case where a blossom $\mathcal{B}$ is shrunk, $\mathcal{B}$ is full, and, therefore, its potential $z_{\mathcal{B}}$ becomes accessible for future dual adjustments, as will be explained below.

When an augmenting path p consisting of tight edges has been found, the current matching M is augmented by p to M'. As a result, r will be matched thereafter and the new characteristic vector x' of M' no longer violates (CS)(2), as desired. All surface blossoms in T get unlabeled and T is destroyed. However, note the following difference. In the unweighted matching case, all non–trivial surface blossoms have been deleted when T was destroyed (see also Algorithm 1.3.2). For the weighted matching case the situation is different. It is crucial that non–trivial surface blossoms with $z_{\mathcal{B}} > 0$ retain their identity; deleting them would change the dual solution. As a consequence, non–trivial blossoms can occur outside of an alternating tree or as even or odd labeled tree blossoms.

When T is abandoned by the search this is due to the non–existence of further tight edges uv incident to any vertex $u^+ \in T$. In such cases, a dual adjustment is initiated as described below. New tight edges might exist thereafter and the search resumes with T.

Performing a dual adjustment. Consider a situation where the search for an augmenting path from r fails because there are no more tight edges incident to any vertex $u^+ \in T$.

We want to alter the potentials (y, z) of the vertices and non–singleton odd cardinality sets such that (I5) to (I8) are met. One way to achieve this is by adjusting (y, z) to (y', z') as stated below. The value of $\delta > 0$ will be determined shortly.

$$\begin{array}{rcll} y'_v &=& y_v - \delta & \text{for all } v^+ \in T, \\ y'_v &=& y_v + \delta & \text{for all } v^- \in T, \\ y'_v &=& y_v & \text{for all } v^{\{\varnothing|+\}} \notin T, \\ z'_{\mathcal{B}} &=& z_{\mathcal{B}} + 2\delta & \text{for all } \mathcal{B}^+ \in T, \\ z'_{\mathcal{B}} &=& z_{\mathcal{B}} - 2\delta & \text{for all } \mathcal{B}^- \in T, \\ z'_{\mathcal{B}} &=& z_{\mathcal{B}} & \text{for all } \mathcal{B}^{\{\varnothing|+\}} \notin T. \end{array}$$

Note that the adjustment has to be interpreted as follows. The potentials of all vertices in T are adjusted — including those that are contained in a non–trivial blossom. On the other hand, a potential $z_{\mathcal{B}}$ of a non–singleton odd cardinality set $\mathcal{B}$ is only adjusted when $\mathcal{B}$ is a non–trivial surface blossom of G.

We demonstrate that all conditions stated above are met when a dual adjustment by an appropriate value δ is performed.

First, we claim that the objective value of $(\overline{\text{WM}})$ strictly decreases by δ. Since $\delta > 0$, that will imply the correctness of (I5). We consider the rate of change $\Delta f = f' - f$ in the objective value of $(\overline{\text{WM}})$, where f and f' denote the objective value before and after the dual adjustment, respectively. The rate of change that is contributed to Δf by a trivial blossom u or non–trivial blossom $\mathcal{B}$ is denoted by Δf_u and $\Delta f_{\mathcal{B}}$. An odd labeled trivial surface blossom $v^- \in T$ obviously contributes $\Delta f_{v^-} = \delta$ to Δf. Analogously, $\Delta f_{v^+} = -\delta$ for an even labeled trivial surface blossom $v^+ \in T$. Let $\mathcal{B}^-$ be an odd labeled non–trivial surface blossom of T. Then,

$$\Delta f_{\mathcal{B}^-} \;=\; |\mathcal{B}|\delta + \lfloor |\mathcal{B}|/2 \rfloor\,(-2\delta) \;=\; |\mathcal{B}|\delta - (|\mathcal{B}|-1)\delta \;=\; \delta.$$

Analogously, for an even labeled non–trivial surface blossom $\mathcal{B}^+ \in T$ we have:

$$\Delta f_{\mathcal{B}^+} \;=\; |\mathcal{B}|(-\delta) + \lfloor |\mathcal{B}|/2 \rfloor\,(2\delta) \;=\; -|\mathcal{B}|\delta + (|\mathcal{B}|-1)\delta \;=\; -\delta.$$

We can conclude the argument now by observing that T always contains more even than odd surface blossoms (trivial or non–trivial). More precisely, let n^+ denote the number of even surface blossoms in T. Correspondingly, let n^- be the total number of odd surface blossoms in T. Since each even surface blossom except the root is matched with an odd surface blossom in T, we have: $n^+ = n^- + 1$. The total rate of change in the objective value is therefore $\Delta f = n^+(-\delta) + n^-\delta = -\delta$.

Let us prove that invariant (I6) holds. We start with the feasibility conditions (I1) and (I2). x stays feasible if it was so before the dual adjustment, since x is not altered at all.
Ensuring that the adjusted dual solution (y', z') is dual feasible entails some restrictions on the value of δ. First, δ cannot be larger than the smallest potential of an even labeled vertex $u^+ \in T$. Second, the potential of all non–trivial blossoms must stay non–negative, and therefore δ is bounded above by the minimal value $z_{\mathcal{B}}/2$ of an odd non–trivial surface blossom $\mathcal{B}^- \in T$. Finally, the reduced cost of all edges must be non–negative after the dual adjustment. This point demands closer inspection.
We only consider edges $e = uv$ with at least one endpoint in T; the reduced costs of edges having none of its endpoints in T do not change. Let π_{uv} denote the reduced cost of e before the dual adjustment and assume further that e does not lie in a blossom $\mathcal{B}$, i.e. $e \notin \gamma(\mathcal{B})$ for a blossom $\mathcal{B}$. We distinguish five cases.

Case 1: $u^+ \in T$

Case 1a: $u^+ \in T$ and $v^+ \in T$
both endpoints of e are decreased by δ. Since the new reduced cost $\pi_{uv} - 2\delta$ is restricted to being non–negative, we obtain an upper bound of $\delta \le \pi_{uv}/2$.

Case 1b: $u^+ \in T$ and $v^{\{\varnothing|+\}} \notin T$
the reduced cost π_{uv} of e will change by $-\delta$, resulting in another bound: $\delta \le \pi_{uv}$

Case 2: $u^+ \in T$ and $v^- \in T$
since u is decreased and v increased by δ, the reduced cost π_{uv} of e will not change.

Case 3: $u^- \in T$

Case 3a: $u^- \in T$ and $v^- \in T$
the potential of each endpoint u and v is increased by δ. The new reduced cost $\pi_{uv} + 2\delta$ of e is, obviously, non–negative.

Case 3b: $u^- \in T$ and $v^{\{\varnothing|+\}} \notin T$
the reduced cost π_{uv} increases to $\pi_{uv} + \delta$ and will hence stay feasible.

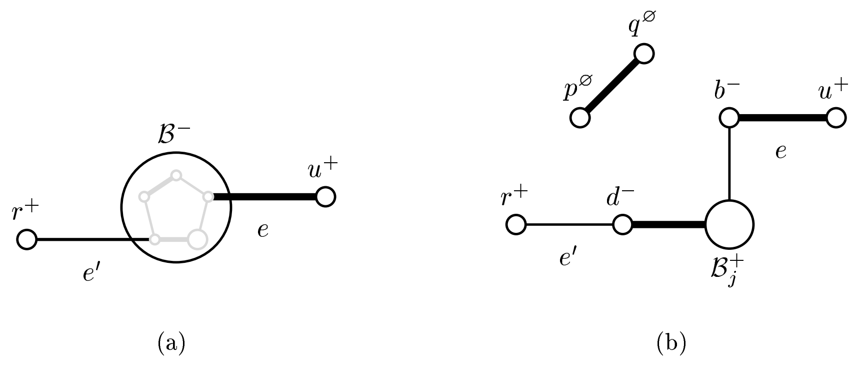

Figure 1.5: Let $\mathcal{B}$ be an odd blossom in the alternating tree T as depicted in (a). Immediate subblossoms of $\mathcal{B}$ are $d, \mathcal{B}_j, b, q$ and p. $\mathcal{B}_j$ is the only non–trivial subblossom of $\mathcal{B}$. When $\mathcal{B}$ is expanded (see (b)), all immediate subblossoms along the even length path from d to b become part of T and are labeled appropriately. p and q are unlabeled and leave T.

Consider the case now where $e = uv \in \gamma(\mathcal{B})$ is embedded in a blossom $\mathcal{B}$. Its reduced cost π_{uv} will not change, since the potentials of the endpoints u and v are both decreased or increased by δ, whereas the potential of $\mathcal{B}$ is increased or decreased by 2δ, respectively. Actually, this is the motivation for only changing the potential of non–trivial surface blossoms.

We conclude by noting that we have obtained the following bounds for the value of δ in order to stay dual feasible.

$$\delta = \min\{\delta_1, \delta_2, \delta_3, \delta_4\}$$

where

$$\begin{aligned}
\delta_1 &= \min_{u \in V} \ \{y_u \ : \ u^+ \in T\}, \\
\delta_2 &= \min_{uv \in E} \ \{\pi_{uv} \ : \ u^+ \in T, \ v^{\{\varnothing|+\}} \notin T\}, \\
\delta_3 &= \min_{uv \in E} \ \{\pi_{uv}/2 \ : \ u^+ \in T, \ v^+ \in T\}, \\
\delta_4 &= \min_{\mathcal{B} \in \mathcal{O}} \ \{z_{\mathcal{B}}/2 \ : \ \mathcal{B}^- \in T\}.
\end{aligned}$$

The convention of defining the minimum of an empty set to be ∞ is adopted here as well.

Finally, from the discussion above one can immediately affirm the validity of (cs)(1) and (cs)(3). This concludes the verification of (i6).

The only invariants not having been affirmed yet are (i7) and (i8). Let δ be chosen as stated above. Each vertex u, edge e or non–trivial blossom $\mathcal{B}$ that is responsible for one of the bounds δ_i, with $1 \le i \le 4$, is called the *responsible* vertex, edge or blossom, respectively.

Algorithm 1.6.3 Generic algorithm of the blossom–shrinking approach to compute a maximum–weight matching (perfect or non–perfect) in a general graph G.

```
let $M$ be the empty matching
set $y_u = \max\{w_e/2 : e \in E\}$ for each vertex $u$ in $G$
label each vertex $u$ in $G$ even
for each vertex $r$ in $G$ {
    if $r$ is matched continue with another vertex
    let $\mathcal{B}_r$ be the only blossom of $T$
    repeat {
        if non–perfect matching case and a vertex $u^+$ in $T$ with $y_u = 0$ exists {
            let $p'$ denote the alternating surface path from $\mathcal{B}_u$ to $\mathcal{B}_r$
            lift $p'$ to an alternating path $p$ from $u$ to $r$ using Lemma 1.3.1
            replace $M$ by $M \oplus p$
        }
        else if an alive edge $uv$ with $u^+$ in $T$ and $\pi_{uv} = 0$ exists {
            case $v^{\varnothing} \notin T$: grow step
            case $v^+ \in T$: shrink step
            case $v^+ \notin T$: augment step
        }
        else if there exists an odd blossom $\mathcal{B}^- \in T$ with $z_{\mathcal{B}} = 0$
            expand step for $\mathcal{B}$
        else {
            determine $\delta$ accordingly
            if $\delta = \infty$ and perfect matching case
                terminate, no perfect matching exists
            perform dual adjustment by $\delta$
        }
    } until $r$ is matched
}
```

Consider the case $\delta = \delta_1$. The potential y_u of the responsible vertex $u^+ \in T$ will be decreased to zero by the dual adjustment. Since u is even, an even (possibly zero) length alternating path p from u to r exists. p starts with a matching edge and ends with a non–matching edge. We can match r by replacing M by $M \oplus p$. Note that u will thereafter be free. However, this is legal since the potential of u equals zero. Thus, the number of violations of (CS)(2) has indeed decreased by one.

Assume now that $\delta = \delta_i$ for $i = 2, 3$. Let $e = uv$ be the responsible edge to δ_i. Obviously, e will become tight and can thus be used either to extend T ($\delta = \delta_2$) or to shrink a new blossom ($\delta = \delta_3$).

Finally, let $\delta = \delta_4$ and $\mathcal{B}^- \in T$ be the responsible blossom. Then, $z_{\mathcal{B}}$ of $\mathcal{B}$ will drop to zero after the dual adjustment and thus cannot participate in another dual adjustment. The action to be taken is to *expand* $\mathcal{B}$, which is somehow the opposite to shrinking a blossom. $\mathcal{B}$ gets expanded by raising all its immediate subblossoms to the surface. Since $\mathcal{B}$ is an odd blossom of T, there must be a matching tree edge e and a non–matching tree edge e' incident to $\mathcal{B}$. Let b and d denote the endpoint of e and e' that is contained in $\mathcal{B}$. There must exists an even length alternating path p from $\mathcal{B}_d$ to $\mathcal{B}_b$, the immediate subblossoms of $\mathcal{B}$ containing d and b, lying exclusively in $\gamma(\mathcal{B})$. Moreover, all edges in p are tight. We add p and thus all immediate subblossoms lying on p to T and label them according to their even or odd length distance to the root blossom $\mathcal{B}_r$ of T. All other immediate subblossoms of $\mathcal{B}$ not lying on p get unlabeled and leave the tree T. In Figure 1.5 an example is given of a so–called *expand step.*

Obviously, (I7) holds whenever $\delta = \delta_i$, with $i = 2, 3$. Moreover, note that $\delta = \delta_1$ will happen at most once per search and the occurrences of $\delta = \delta_4$ during a search are bounded by $O(n)$.[10] Finally, when the potentials are adjusted in the way stated above, (I8) certainly holds.

Let us briefly consider the differences for the maximum–weight perfect matching case. The initial solutions constructed above will certainly validate (J1) to (J4). Moreover, the details to match a free vertex using only tight edges stay the same. Moreover, the stated dual adjustment will assure invariants (J5) to (J7). The only difference is that the potentials of even tree vertices are no longer restricted to being non–negative. As a consequence, δ is not bounded above by δ_1. Therefore, choosing

$$\delta = \min\{\delta_2, \delta_3, \delta_4\}$$

yields the desired result for the maximum–weight perfect matching problem. Note that $\delta = \infty$ might in fact happen in the perfect case, whereas this is prevented by the existence of δ_1 in the non–perfect case.

Finally, we summarize Edmonds' blossom–shrinking approach to find a maximum–weight matching (perfect or non–perfect) in a general graph by the generic algorithm depicted in Algorithm 1.6.3.

1.6.4 Half–Integrality of the Dual Solution

We will use this section to prove an important property of the dual solution constructed by the approach described in the preceding section.

Lemma 1.6.1 Let (y, z) be an optimal solution to ($\overline{\text{WM}}$), where all edge weights are integral. Then (y, z) is half–integral, or more precisely:

$$y_u \equiv 0 \pmod{\tfrac{1}{2}} \quad \text{for all } u \in V\text{, and} \tag{1}$$

$$z_{\mathcal{B}} \equiv 0 \pmod{1} \quad \text{for all } \mathcal{B} \in \mathcal{O}. \tag{2}$$

Proof:
Assume the algorithm starts with the initial solution (y, z) as described above, i.e. $y_u = \max\{w_e/2 : e \in E\}$ for each vertex u and $z_{\mathcal{B}} = 0$ for all non–singleton odd cardinality sets $\mathcal{B}$. When w_e is integral for each edge e, (1) and (2) hold.

[10] Observe that once a blossom becomes an even tree blossom, it will stay even labeled and in T for the rest of the search.

Let (y, z) be a dual solution satisfying (1) and (2). Consider a dual adjustment by $\delta > 0$ and let (y', z') be the resulting dual solution. (1) and (2) will remain true for (y', z') when δ can be proved to be half–integral. δ is obviously half–integral when $\delta = \delta_1$ or $\delta = \delta_4$ (actually, δ is integral iff $\delta = \delta_4$). The reduced cost π_{uv} of an edge uv is guaranteed to be half–integral by definition and (1) and (2). Thus $\delta = \delta_2$ is half–integral. Finally, consider the case $\delta = \delta_3$. We will show that the reduced cost π_{uv} of an edge uv with $u^+ \in T$ and $v^+ \in T$ must be integral. To see this, note that all edges $e = \hat{u}\hat{v}$ in T are tight and all edge weights are integral. Thus, for these edges we have:

$$y_{\hat{u}} + y_{\hat{v}} + \sum_{\substack{\mathcal{B} \in \mathcal{O} \\ \hat{u}\hat{v} \in \gamma(\mathcal{B})}} z_{\mathcal{B}} = w_{\hat{u}\hat{v}}$$

which implies that $y_{\hat{u}} + y_{\hat{v}} \equiv 0 \pmod 1$ must hold for any two vertices $\hat{u}$ and $\hat{v}$ in T. Thus we can infer that the reduced cost π_{uv} of the edge uv is integral, and this concludes the proof. □

One can immediately affirm the following corollary for the maximum–weight perfect matching case.

Corollary 1.6.1 Let (y, z) be an optimal solution to $(\overline{\text{WPM}})$, where all edge weights are integral. Then (y, z) is half–integral.

1.6.5 Using the Alternative LP Formulation — Algorithmic Consequences

As was mentioned above, the details of the primal–dual method we have developed depend on the underlying linear programming formulation. Using the alternative linear programming formulation (WPM*) introduced in Section 1.4.3 one may hope to obtain a different approach for the maximum–weight perfect matching problem — which could be implemented more efficiently.
The differences of a primal–dual method based on the linear programming formulation (WPM*) are the subject of this section.

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

The dual linear program $(\overline{\text{WPM}^*})$ to (WPM*) is given below.

$$(\overline{\text{WPM}^*}) \quad \begin{array}{llcll} \text{minimize} & \sum_{u \in V} y_u + \sum_{\mathcal{B} \in \mathcal{O}} z_{\mathcal{B}} & & & \\ \text{subject to} & z_{\mathcal{B}} & \geq & 0 & \text{for all } \mathcal{B} \in \mathcal{O}. \quad (1) \\ & y_u + y_v + \sum_{\substack{\mathcal{B} \in \mathcal{O} \\ uv \in \delta(\mathcal{B})}} z_{\mathcal{B}} & \geq & w_{uv} & \text{for all } uv \in E, \quad (2) \end{array}$$

Note that the reduced cost π_{uv} of an edge uv with respect to a dual solution (y, z) is now defined differently:

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

The complementary slackness conditions are thus:

$$\begin{array}{lrclcrcll} (\text{PCS}^*) & x_{uv} & > & 0 & \Longrightarrow & \pi_{uv} & = & 0 & \text{for all edges } uv \in E, & (1) \\ & z_{\mathcal{B}} & > & 0 & \Longrightarrow & x(\delta(\mathcal{B})) & = & 1 & \text{for all } \mathcal{B} \in \mathcal{O}. & (2) \end{array}$$

All details of the primal–dual method for the weighted perfect matching case apply. However, the dual adjustment is performed differently in order to assure (J5) to (J7). The potentials are adjusted for surface blossoms (trivial or non–trivial) only.

$$\begin{array}{rcll} y'_v & = & y_v - \delta & \text{for all } v^+ \in T, \\ y'_v & = & y_v + \delta & \text{for all } v^- \in T, \\ y'_v & = & y_v & \text{for all } v^{\{\varnothing|+\}} \notin T, \\ z'_{\mathcal{B}} & = & z_{\mathcal{B}} - \delta & \text{for all } \mathcal{B}^+ \in T, \\ z'_{\mathcal{B}} & = & z_{\mathcal{B}} + \delta & \text{for all } \mathcal{B}^- \in T, \\ z'_{\mathcal{B}} & = & z_{\mathcal{B}} & \text{for all } \mathcal{B}^{\{\varnothing|+\}} \notin T. \end{array}$$

It seems one can implement the dual adjustment stated above more efficiently since only surface blossoms have to be considered. However, the crux of using the linear programming formulation (WPM*) is the computation of the reduced cost of an edge.

During the course of Algorithm 1.6.3 the reduced cost π_{uv} of alive edges uv will have to be computed frequently. Using the approach discussed in the preceding section, this can be achieved by taking the potentials of u and v and the edge weight w_{uv} into consideration.[11] In the approach just sketched, one would additionally have to take into consideration all potentials $z_{\mathcal{B}}$ of blossoms $\mathcal{B}$ with $uv \in \delta(\mathcal{B})$.

1.7 Survey of Different Realizations

Over the last four decades various polynomial–time realizations of the blossom–shrinking approach discussed in Section 1.6.3 have evolved. The first was suggested by Edmonds himself as early as 1965. Its theoretical running–time was bounded by $O(n^2 m)$. Permanent improvements regarding the theoretical running–time have been achieved successively using different strategies and data structures. The current best and optimal approach for general edge weights has a running–time of $O(n(m + n \log n))$ and is due to Gabow [Gab90]. We wish to use this section to portray the main ideas behind four different polynomial–time realizations of the blossom–shrinking approach.

One can view the blossom–shrinking approach as working in *phases*. A phase terminates when an additional violation has been eliminated, i.e. a violation of (CS)(2) in the

[11]Remember that alive edges are not contained in any blossom and hence $\sum_{\substack{\mathcal{B} \in \mathcal{O} \\ uv \in \gamma(\mathcal{B})}} z_{\mathcal{B}} = 0$.

maximum–weight matching and a violation of (WPM)(1) in the maximum–weight perfect matching case. Since at most n violations exist, where n denotes the number of vertices in G, we have $O(n)$ phases. Next, we will argue that the number of dual adjustments per phase is bounded by $O(n)$. Therefore, observe that $\delta = \delta_1$ occurs at most once in a phase. When $\delta = \delta_i$, with $i = 2, 3$, at least one (formerly non–even labeled) vertex becomes an even tree vertex. When a vertex has become an even tree vertex it will stay even and reside in the tree until the end of the phase. Thus $\delta = \delta_i$, $i = 2, 3$, occurs $O(n)$ times. Finally, whenever $\delta = \delta_4$, a blossom gets expanded. Since a blossom cannot contain more than n vertices this will also happen at most $O(n)$ times.

A non–trivial part of the algorithm is to maintain the surface graph G. We sketch the idea of using a *union–find* data structure that additionally supports a *split* operation. Each vertex knows its surface blossom, e.g. by a pointer, and each surface blossom maintains a list of all its vertices. Identifying the surface blossom of a vertex thus takes time $O(1)$. Two blossom $\mathcal{B}_i$ and $\mathcal{B}_j$ are *united by size*. That is, w.l.o.g. let $|\mathcal{B}_i| > |\mathcal{B}_j|$. The pointer of each vertex in $\mathcal{B}_j$ is set to $\mathcal{B}_i$, the list of $\mathcal{B}_j$ is appended to $\mathcal{B}_i$ and $\mathcal{B}_j$ is destroyed. $\mathcal{B}_i$ then represents the new blossom. Split operations, too, are done by size. The list of a surface blossom $\mathcal{B}_i$ is split into two lists $\mathcal{B}_i$ and $\mathcal{B}_j$. Again, the larger blossom, say $\mathcal{B}_i$, is reused and each pointer of a vertex in the smaller blossom is set to $\mathcal{B}_j$. By always resetting the pointers of the smaller blossom, we can assure that each fixed vertex contributes no more than $O(\log n)$ time to a series of n union or split operations. Note, however, that the claimed time bound only holds for a series of split followed by a series of union operations, or vice versa; and not for an arbitrary order of intermixed union and split operations. But since a vertex can participate in a series of at most $O(n)$ split (expand steps) followed by a series of $O(n)$ union (shrink steps) operations, a total time of $O(n \log n)$ per phase results. This will be sufficient for all four realizations presented next.

The realizations differ in the way they find tight edges, determine the value of δ and perform a dual adjustment.

1.7.1 An $O(n^2m)$ Approach

A simple realization needs time $O(m)$ to find tight edges and to determine the value of δ (each edge is inspected once). Performing a dual adjustment can be achieved by explicitly updating the potential of each vertex and non–trivial surface blossom which takes time $O(n)$. The total running–time is thus $O(n^2(n + m)) = O(n^2m)$ or $O(n^4)$, since m is bounded above by n^2. This approach is essentially the one which was suggested first by Edmonds [Edm65a].

1.7.2 An $O(n^3)$ Approach

The only parts that need more than $O(n)$ time per dual adjustment in the above realization are the identification of tight edges and the determination of δ, or, to be more precise, the determination of δ_2 and δ_3.[12] As we shall see, either can be achieved

[12]Obviously, the determination of δ_1 and δ_4 can easily be achieved in time $O(n)$.

in time $O(n)$. The resulting $O(n^3)$ approach is due to Lawler [Law76].

We say an alive edge e incident to a surface blossom $\mathcal{B}$ (trivial or non–trivial) is a *best edge* of $\mathcal{B}$ when its reduced cost is minimal, i.e.

$$\pi_e = \min\{\pi_{uv} \,:\, uv \in \delta(\mathcal{B}) \text{ and } uv \text{ is alive}\}.$$

When several such edges exist for $\mathcal{B}$, *the* best edge of $\mathcal{B}$ refers to an arbitrary one of these.

To handle δ_2, each vertex $u^{\{\varnothing|+\}} \notin T$ stores its *best edge* uv from u to an even labeled tree vertex $v^+ \in T$. Moreover, u stores the reduced cost π_{uv} of its best edge. Since odd vertices might leave T and get unlabeled (due to an expand step), the same data must be available for each odd tree vertex $u^- \in T$. Finding tight edges and determining δ_2 can then be achieved in time $O(n)$ by inspecting each best edge of $u^{\{\varnothing|+\}} \notin T$ vertices. A dual adjustment is performed by updating the reduced cost of all best edges of $u^{\{\varnothing|+\}} \notin T$ vertices which takes at most $O(n)$ time.

Each surface blossom $\mathcal{B}_k^+ \in T$ stores for each adjacent surface blossom $\mathcal{B}_j^+ \in T$, with $\mathcal{B}_k \neq \mathcal{B}_j$, the *best edge* $e_{kj} = uv$ with $u \in \mathcal{B}_k$ and $v \in \mathcal{B}_j$. Moreover, $\mathcal{B}_k$ knows the reduced cost π_{e_k} of the best edge e_k of all best edges e_{kj}. Finding tight edges is achieved by inspecting all best edges of the blossom whose best edge has reduced cost zero. The time needed to do so is bounded by $O(n)$. By exploring the reduced cost π_{e_k} of the best edge e_k to each blossom $\mathcal{B}_k^+ \in T$, δ_3 can be determined in time $O(n)$. Adjusting the reduced cost π_{e_k} of each blossom $\mathcal{B}_k^+ \in T$ needs time $O(n)$.

It remains to be shown, however, that the information associated with the maintenance of δ_2 and δ_3 can be kept correct without using time more than $O(n^2)$ per phase. Whenever a vertex becomes an even tree vertex, its edges are scanned and the information for δ_2 and δ_3 is updated. When a new blossom $\mathcal{B}_k$ is formed by s immediate subblossoms, it takes time $O(sn)$ to set up the data for $\mathcal{B}_k$.[13] The total cost $T(n)$ per phase to maintain δ_3 can then be computed by the following recursion: $T(n) = O(sn) + T(n-s)$. By induction it follows that $T(n)$ equals $O(n^2)$. Since each edge is scanned at most twice in a phase (once from each endpoint) this contributes time $O(m)$ per phase. Altogether, the approach needs time $O(n(n^2+m)) = O(n^3)$.

Finally, observe that the idea of maintaining the best (alive) edge to each pair of even surface blossoms gives a lower bound of $\Omega(n^2)$ per phase.

1.7.3 An $O(nm\log n)$ Approach

Another realization, which improves the theoretical running–time to $O(nm\log n)$, is due to Galil, Micali and Gabow [GMG86]. This approach is superior to the $O(n^3)$

[13]We give details to derive the claimed time bound. Let $\mathcal{B}_i^-$, $1 \le i \le \lfloor s/2 \rfloor$, denote the immediate odd subblossoms of $\mathcal{B}_k$. Each $\mathcal{B}_i$, $1 \le i \le \lfloor s/2 \rfloor$ is made even. All edges of the vertices contained in $\mathcal{B}_i$, $1 \le i \le \lfloor s/2 \rfloor$, are scanned to update the information for δ_2 and to determine the best edges e_{ij} of $\mathcal{B}_i$ to other even tree blossoms $\mathcal{B}_j^+ \in T$. Thereafter, the best edges e_{kj} and the reduced cost of the best edge e_k of all best edges of $\mathcal{B}_k$ can be determined in time $O(sn)$. We have to update the best edge information of each blossom $\mathcal{B}_j^+ \in T$ adjacent to the new blossom $\mathcal{B}_k$. This will need time $O(sn)$. To see this, consider a fixed blossom $\mathcal{B}_j^+ \in T$ that is adjacent to $\mathcal{B}_k$. Deleting all best edges e_{ji} to an even subblossom $\mathcal{B}_i^+ \in T$ of $\mathcal{B}_k$ takes time $O(s)$. Updating the best edge e_{jk} to $\mathcal{B}_j$ takes time $O(1)$. Since the number of adjacent blossoms is bounded by n the total time of $O(sn)$ results.

approach for sparse graphs. More or less the same ideas as in the $O(n^3)$ approach are reused. However, priority queues will help to achieve an $O(m \log n)$ time bound per phase. We will not go into detail here but postpone the discussion to Chapter 2. Only a few differences are outlined.

For example, δ_3 will be maintained by a priority queue. At first glance the usage of priority queues does not seem to work, due to the frequent changes of the priorities after a dual adjustment. Taking advantage of the fact that all priorities change uniformly will help to circumvent this problem in an efficient way.
Another major difference to the $O(n^3)$ approach is that we abandon the goal of only keeping track of the alive edges between even tree blossoms. Instead, a *lazy deletion* strategy is used to maintain δ_3. That is, every alive edge that might be of interest for δ_3 is inserted into δ_3. As a consequence, after a series of shrinkings, δ_3 might contain edges that are no longer alive. These edges are deleted when they are encountered as the minimal element of δ_3. Actually, this will be the only point where the running–time of $O(m + n \log n)$ per phase is exceeded.

For the sake of completeness we state the theoretical running–time for finding tight edges, for determining δ and for performing a dual adjustment. Finding a new tight edge will correspond to a *delete min* operation on a priority queue and thus takes, at most, time $O(\log n)$. The same will hold for the determination of δ. To perform a dual adjustment, however, will only take time $O(1)$ and is thus an immense speed–up compared to the $O(n^3)$ approach.

1.7.4 An $O(n(m + n \log n))$ Approach

In 1990, Gabow [Gab90] presented a data structure that can be used to realize a phase of Edmonds' blossom–shrinking approach in theoretical running–time $O(m + n \log n)$. Gabow claimes this time bound to be optimal: sorting n numbers can be reduced to a search for an augmenting path in Edmonds' blossom–shrinking approach; originally, a similar argument was given by Fredman and Tarjan [FT87] to prove optimality of Dijkstra's algorithm. Since each edge may be considered once during a search, a lower bound of $\Omega(m + n \log n)$ per phase results.

The details of the approach are complex and will not be given here. We attempt to sketch the idea, although this is difficult without going into detail. As mentioned before, only the maintenance of δ_3 needs special refinement. Therefore, a kind of alternating tree $\mathcal{T}$ is grown. Each blossom forming edge uv (i.e. the edge uv with $u^+ \in T$ and $v^+ \in T$) is replaced by two (directed) back edges ul and vl, where l denotes the lowest common ancestor of u and v in $\mathcal{T}$. A shrink operation then corresponds to uniting all surface blossoms along the cycle $C_1 = (l, \ldots, u, l)$ and $C_2 = (l, \ldots, v, l)$. These back edges are further partitioned into $\log n$ sets called *packets*. Roughly speaking, by dealing with the packets' minima (in terms of reduced cost) the desired time bound can be achieved.

However, the underlying data structures are complex and we doubt that an implementation would be efficient in practice.

Chapter 2

$O(nm \log n)$ Approach

The demanding and costly parts in Edmonds' blossom–shrinking approach are the performing of dual adjustments and the determination of the value of δ (see Section 1.6.3). In 1986, Galil, Micali, and Gabow [GMG86] presented a strategy that enables a phase of Edmonds' blossom–shrinking approach to be realized in time $O(m \log n)$. The time bound is achieved by using a sophisticated data structure, which they call *generalized priority queues*. Generalized priority queues support all standard priority queue operations. Additionally, the priorities of certain subgroups of elements in the queue can be uniformly changed by a single operation.

The ideas we will develop in this chapter are similar to or have been developed from the ideas of Galil et al. However, our approach differs with regard to the maintenance of the varying priorities. Galil et al. handle these changes within the priority queue data structure, whereas we will establish a series of formulae that enable us to compute these priorities as needed.

First, we shall illustrate how the blossom potentials and reduced costs of edges can be computed after a series of dual adjustments. As a consequence, the time required to perform a dual adjustment will be reduced to $O(1)$. Next, the concept of using priority queues to determine the value δ and thus also the responsible vertex, edge or blossom will be considered more closely. Finally, an obvious but mistaken realization will motivate the application of *concatenable priority queues*.

2.1 Varying Potentials and Reduced Costs

The frequent modifications of the blossom potentials and, consequently, the reduced cost of edges caused by a dual adjustment make it difficult to realize a phase in the blossom–shrinking approach efficiently. However, the a considerable advantage is that these modifications occur in a uniform manner. In the subsequent sections we will illustrate how to take advantage of that fact.

2.1.1 Potential Update

Consider a dual adjustment that is performed during the course of Algorithm 1.6.3 (described in Section 1.6.3). Let T denote the current alternating tree. A dual adjustment by δ affects the potentials of all vertices and non–trivial surface blossoms as follows. The vertex potential changes by $-\delta$ for an even tree vertex $u^+ \in T$, by $+\delta$ for an odd tree vertex $u^- \in T$ and by 0 for an non–tree vertex $u^{\{\varnothing|+\}} \notin T$. Correspondingly, the potential of a non–trivial surface blossom is adjusted by $+2\delta$ for an even tree blossom $\mathcal{B}^+ \in T$, by -2Δ for an odd tree blossom $\mathcal{B}^- \in T$ and by 0 for a non–tree blossom $\mathcal{B}^{\{\varnothing|+\}} \notin T$.

Therefore, after a series $\delta_1, \delta_2, \ldots, \delta_k$ of dual adjustments the so–called *actual potential* of a vertex or non–trivial surface blossom can be computed by taking its initial potential, its *status* and the value of $\Delta = \sum_{i=1}^{k} \delta_i$ into consideration. The status of a blossom (trivial or non–trivial) is given by its label and the property of either being a tree or a non–tree blossom.

More precisely, as long as the status of a vertex u does not change it is possible to obtain its actual potential $\widetilde{y}_u$ from its initial potential y_u by the following formula:

$$\widetilde{y}_u = y_u + \sigma\Delta.$$

Similarly, on the assumption that the status of a non–trivial surface blossom $\mathcal{B}$ is invariant, the actual potential $\widetilde{z}_{\mathcal{B}}$ can be computed via its initial potential $z_{\mathcal{B}}$:

$$\widetilde{z}_{\mathcal{B}} = z_{\mathcal{B}} - 2\sigma\Delta.$$

The coefficient σ depends on the current status of a blossom $\mathcal{B}$ (trivial or non–trivial) and will be called the *status indicator*:

$$\sigma = \begin{cases} -1 & \text{when } \mathcal{B}^+ \in T, \\ 1 & \text{when } \mathcal{B}^- \in T, \text{ and} \\ 0 & \text{when } \mathcal{B}^{\{\varnothing|+\}} \notin T. \end{cases}$$

However, the status of a blossom changes during the course of the algorithm and the formulae given above are somewhat oversimplified. We next refine these formulae such that arbitrary status changes can be handled as well.

Consider first of all a status change for a vertex u. Let Δ_1 and Δ_2 denote the sum of dual adjustments before and after the status change and assume u changes its status indicator from σ to σ'. Then,

$$\widetilde{y}_u = y_u + \sigma\Delta_1 + \sigma'\Delta_2 \overset{\Delta=\Delta_1+\Delta_2}{=} y_u + (\sigma - \sigma')\Delta_1 + \sigma'\Delta.$$

That is, we need to *correct* the potential y_u by $+(\sigma - \sigma')\Delta$ at the point of time when u changes its status indicator from σ to σ' (and thus $\Delta = \Delta_1$).

Analogously, let us consider a status change for a non–trivial surface blossom $\mathcal{B}$. It is

$$\widetilde{z}_{\mathcal{B}} = z_{\mathcal{B}} - 2\sigma\Delta_1 - 2\sigma'\Delta_2 = z_{\mathcal{B}} - 2(\sigma - \sigma')\Delta_1 - 2\sigma'\Delta$$

and therefore the potential $z_{\mathcal{B}}$ is corrected by $-2(\sigma - \sigma')\Delta$ when the current status indicator σ of $\mathcal{B}$ changes to σ'. Again, at this point Δ will equal Δ_1.

As will become clear shortly, we cannot afford to correct the potential of each vertex contained in a non–trivial surface blossom separately. Observe, however, that the *correction value* of a vertex potential and that of the non–trivial surface blossom containing that vertex differs by a multiplicative factor of -2. We can therefore simulate the potential corrections by means of an *offset* assigned to each surface blossom, as described next.

Each surface blossom $\mathcal{B}$ (trivial or non–trivial) has an offset denoted by $\mathit{offset}_{\mathcal{B}}$. The actual potential of a vertex u is then computed by

$$\widetilde{y}_u = y_u + \mathit{offset}_{\mathcal{B}} + \sigma\Delta, \tag{2.1}$$

where $\mathcal{B}$ corresponds to the surface blossom containing u (trivial or non–trivial). Accordingly, the actual potential of a non–trivial surface blossom $\mathcal{B}$ can be obtained by

$$\widetilde{z}_{\mathcal{B}} = z_{\mathcal{B}} - 2\mathit{offset}_{\mathcal{B}} - 2\sigma\Delta. \tag{2.2}$$

A status change for a surface blossom $\mathcal{B}$ then reduces to an update of its offset value:

$$\mathit{offset}_{\mathcal{B}} = \mathit{offset}_{\mathcal{B}} + (\sigma - \sigma')\Delta, \tag{2.3}$$

where Δ denotes the sum of dual adjustments up to the time of the status change.

During the course of the algorithm, the offset of a surface blossom $\mathcal{B}$ is adjusted as in (2.3) whenever its status changes. The discussion of how to handle the offsets in a shrink or an expand step is postponed to the next but one section.

2.1.2 Maintenance of Reduced Costs

For each vertex $u^{\{\varnothing|+\}} \notin T$ and $u^- \in T$ we will need to keep track of the best edge uv to an even labeled tree vertex and the reduced cost π_{uv} of that edge. We will do so by assigning a pair (π_{uv}, uv) to each such vertex, where uv denotes an edge incident to u, with $v^+ \in T$, having reduced cost π_{uv}.

In the context of this chapter it would be sufficient to handle only one such pair per vertex. However, we will consider a more general case where each vertex u is associated with a series $(\pi_{uv_1}, uv_1), (\pi_{uv_2}, uv_2), \ldots, (\pi_{uv_k}, uv_k)$ of pairs. Each pair (π_{uv_i}, uv_i) keeps an edge uv_i incident to u, with $v_i^+ \in T$, and the reduced cost π_{uv_i} of that edge.
This extended view will turn out to be reasonable in Chapter 3 (Section 3.4), where various alternating trees are simultaneously grown and hence several edges and their reduced costs will be associated with any vertex, i.e. also with even labeled tree vertices. In the subsequent sections, we will explicitly mention when the results apply to the extended view only.

The reduced costs of all edges associated with a vertex u may vary with dual adjustments. As for the blossom potentials, we will elaborate a formula which enables the *actual reduced costs* of these edges to be computed.

For a vertex $u^+ \in T$, $u^- \in T$ or $u^{\{\varnothing|+\}} \notin T$, a dual adjustment by δ changes the reduced costs of all edges associated with u uniformly by -2δ, 0 or $-\delta$, accordingly. Therefore, as long as u does not change its status, we can again compute the *actual reduced cost* $\widetilde{\pi}_{uv_i}$ of an edge uv_i after a series of dual adjustments taking its initial reduced cost, the status of u and the total dual adjustment value Δ into consideration. The computation formula can even be expressed by means of u's status indicator σ as introduced in the preceding section:

$$\widetilde{\pi}_{uv_i} = \pi_{uv_i} + (\sigma - 1)\Delta.$$

Let us, once again, consider the value by which the reduced cost π_{uv_i} has to be corrected, when u changes its status indicator from σ to σ'. As before, Δ_1 and Δ_2 denote the sum of dual adjustments that have been performed before and after the status change:

$$\widetilde{\pi}_{uv_i} = \pi_{uv_i} + (\sigma - 1)\Delta_1 + (\sigma' - 1)\Delta_2 = \pi_{uv_i} + (\sigma - \sigma')\Delta_1 + (\sigma' - 1)\Delta.$$

Thus, we would have to increase the reduced cost π_{uv_i} of each edge uv_i by $(\sigma - \sigma')\Delta$ whenever u changes its status. Observe that the offset of the surface blossom $\mathcal{B}$ containing u is increased by exactly this amount, and we can therefore attain the same result by computing the actual reduced cost with respect to that offset also:

$$\widetilde{\pi}_{uv_i} = \pi_{uv_i} + \mathit{offset}_{\mathcal{B}} + (\sigma - 1)\Delta. \tag{2.4}$$

Thus, we postulate that the actual reduced cost $\widetilde{\pi}_{uv_i}$ of any edge uv_i associated with u is computed by Formula (2.4).

One final remark must be made here. Imagine that at some point of time a new edge uv_{k+1} having actual reduced cost $\widetilde{\pi}_{uv_{k+1}}$ is to be added to u. The reduced cost $\pi_{uv_{k+1}}$ that is actually stored with the new pair $(\pi_{uv_{k+1}}, uv_{k+1})$ of u must then equal

$$\pi_{uv_{k+1}} = \widetilde{\pi}_{uv_{k+1}} - \mathit{offset}_{\mathcal{B}} - (\sigma - 1)\Delta. \tag{2.5}$$

We will therefore call $\pi_{uv_{k+1}}$ the *stored reduced cost* of the edge uv_{k+1}, also.

2.1.3 Managing the Blossom Offsets

In the preceding two sections several formulae have been developed to compute both the actual potential of a blossom and the actual reduced cost of edges associated with a vertex. In either case, the value of interest is obtained by taking the offset of the surface blossom into consideration. What remains to be shown is how one can handle these offsets when a shrink or an expand step occurs.

Managing the Blossom Offsets — Shrink Step

Let $\mathcal{B}$, with (immediate) subblossoms $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{2k+1}$, denote the blossom to be formed. Each odd labeled subblossom $\mathcal{B}_i$ is made even by adjusting its offset as previously described (see (2.3)). The 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 achieve a situation where the actual potential and also the actual reduced costs associated with each vertex $u \in \mathcal{B}_i$ can be computed with respect to the new surface blossom offset $\mathit{offset}_{\mathcal{B}}$.

The following strategy assures that the offsets of all (immediate) subblossoms $\mathcal{B}_i$, $1 \leq i \leq 2k+1$, are equally set to zero. The actual potential and reduced costs associated with any vertex contained in the new blossom $\mathcal{B}$ can thus be computed with respect to the offset $\mathit{offset}_{\mathcal{B}} = 0$.

When a surface blossom $\mathcal{B}'$ (trivial or non–trivial) becomes an even tree blossom for the first time during a phase, its offset $\mathit{offset}_{\mathcal{B}'}$ is set to zero. Thus, in order to preserve the validity of (2.1) and (2.2) 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:

$$\begin{aligned} y_u &= y_u + \mathit{offset}_{\mathcal{B}'}, \text{ and} \\ z_{\mathcal{B}'} &= z_{\mathcal{B}'} - 2\mathit{offset}_{\mathcal{B}'}. \end{aligned}$$

In the extended view mentioned in the preceding section, even tree vertices are also associated with a series of pairs. In this case, the stored reduced cost π_{uv} of each such pair (π_{uv}, uv) associated with u is subject to correction:

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

We wish to emphasize that the described updates are performed for every blossom that becomes an even tree blossom (an even tree blossom need not necessarily participate in a shrink step). We thus decided to call this strategy the *provident strategy.*

Observe that the adjustments are performed at most once for a fixed vertex per phase. Thus, the time required for the potential adjustments is $O(n)$ per phase. The correction of the reduced costs contributes time $O(m\, t_{\text{adj}})$ per phase, where t_{adj} denotes the time needed by the operation to change the stored reduced cost.[1] We will keep the pairs associated with a vertex in a priority queue (as will be explained in Section 3.4) and thus t_{adj} is bounded by $O(\log n)$. In summary, a total time bound of $O(n + m \log n)$ per phase results.

We wish to present another strategy, in which the new offset $\mathit{offset}_{\mathcal{B}}$ is determined by the offset value offset^* of the (immediate) subblossom $\mathcal{B}_i$ that survives in the following procedure.

We iterate over all (immediate) subblossoms of $\mathcal{B}$. Initially, offset^* is set to the offset of $\mathcal{B}_1$. In each stage i, $1 \leq i \leq 2k$, the actual potential of each vertex contained in a subblossom $\mathcal{B}_j$ with $j \leq i$ is computed with regard to the offset offset^*. Let $c_i = \sum_{j=1}^{i} |\mathcal{B}_j|$ denote the total number of these vertices. We use c_{i+1} to denote the number of vertices contained in the subblossom $\mathcal{B}_{i+1}$. When $c_i \geq c_{i+1}$, offset^* survives and all vertices of $\mathcal{B}_{i+1}$ lose; otherwise $\mathit{offset}_{\mathcal{B}_{i+1}}$ survives and all vertices of the $\mathcal{B}_j$'s lose. offset^* is set to the survivor offset and the other offset is denoted by offset_l (the actual potentials of all loser vertices are computed with respect to that offset). The potentials y_u of all loser vertices u are adjusted such that their actual potentials are computed correctly with respect to the offset offset^*, i.e.

$$y_u = y_u + \mathit{offset}_l - \mathit{offset}^*.$$

[1]The total number of pairs stored for all vertices will be bounded by $O(m)$.

As before, it is only in the extended view (where not only unlabeled and odd labeled vertices but also even tree vertices are associated with a series of pairs) that the stored reduced cost π_{uv} of each such pair (π_{uv}, uv) needs to be adjusted:

$$\pi_{uv} = \pi_{uv} + \mathit{offset}_l - \mathit{offset}^*.$$

We do not adjust the potentials $z_{\mathcal{B}_i}$ of the non–trivial subblossoms $\mathcal{B}_i$ here, since they in any case require a special treatment, as will be described below. We call this strategy the *non–provident* strategy, since the necessary adjustments are performed on demand, i.e. when the corresponding vertex in fact participates in a shrink step.

Let us proceed to the determination of the time needed by these adjustments. Since we always adjust the potentials and stored reduced costs of losing vertices only, the necessary adjustments for a fixed vertex u will be performed at most $O(\log n)$ times per phase.[2] The cost for updating the potential of u is $O(1)$. Moreover, since the number of edges associated with u will be no greater than its degree $\mathit{deg}(u)$, the time required to adjust the stored reduced costs is $O(\mathit{deg}(u)\, t_{\text{adj}})$, where t_{adj} denotes (as above) the time needed by the operation to change the stored reduced cost. We conclude that each vertex u contributes $O((1 + \mathit{deg}(u)\, t_{\text{adj}}) \log n)$ time per phase. As mentioned previously, t_{adj} is bounded by $O(\log n)$. Thus, summing over all vertices we obtain a total time bound of $O((n + m \log n) \log n) = O(n \log n + m(\log n)^2)$ per phase. That is, the theoretical time bound of $O(m \log n)$ per phase is exceeded. In practice, however, the non–provident strategy turned out to be slightly more efficient than the provident strategy (as will be presented in Section 3.6). It therefore seems to us that this strategy is worth being considered.[3] Our multiple search tree implementation (discussed in Section 3.4) implements both the provident and the non–provident strategy.

After the new blossom $\mathcal{B}$ has been formed, the actual potential $\widetilde{z}_{\mathcal{B}_i}$ of each non–trivial subblossom $\mathcal{B}_i$ is no longer affected by future dual adjustments. We therefore *freeze* the potential of these blossoms by adopting the following convention. For any non–trivial subblossom $\mathcal{B}_i$, we ensure that the potential $z_{\mathcal{B}_i}$ equals its actual potential $\widetilde{z}_{\mathcal{B}_i}$. Thus, at the time of shrinking we set:

$$z_{\mathcal{B}_i} = z_{\mathcal{B}_i} - 2\mathit{offset}_{\mathcal{B}_i} - 2\sigma\Delta,$$

and since every subblossom of $\mathcal{B}$ has been made even before, the above equation reduces to

$$z_{\mathcal{B}_i} = z_{\mathcal{B}_i} - 2\mathit{offset}_{\mathcal{B}_i} + 2\Delta.$$

The time needed to perform these potential freezings is proportional to the number of non–trivial subblossoms of $\mathcal{B}$.

[2] There are n vertices and after each adjustment for u, u will reside in a group of cardinality at least twice as large as before.

[3] Moreover, note that *all* stored reduced costs of a fixed vertex u are adjusted by the same amount. Therefore, the underlying priority queue data structure, which organizes these reduced costs, could also implement an operation, say *change_all_priorities*, which changes all priorities in the queue by the same amount in time $O(\mathit{deg}(u))$. For example, assume the priority queue is realized by a balanced binary tree (i.e. having height $O(\log(\mathit{deg}(u)))$), where the items of the priority queue correspond to the leaves of the tree. Then, traversing each vertex (i.e. non–leaf vertices too) of the tree and explicitly updating the stored priority of that vertex will take time proportional to $\mathit{deg}(u)$. As a consequence, the total time needed by the non–provident strategy would be reduced to $O((n+m)\log n)$ per phase, as desired.

Managing the Blossom Offsets — Expand Step

The details for an expand step now ensue easily. Let $\mathcal{B}$ denote the odd surface blossom that is going to be expanded. As for the shrink step, the (immediate) subblossoms of $\mathcal{B}$ are denoted by $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{2k+1}$. The actual potential and the actual reduced costs associated with each vertex contained in $\mathcal{B}$ can be computed by the formulae established above; these values depend on the offset $\mathit{offset}_{\mathcal{B}}$ of $\mathcal{B}$, which is therefore assigned to each of the subblossom offsets, i.e. $\mathit{offset}_{\mathcal{B}_i} = \mathit{offset}_{\mathcal{B}}$. Moreover, each subblossom becomes odd labeled and the potential $z_{\mathcal{B}_i}$ of all non–trivial subblossoms $\mathcal{B}_i$ undergo *unfreezing* with respect to the new offset:

$$z_{\mathcal{B}_i} = z_{\mathcal{B}_i} + 2\mathit{offset}_{\mathcal{B}_i} - 2\sigma\Delta$$

which equals

$$z_{\mathcal{B}_i} = z_{\mathcal{B}_i} + 2\mathit{offset}_{\mathcal{B}_i} - 2\Delta$$

since each $\mathcal{B}_i$ is labeled odd. Obviously, the time required to unfreeze the blossom potentials is proportional to the number of non–trivial subblossoms of $\mathcal{B}$. Afterwards, the necessary status changes for some of the subblossoms $\mathcal{B}_i$, for instance for those that leave T, can be handled by an offset adjustment as discussed above (see Equation (2.3)).

Summarizing, we have established a convenient way to handle the varying blossom potentials as well as the reduced costs of edges associated with a vertex. The value of interest can be computed on demand by the formulae developed. Here, making an offset available to each surface blossom and keeping track of the total amount Δ of dual adjustments turned out to be the key ideas. The additional overhead produced by the offset maintenance has been proved to consume $O(m \log n)$ time per phase. A dual adjustment by δ reduces to an increase of Δ by δ and can thus be performed in time $O(1)$.

2.2 Determination of δ — towards a Priority Queue Approach

We next consider more closely the idea of using priority queues to determine the value of δ, and show how all priorities stored in a priority queue can be adjusted in a uniform manner.

Recall that δ is chosen as the minimum of the four values $\delta_1, \delta_2, \delta_3$ and δ_4 (see Section 1.6.3). In order to determine each one of these we keep a corresponding priority queue *delta1*, *delta2*, *delta3* and *delta4*.[4] The priorities in each of the priority queues change with each dual adjustment and at first glance there seems to be little hope that this approach will turn out to be efficient. However, an essential observation is that

[4]We assume some familiarity with the *priority queue* data type. All standard operations, like *insert*, *delete min*, *find min* etc, are assumed to take time no more than $O(\log n)$, where n denotes the number of items stored in the priority queue. For a detailed discussion of these operations see, for example, Cormen et al. [CLR92].

one can arrange the priority queues such that all priorities decrease uniformly by the dual adjustment value δ. Consequently, the so–called *actual priority*, denoted by $\widetilde{p}$, of any item in each of the priority queues can be computed from its *stored priority* p, i.e. the priority which is stored with that item in the priority queue, and the total dual adjustment value Δ as defined above.

More precisely, we ensure that a priority p stored in any of these four priority queues corresponds to the actual priority $\widetilde{p} = p - \Delta$. As a consequence, when a new item having (actual) priority $\widetilde{p}$ has to be inserted into one of the priority queues, the priority p which is stored with that item is set to $\widetilde{p} + \Delta$, where Δ denotes the total dual adjustments performed up to this point. We next discuss the semantics of the items contained in each of the priority queues.

delta1 consists of all items $\langle p, u\rangle$ with $u^+ \in T$. The actual priority $p - \Delta$ corresponds to the actual potential $\widetilde{y}_u$ of u.

Each item $\langle p, uv\rangle$ in *delta2* represents the best edge of a vertex $v^{\{\varnothing|+\}} \notin T$ to an even labeled tree vertex $u^+ \in T$. The actual reduced cost $\widetilde{\pi}_{uv}$ of this edge equals the actual priority $p - \Delta$.

delta3 keeps track of the edges uv connecting two even labeled tree vertices $u^+ \in T$ and $v^+ \in T$. The edges inserted into *delta3* are ensured to be alive; however, during the course of the algorithm some of the edges stored in *delta3* might become dead. We use a *lazy–deletion* strategy for these edges: dead edges are simply discarded when they occur as the minimal item of *delta3*. Each edge is represented by an item $\langle p, uv\rangle$ in *delta3*. The actual priority $p - \Delta$ corresponds to one half of the actual reduced cost of uv, i.e. $p - \Delta = \widetilde{\pi}_{uv}/2$.

The priority queue *delta4* contains for each odd labeled non–trivial surface blossom $\mathcal{B}^- \in T$ an item $\langle p, \mathcal{B}\rangle$. The actual priority $p - \Delta$ of this item is equal to one half of the actual potential of $\mathcal{B}$: $p - \Delta = \widetilde{z}_{\mathcal{B}}/2$.

We briefly argue that all actual priorities of *delta1*, *delta2*, *delta3* and *delta4* decrease by the dual adjustment value δ. The potentials of all vertices $u^+ \in T$ are decreased by δ and therefore the actual priorities of *delta1* decrease by δ. Since only the potential of the endpoint $u^+ \in T$ for all edges uv stored in *delta2* is decreased by δ, each actual priority of *delta2* decreases by δ. For each edge uv stored in *delta3* the potential of both endpoints is decreased by δ. The actual priority of each edge is one half of the reduced cost of that edge. Therefore, each actual priority decreases by δ. Finally, the potential of each non–trivial surface blossom is reduced by 2δ, and its actual potential in *delta4* thus decreases by δ.

Let us suppose that the priority queues *delta1*, *delta2*, *delta3* and *delta4* are maintained correctly. Each value $\delta_1, \delta_2, \delta_3$ and δ_4 can then be determined by a *find min* operation on *delta1*, *delta2*, *delta3* and *delta4*, respectively. Moreover, finding the responsible vertex, edge or blossom reduces to a *delete min* operation on the priority queue from which δ results. Summarizing, both the determination of δ and also of the responsible vertex, edge or blossom can be achieved in time $O(\log n)$.[5]

[5]A comment is in order at this point. *delta3* might contain up to m items and thus an upper bound of $O(\log m)$ results. At first sight, it seems that the stated time bound of $O(\log n)$ is exceeded; but note that $m \leq n^2$, and therefore $O(\log m) = O(\log n)$.

2.3 A Misleading Strategy — Traps and Pitfalls

Having the outlined ideas in mind, it seems as if one could immediately implement the blossom-shrinking approach that guarantees the stated time bound of $O(m \log n)$ per phase. However, the realization described next will not fully comply with the stated time bound. The reasons for our decision to present a mistaken realization are substantiated by the following three arguments. First, we have been misled by this realization ourselves. Second, it will serve as a basis that can easily be extended to a correct approach. And finally, it will eventually provide us with an intuitive grasp such that we will affirm the need for concatenable priority queues.

An alternating tree T is grown from a free vertex r as described in Algorithm 1.6.3. Initially each vertex is a surface blossom having offset value 0, and the value of Δ is set to 0. The initial potential stored with each vertex equals its actual potential. At the beginning of a phase, each of the priority queues *delta1*, *delta2*, *delta3* and *delta4* is made empty.

For each vertex $u^{\{\varnothing|+\}} \notin T$ or $u^- \in T$ we keep track of its best edge uv to an even labeled tree vertex and of the (stored) reduced cost π_{uv} of that edge by means of a pair (π_{uv}, uv). Moreover, we adopt the convention that a designated pair $(\infty, \emptyset)$ is assigned to u, when no such edge has been encountered during the current phase.

Whenever a vertex u becomes an even tree vertex, its actual potential $\widetilde{y}_u$ is computed by (2.1) and a corresponding item $\langle \widetilde{y}_u + \Delta, u \rangle$ is inserted into *delta1*. We go through all incident edges uv of u in order to keep *delta2* and *delta3* correct; this will contribute $O(m)$ time per phase, since each vertex that becomes an even tree vertex will stay even and remain in T for the rest of the phase. When uv is dead or is a tree edge, it is simply discarded. Otherwise, uv is alive and we can thus compute its actual reduced cost:

$$\widetilde{\pi}_{uv} = \widetilde{y}_u + \widetilde{y}_v - w_{uv},$$

where $\widetilde{y}_v$ is obtained by Formula (2.1) and w_{uv} denotes the weight of that edge.[6] The action to be taken depends on the status of the endpoint v:

Case 1: $v^{\{\varnothing|+\}} \notin T$

We consider only the case where uv is the new best edge of v, since otherwise nothing has to be done.

When $(\infty, \emptyset)$ is the pair stored with v, uv will be the new best edge to v and we therefore replace that pair by (π_{uv}, uv), where the stored reduced cost π_{uv} is computed by Formula (2.5). A new item $\langle \widetilde{\pi}_{uv} + \Delta, uv \rangle$ is inserted into *delta2*.

Otherwise, let $(\pi_{\hat{u}v}, \hat{u}v)$ denote the pair stored with v. We can compute the actual reduced cost $\widetilde{\pi}_{\hat{u}v}$ of $\hat{u}v$ by Equation (2.4). When $\widetilde{\pi}_{uv} < \widetilde{\pi}_{\hat{u}v}$, uv will

[6] Recall that the reduced cost π_{uv} of an edge uv has been defined as repeated below, where in the current context the potentials refer to the actual potentials. Moreover, since uv is alive the sum over all blossom potentials containing that edge must equal 0.

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

be the new best edge of v. The pair stored with v is replaced by (π_{uv}, uv), where π_{uv} refers to the stored reduced cost (obtained by Equation (2.5)), and the item stored for v in *delta2* is replaced by $\langle \widetilde{\pi}_{uv} + \Delta, uv\rangle$.

Case 2: $v^+ \in T$
The item $\langle \widetilde{\pi}_{uv}/2 + \Delta, uv\rangle$ is simply inserted into *delta3*.

Case 3: $v^- \in T$
The same description as for $v^{\{\varnothing|+\}} \notin T$ applies, but no item is inserted or replaced in *delta2*.

When a non–tree blossom $\mathcal{B}$ enters T, each item in *delta2* corresponding to a vertex $u \in \mathcal{B}$ is deleted. Moreover, when $\mathcal{B}$ is non–trivial and becomes an odd tree blossom, we compute its actual potential $\widetilde{z}_{\mathcal{B}}$ by (2.2) and insert the item $\langle \widetilde{z}_{\mathcal{B}}/2 + \Delta, \mathcal{B}\rangle$ into *delta4*.

When a vertex $v^- \in T$ leaves T due to an expand step, we use the pair (π_{uv}, uv) stored with v in order to set up an item for *delta2*; if $(\infty, \emptyset)$ is assigned to v, we do nothing. The actual reduced cost $\widetilde{\pi}_{uv}$ of uv is computed as in (2.4), and the item $\langle \widetilde{\pi}_{uv} + \Delta, uv\rangle$ is inserted into *delta2*.

Determining δ, performing a dual adjustment and finding the responsible vertex, edge or blossom are achieved as previously described. All remaining details, e.g. shrinking or expanding a blossom etc., follow easily from the discussion above.

One final remark is in order at this point. Since an unlabeled vertex might become an odd tree vertex and then leave T again due to an expand step, one may wonder why it is sufficient to maintain for each vertex $v^{\varnothing} \notin T$ and $v^- \in T$ the best edge uv to a vertex $u^+ \in T$ only. Note, however, that once the best edge uv of a vertex $v^{\varnothing} \notin T$ has been used for a grow step, v will stay in T for the rest of the phase — even if all blossoms containing v are expanded during that phase.

So far, it seems that the running–time of $O(m \log n)$ per phase has been achieved. Observe, however, that each vertex entering or leaving T causes a deletion or insertion on *delta2*. In the rest of this section we will justify in detail the claim that it is due to the expansion of blossoms that these insertions and deletions may be executed up to $O(n^2)$ times and thus exceed the claimed time bound.
Readers who are not interested in these details are advised to skip to Section 2.4.

2.3.1 Maximum Height of a Blossom Tree

First, we need to introduce the concept of a so–called *blossom tree*, which represents the nesting of a blossom $\mathcal{B}$.

Let $\mathcal{B}$ be a blossom. Each subblossom $\mathcal{B}_i \subseteq \mathcal{B}$ corresponds to a node u_i in the blossom tree $BT_{\mathcal{B}}$ of $\mathcal{B}$.[7] The root node u of $BT_{\mathcal{B}}$ stands for the blossom $\mathcal{B}$ itself. Consider a node u_i in $BT_{\mathcal{B}}$ that corresponds to a subblossom $\mathcal{B}_i \subseteq \mathcal{B}$. The children of u_i in $BT_{\mathcal{B}}$ are the nodes $u_{i_1}, u_{i_2}, \ldots, u_{i_k}$, where each u_{i_j}, $1 \le j \le k$, corresponds to an immediate subblossom $\mathcal{B}_j$ of $\mathcal{B}_i$ (see Figure 2.1 for an example). From the construction of $BT_{\mathcal{B}}$ it follows that the vertices contained in $\mathcal{B}$ correspond to the leaves of $BT_{\mathcal{B}}$.

[7]In order to avoid confusion, we will use the term *node* when referring to vertices in the blossom tree.

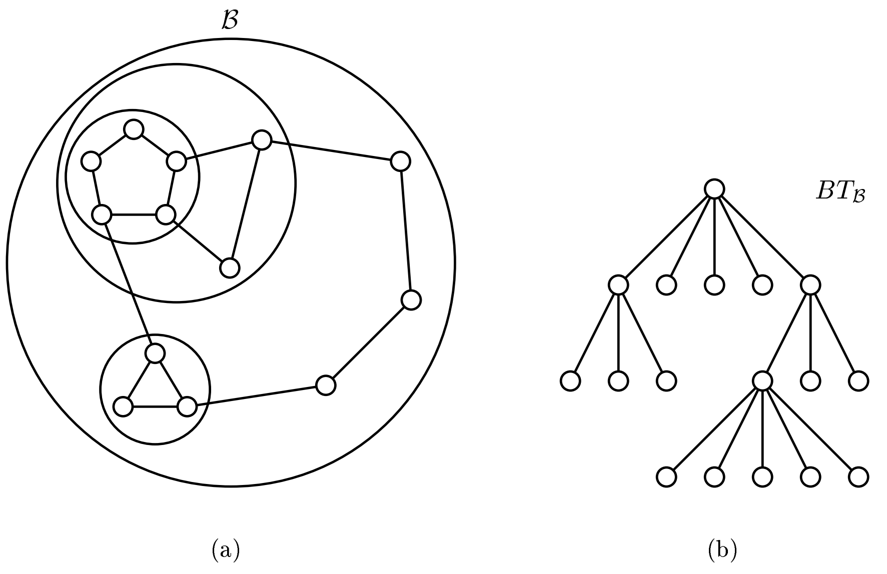

Figure 2.1: Let $\mathcal{B}$ be a blossom whose nesting structure is given in (a). The blossom tree $BT_\mathcal{B}$ of $\mathcal{B}$ is formed as depicted in (b). In $BT_\mathcal{B}$, the root node stands for $\mathcal{B}$, each internal (i.e. non–leaf) node represents a non–trivial subblossom of $\mathcal{B}$ and every leaf corresponds to a vertex of $\mathcal{B}$.

The *cardinality* p of a blossom $\mathcal{B}$ is defined as the number of vertices contained in $\mathcal{B}$. Furthermore, a blossom is said to be of *size* s, when it contains s immediate subblossoms.

Lemma 2.3.1 Let $\mathcal{B}$ be a blossom of cardinality p and $BT_\mathcal{B}$ the corresponding blossom tree. The height h of $BT_\mathcal{B}$ is bounded by $O(p)$.

Proof:
Let L_h denote the number of leaves in a blossom tree of height h. We have $L_0 = 1$, since a blossom tree of height zero represents a trivial blossom. A blossom tree of height one has at least three leaves: $L_1 \geq 3$.

Generally speaking, a blossom tree of height h has $L_h \geq L_{h-1} + 2$ leaves. The recursion can easily be solved:

$$L_h \geq L_{h-1} + 2 \geq L_{h-2} + 2 + 2 \geq \ldots \geq L_0 + \underbrace{2 + \ldots + 2}_{\text{h times}} = 2h + 1$$

Since the number of leaves in $BT_\mathcal{B}$ corresponds to the cardinality p of $\mathcal{B}$, we obtain: $h \leq (p-1)/2$. □

Lemma 2.3.1 implies that the height of a blossom tree $BT_\mathcal{B}$ to a blossom $\mathcal{B}$ having maximum cardinality may be $O(n)$.

In Section 1.6.5, the disadvantage of computing the reduced cost of an edge using the alternative linear programming formulation of the maximum–weight perfect matching problem was outlined.[8] Using the terms introduced above, the time required to compute the reduced cost of an edge uv connecting the blossoms $\mathcal{B}_u$ and $\mathcal{B}_v$ is proportional to the height of the blossom trees $BT_{\mathcal{B}_u}$ and $BT_{\mathcal{B}_v}$. That is, by Lemma 2.3.1, it takes time $O(p_u + p_v)$ in the worst case to compute the reduced cost of an edge uv, where p_u and p_v denote the cardinality of $\mathcal{B}_u$ and $\mathcal{B}_v$, respectively.

Applegate and Cook [App93] use the blossom tree to compute the reduced cost of an edge as follows. For each blossom tree of a surface blossom $\mathcal{B}$, they broadcast the sum of all blossom potentials along the path from the root to each leaf. Each leaf in any blossom tree $BT_{\mathcal{B}}$ that corresponds to a vertex u of $\mathcal{B}$ then knows the sum $\Sigma_u = y_u + \sum_{u \in \mathcal{B}_i} z_{\mathcal{B}_i}$. The reduced cost of an alive edge uv can then be computed by $\Sigma_u + \Sigma_v - w_{uv}$. However, broadcasting the sum of all potentials to each leaf in the blossom tree takes time $O(p)$ for a blossom having cardinality p.

In either case, a lower bound of $\Omega(n^2)$ per phase results for the algorithm based on the alternative formulation.

2.3.2 Expanding a Blossom — Number of Status Changes

Consider an odd tree blossom $\mathcal{B}^- \in T$ with cardinality p. When $\mathcal{B}$ is expanded, some subblossoms of $\mathcal{B}$ may leave T and later become odd tree blossoms. Following the strategy described above, each vertex of a blossom that leaves T is *touched*, e.g. in order to insert an appropriate item into *delta2*. We are now interested in the number of these touches per phase.

More precisely, let $e(p)$ denote the total number of status changes caused by vertices of $\mathcal{B}$ that leave T during a phase. Obviously, a blossom $\mathcal{B}$ with maximum nesting structure will form the most disadvantageous case. Or to put it differently, a blossom $\mathcal{B}$ whose blossom tree $BT_{\mathcal{B}}$ has largest height possible will contribute most to $e(p)$. Therefore, we consider $\widetilde{e}(p)$ which equals $e(p)$ in the worst case only, i.e. $e(p) \leq \widetilde{e}(p)$.

$\widetilde{e}(p)$ can easily be defined as a recursive function:

$$\widetilde{e}(p) = \begin{cases} 0 & \text{for } p \leq 1, \text{ and} \\ (p-2) + 1 + \widetilde{e}(p-2) & \text{otherwise.} \end{cases}$$

The recursion is substantiated as follows. Obviously, blossoms having cardinality $p = 1$ are trivial and thus cannot contribute anything to $\widetilde{e}(p)$. Otherwise, when a blossom $\mathcal{B}$ with cardinality $p > 1$ is expanded, at least one vertex (namely the base) must stay in T. A large subblossom of cardinality $p-2$ and a single vertex get unlabeled and thus contribute $(p-2)+1$ to $\widetilde{e}(p)$. Later, the large (sub)blossom might become an odd blossom of T and be expanded itself, producing cost $\widetilde{e}(p-2)$.

We are now interested in the number i of applications of the recursion stated above such that $\widetilde{e}(p - 2i) = 0$. Since by definition $\widetilde{e}(p) = 0$ for $p \leq 1$, we have $i = (p-1)/2$, which is the maximum height of the blossom tree $BT_{\mathcal{B}}$. We thus have:

$$\widetilde{e}(p) \quad = \quad \sum_{i=1}^{\frac{p-1}{2}} (p-2i) + 1 = \frac{p-1}{2}\left(p + 1 - \left(\frac{p-1}{2} + 1\right)\right) = \frac{p^2-1}{4}$$

[8]Recall that, using the alternative linear programming formulation, the reduced cost π_{uv} of an edge uv was defined differently:

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

At this point it becomes apparent that touching each vertex would increase the running–time to $O(n^2)$ per phase and hence destroy the claimed bound of $O(m \log n)$.

2.4 Concatenable Priority Queues

The problem indicated in the preceding section is overcome, however, using so–called *concatenable priority queues*. A concatenable priority queue supports all the usual priority queue operations plus the two additional operations specified below. The items in a queue are regarded as forming a sequence.

concat(*pq1*, *pq2*) concatenates the underlying sequences of *pq1* and *pq2*.
The resulting priority queue *pq* contains all items of *pq1* in their original order followed by all items of *pq2* in their original order.

split_at_item(*pq*, *it*) splits the sequence of *pq* at item *it* into *pq1* and *pq2*.
All items preceding the item *it* (inclusively) in *pq* then belong to *pq1* and all other items belong to *pq2*.

As we will show at the end of this section, both operations can be achieved in time $O(\log n)$.

Motivated by the fact that we cannot afford to insert an item into *delta2* for each vertex separately, one may think of inserting just one item for each surface blossom. Therefore, we assume that each surface blossom $\mathcal{B}$ maintains its own concatenable priority queue, which we will denote by $P_{\mathcal{B}}$.

The queue $P_{\mathcal{B}}$ of a non–tree blossom $\mathcal{B}^{\{\varnothing|+\}} \notin T$ or an odd tree blossom $\mathcal{B}^- \in T$ incorporates all pairs stored with the vertices $v \in B$. That is, every vertex $v \in \mathcal{B}$ has an item $\langle \pi_{uv}, uv \rangle$ in $P_{\mathcal{B}}$. The priority π_{uv} equals the stored reduced cost of the best edge uv connecting v to an even labeled tree vertex $u^+ \in T$. As before, the item in $P_{\mathcal{B}}$ corresponding to v may be set to $\langle \infty, \emptyset \rangle$ in order to indicate the non–availability of such an edge for v.
We also maintain a concatenable priority queue $P_{\mathcal{B}}$ for each even tree blossom $\mathcal{B}^+ \in T$. Again, for each vertex $v \in B$ we have a corresponding item in $P_{\mathcal{B}}$. However, the contents of these items is set arbitrarily.

Each non–tree blossom $\mathcal{B}^{\{\varnothing|+\}} \notin T$ sends its minimum item $\langle \pi_{uv}, uv \rangle$, representing the best edge uv (along all best edges) of $\mathcal{B}$, to *delta2*; however $\mathcal{B}$ does not send an item to *delta2* when the minimum item equals $\langle \infty, \emptyset \rangle$. More precisely, let $\langle p, uv \rangle$ denote the item in *delta2* that has been sent by $\mathcal{B}$. The actual priority $\widetilde{p} = p - \Delta$ then equals the actual reduced cost $\widetilde{\pi}_{uv}$ of the best edge uv of $\mathcal{B}$ (which can be computed by (2.4)).
Whenever a non–tree blossom $\mathcal{B}$ becomes a tree blossom, its corresponding item is deleted from *delta2*. Conversely, when a tree blossom leaves T, a corresponding item is inserted into *delta2*. Finally, when the minimum in a priority queue $P_{\mathcal{B}}$ with $\mathcal{B}^{\{\varnothing|+\}} \notin T$ changes, the corresponding item in *delta2* is updated accordingly.

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 $\mathcal{B}$. Here, we keep track of each t_i, $1 \leq i \leq 2k+1$, the last item in

$\mathcal{B}_i$. Later, when $\mathcal{B}$ is expanded, the priority queues to $\mathcal{B}_1, \mathcal{B}_2, \dots, \mathcal{B}_{2k+1}$ can easily be recovered by splitting the priority queue of $\mathcal{B}$ at each item t_i, $1 \leq i \leq 2k+1$. In fact, we handle the concatenable priority queues for even tree blossoms only because of structural reasons.
All other details apply as one would expect and as is described in Section 2.3.

Assuming that both operations *concat* and *split_at_item* take $O(\log n)$ time, the claimed time bound of $O(m \log n)$ per phase is achieved. In order to demonstrate this, recall that there will be at most $O(n)$ *concat* and *split_at_item* operations during a phase. Moreover, observe that only the maintenance of *delta3* uses time $O(m + n \log n)$ per phase.

In the rest of this section, a realization of the data type *concatenable priority queue* based on (a,b)*–trees* will be briefly outlined. We will concentrate mainly on the two additional operations *concat* and *split_at_item*. For a more detailed discussion, see Aho et al. [AHU74, Section 4.12] and Mehlhorn [Meh84, Section III.5.3].

Many realizations of the priority queue data type use a *balanced tree* as the underlying data structure, i.e. a tree T whose height is bounded by $O(\log n)$ where n denotes the number of leaves in T.

Definition 2.4.1 ((a,b)–Tree) Let $T = (V, E)$ denote a tree rooted at r. The number of outgoing edges of a vertex $u \in V$ is denoted by $outdeg(u)$. T is called an (a,b)*–tree*, with $b \geq 2a - 1$, iff the following holds:

(1) for each non–leaf vertex $u \in V$, with $u \neq r$: $outdeg(u) \geq a$,

(2) for each non–leaf vertex $u \in V$: $outdeg(u) \leq b$, and

(3) all leaves reside on the same level.

The next theorem states that each (a,b)–tree is balanced in the sense mentioned above.

Theorem 2.4.1 Let T be an (a,b)–tree of height h and let n denote the number of leaves in T. For the height h of T it is:

$$\log_b n \;\leq\; h \;\leq\; 1 + \log_a(n/2).^{9}$$

All standard operations of the priority queue data type can be implemented for (a,b)–trees as well. Each such operation will take time (at most) $O(\log n)$ (see the references mentioned above).

However, we want to emphasize a major difference to the common view of an (a,b)–tree T that represents a priority queue. Normally, the leaves are arranged in ascending order (e.g. from left to right) with respect to the priority of an item. However, the kind of concatenable priority queue we specified above requires the leaves to be part of a sequence (independently of their priority).

Operation: *concat*(*pq1*, *pq2*)

Let T_1 and T_2 be the two (a,b)–trees of *pq1* and *pq2* having height h_1 and h_2, respectively. When $h_1 = h_2$, a new root vertex r is created and T_1 and T_2 become the left and right child of r.

[9] The right side can be understood by observing that the root r has $outdeg(r) \geq 2$ and every other non–leaf vertex u has $outdeg(u) \geq a$ and thus $n \geq 2a^{h-1}$.

Now assume $h_1 > h_2$; the other case is treated analogously. Let v denote the rightmost vertex in T_1 of height $h_1 - h_2$ and let f denote the parent of v. The root of T_2 becomes the rightmost child of f. If, afterwards, f has more than b children the tree is *repaired* as we suppose to be known (f is split into two vertices having $\lceil (b+1)/2 \rceil$ and $\lfloor (b+1)/2 \rfloor$ children etc).

Lemma 2.4.1 Let *pq1* and *pq2* denote two priority queues having n_1 and n_2 items. A *concat*(*pq1*, *pq2*) operation can be performed in time $O(|\log n_1 - \log n_2|)$.

Operation: *split_at_item*(*pq*, *it*)

Consider the tree T that corresponds to the priority queue *pq*. Let v denote the leaf vertex in T that stores the item *it* and let p denote the path from v up to the root vertex of T. By deleting p, T decomposes into two forests; a forest F_l to the left which consists of all trees with leaves to the left of v (inclusively) and a forest F_r to the right which consists of all trees with leaves to the right of v (exclusively). Let $F_l = (LT_i, LT_{i-1}, \ldots, LT_1)$ be the ordered sequence (from left to right) of all trees to the left and $F_r = (RT_1, RT_2, \ldots, RT_j)$ the ordered sequence (from left to right) of all trees to the right. Iteratively concatenating all trees LT_k, with $k = 1, 2, \ldots, i$, results in a tree T_1 which represents *pq1*. Analogously, the tree T_2 of *pq2* can be constructed by concatenating all trees RT_k in the order $k = 1, 2, \ldots, j$.

Lemma 2.4.2 Let *pq* denote a priority queue containing n items. A *split_at_item*(*pq*, *it*) operation for any item *it* of *pq* takes time $O(\log n)$.

Proof:
When p is deleted from T, there will be at most $b-1$ trees of a fixed height h in $F_l \cup F_r$. The only exception are trees of height 0, of which there can be b many. The concatenation of b trees of height h will result in a tree having height at most $h+1$. Therefore, at most b trees of any given height occur during the concatenation process described above.
To concatenate two trees of the same height takes time $O(1)$, and $O(\Delta h)$ when their heights differ by Δh. Therefore, the time spent concatenating all trees is $O(bh + \log \Delta h_{\max})$, where $\Delta h_{\max}$ denotes the maximum difference of any two trees that are concatenated. Since $\Delta h_{\max}$ is bounded by $O(\log n)$, *split_at_item* takes $O(\log n)$ time. $\square$

Chapter 3

Implementation and Tests

One major objective in the field of theoretical computer science is to obtain algorithms that are efficient with respect to the theoretical running–time. However, not seldom there is a big trade–off between a theoretically efficient algorithm and its technical feasibility. The utilization of complex data structures that make the algorithm fast in theory often has a drastic impact on its efficiency in practice.

In this chapter we will present an implementation of Edmonds' blossom–shrinking approach based on the use of *concatenable priority queues.* At the time we started, it was not foreseeable whether the implementation would be fast in practice. Moreover, a highly efficient algorithm for the maximum–weight perfect matching problem was available such that there was little hope of improving upon it. The algorithm referred to is known as the *Blossom IV* algorithm and is implemented in C. It is due to Cook and Rohe [CR97] and is based on earlier work by Applegate and Cook [App93]. Cook and Rohe do not claim a theoretical time bound, but it will be no better than $\Omega(n^3)$.

Our implementation is innovative in the sense that there is no other algorithm using priority queues accessible at the moment. We used C++ as the programming language since the algorithm uses and is intended to become part of the **L***ibrary of* **E***fficient* **D***ata Structures and* **A***lgorithms*, called LEDA for short, developed at the Max–Planck Institute for Computer Science in Saarbrücken, Germany.[1] We assume that the reader is familiar with some basic data types and concepts of LEDA; most of the data types used in our implementation will be self–explanatory.

We implemented two versions of Algorithm 1.6.3. A so–called *single search tree* approach and a *multiple search tree* approach. In the former, only one tree is grown at a time, whereas in the latter various search trees are grown concurrently. Surprisingly, the difference between these two approaches with regard to their practical efficiency is immense. Both algorithms guarantee a worst–case running–time of $O(nm \log n)$.

The chapter is organized as follows. In Section 3.1 we will define the interface functions and outline their functionality. Then, the data structure *concat_pq*, which realizes the

[1] For an extensive reference describing all issues concerning LEDA see the book by Mehlhorn and Näher [MN99]. LEDA is freely available for academic research and teaching at:

`http://www.mpi-sb.mpg.de/LEDA.`

data type *concatenable priority queue* discussed in Section 2.4, is introduced. Since we do not want to go into the implementation details of this data structure, all operations needed will be specified in Section 3.2. After this, our implementation of the single search tree algorithm will be presented. The ensuing differences for the multiple search tree algorithm are the subject of Section 3.4. The efficiency of both algorithms is considerably improved by constructing a better initial solution, as will be outlined in Section 3.5 below. Finally, some running–time experiments will reveal the efficiency of our implementation in practice.

3.1 Functionality

The function

```
list<edge> MAX_WEIGHT_MATCHING(const ugraph &G,
                               const edge_array<NT> &w,
                               bool check, int heur)
```

returns a maximum–weight matching and

```
list<edge> MAX_WEIGHT_PERFECT_MATCHING(const ugraph &G,
                                       const edge_array<NT> &w,
                                       bool check, int heur)
```

a maximum–weight perfect matching for the undirected graph G (type *ugraph*) with weight function w. Both functions accept edge weights of any number type NT.[2] The matching is represented by the list of edges returned. When *check* is set to true, the optimality of the computed matching is checked internally. Depending on the value of *heur*, the algorithm starts either with an empty matching ($heur = 0$), a *greedy matching* ($heur = 1$), or with a *jump–start* or *fractional matching* ($heur = 2$), as will be explained in Section 3.5.

The interface functions are specified in the header file `MWM.t`. The user can switch between the single search tree and the multiple search tree approach by defining the token `_SST_APPROACH`.[3]

⟨*MWM.t: maximum–weight matching algorithm*⟩≡

```
template<class NT>
list<edge> MAX_WEIGHT_MATCHING(const ugraph &G,
                               const edge_array<NT> &w,
                               bool check = true, int heur = 1) {
  edge_array<NT> w_mod(G);
  ⟨scale edge weights⟩
  node_array<NT>  pot(G);
  node_array<int> b(G, -1);
```

[2]We suppose, however, that the number type NT provides a division operation.

[3]That is, in order to use the single search tree approach, type `#define _SST_APPROACH` before the file `MWM.t` is included.

```
    array<two_tuple<NT, int> > BT;

    total_t = used_time();
  #if defined(_SST_APPROACH)
    list<edge> M = MWM_SST(G, w_mod, pot, BT, b, heur, false);
  #else
    list<edge> M = MWM_MST(G, w_mod, pot, BT, b, heur, false);
  #endif
    total_t = used_time(total_t);

    check_t = used_time();
    if (check) CHECK_MAX_WEIGHT_MATCHING(G, w_mod, M, pot, BT, b);
    check_t = used_time(check_t);

    return M;
  }
```

We compute a maximum–weight matching with respect to a modified weight function *w_mod* which equals w unless the number type NT is *int*, where *w_mod* equals $4w$.

⟨*scale edge weights*⟩≡

```
  edge e;
  bool INT = LEDA_TYPE_ID(NT) == INT_TYPE_ID;
  forall_edges(e, G) w_mod[e] = (INT ? 4*w[e] : w[e]);
```

Thus, the dual solution and also the reduced cost of each edge will remain integral during the course of the algorithm (see also Lemma 1.6.1).[4]

When the _SST_APPROACH token has been defined, the function

```
  list<edge> MWM_SST(const ugraph &G, const edge_array<NT> &w,
                     node_array<NT> &pot, array<two_tuple<NT, int> > &BT,
                     node_array<int> &b, int heur, bool perfect)
```

is called. Its implementation will be the subject of Section 3.3. The implementation details of the function

```
  list<edge> MWM_MST(const ugraph &G, const edge_array<NT> &w,
                     node_array<NT> &pot, array<two_tuple<NT, int> > &BT,
                     node_array<int> &b, int heur, bool perfect)
```

will be the subject of Section 3.4. Both functions compute a perfect matching iff *perfect* is set to true. The total time needed (in CPU seconds) to compute an optimal matching is stored in a global variable *total_t* (type *float*).

The additional parameters *pot*, b and BT are used to prove optimality of the computed matching M. Their semantics is as follows. The potential of each vertex is stored in the *node_array pot*. BT represents the nested family of odd cardinality sets (see Section 1.3). Each *two_tuple* $(z_{\mathcal{B}}, p_{\mathcal{B}})$ in BT represents a non–trivial blossom $\mathcal{B}$ having potential $z_{\mathcal{B}}$ and *parent index* $p_{\mathcal{B}}$. The parent index $p_{\mathcal{B}}$ is set to -1 if $\mathcal{B}$ is a surface blossom. Otherwise, $p_{\mathcal{B}}$ stores the index of the entry corresponding to the immediate

[4]Apparently, the same purpose could have been achieved by a multiplication by two. However, the edge weights are multiplied by four so as to ensure that one–half the reduced cost of any edge remains integral too.

superblossom of $\mathcal{B}$.[5] The index range of BT is $[0, \ldots, k-1]$, where k denotes the number of non–trivial blossoms. When $\mathcal{B}_i$ is a subblossom of $\mathcal{B}$, the index of the entry corresponding to $\mathcal{B}_i$ is smaller than the one of $\mathcal{B}$. The parent index for a vertex u is stored in the *node_array* b.

Using this data, the function

```
void CHECK_MAX_WEIGHT_MATCHING(const ugraph &G,
                               const edge_array<NT> &w,
                               const list<edge> &M,
                               const node_array<NT> &pot,
                               const array<two_tuple<NT, int> > &BT,
                               const node_array<int> &b)
```

can check all optimality conditions given in Section 1.6. The time (in CPU seconds) needed by the checker is kept in a global variable *checker_t* (type *float*). We will not discuss the realization of that function and instead refer to Mehlhorn and Näher [MN99].

The interior of the function that computes a maximum–weight perfect matching looks similar.

⟨*MWM.t: maximum–weight perfect matching algorithm*⟩≡

```
template<class NT>
list<edge> MAX_WEIGHT_PERFECT_MATCHING(const ugraph &G,
                                       const edge_array<NT> &w,
                                       bool check = true, int heur = 1) {
  edge_array<NT> w_mod(G);
  ⟨scale edge weights⟩
  node_array<NT>  pot(G);
  node_array<int> b(G, -1);
  array<two_tuple<NT, int> > BT;
  total_t = used_time();
#if defined(_SST_APPROACH)
  list<edge> M = MWM_SST(G, w_mod, pot, BT, b, heur, true);
#else
  list<edge> M = MWM_MST(G, w_mod, pot, BT, b, heur, true);
#endif
  total_t = used_time(total_t);
  check_t = used_time();
  CHECK_MAX_WEIGHT_PERFECT_MATCHING(G, w_mod, M, pot, BT, b);
  check_t = used_time(check_t);
  return M;
}
```

[5]The *immediate superblossom* concept is defined analogously to the immediate subblossom concept given in Section 1.3.

3.2 Concatenable Priority Queues (concat_pq)

We implemented a data structure *concat_pq* supporting all needed operations of data type *concatenable priority queue* as introduced in Section 2.4. The implementation is based on (a, b)–trees; we chose $a = 2$ and $b = 16$. *concat* and *split_at_item* are essentially realized as discussed at the end of Section 2.4. We do not intend to go into the implementation details. Instead, the specification of all operations needed in the subsequent sections is given.

In Section 1.7 we outlined the idea of using a *union–find* data structure with *split* operation to handle the surface graph. The method we use in our implementation is different. Since a concatenable priority queue will be assigned to each surface blossom, we decided to extend the functionality of *concat_pq* such that it also enables the maintenance of the surface graph. We use the underlying (a, b)–trees to identify a setable object (which will be the surface blossom) of a given item (which will correspond to a vertex). The way this is achieved is as follows. Each root of an (a, b)–tree stores a pointer to the object representing that tree. Traversing from an item *it* towards the root, we can identify the (a, b)–tree object containing the item *it*. Moreover, each (a, b)–tree object has a generic pointer *owner* (a generic pointer is of type *void∗*) which is setable by the user; see operation *set_owner*. Consequently, the *owner* of any item *it* can be identified (operation *get_owner*) in time $O(\log n)$, where n denotes the number of items in the (a, b)–tree.

1. Definition

An instance Q of the parameterized data type *concat_pq*<P, I> is a collection of items (type *c_pq_item*). Every item contains a priority from a linearly ordered type P and an information from an arbitrary type I. We use $\langle p, i\rangle$ to denote a *c_pq_item* with priority p and information i. The data structure requires a designated element *infinity* of P, with *infinity* $\geq p$ for all $p \in P$ and equality holds only if $p =$ *infinity*. An item $\langle p, i\rangle$ with $p =$ *infinity* is *irrelevant* to Q. The number of items in Q is called the *size* of Q. Q is *empty* when all its items are irrelevant, or when Q has size zero. A setable generic pointer *owner* (type *void∗*) is associated with Q.

2. Creation

concat_pq<P, I> Q;	creates an instance Q of type *concat_pq*<P, I> based on the linear order defined by the global compare function *compare*(*const* P&, *const* P&) and initializes it with the empty priority queue. *infinity* is set to the maximum value of type P.

3. Operations

c_pq_item Q.init(P p, I i)	initializes Q to the priority queue containing only the item $\langle p, i\rangle$ and returns that item.

P	*Q*.prio(*c_pq_item it*)	returns the priority of item *it*. *Precondition*: *it* is an item in *Q*.
I	*Q*.inf(*c_pq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in *Q*.
void	*Q*.concat(*concat_pq*<*P*, *I*>& *pq*, *int dir* = *LEDA*::*after*)	concatenates *Q* with *pq*. The items in *Q* precede (succeed) the items of *pq*, when *dir* = *after* (*dir* = *before*). *pq* is made empty, i.e. contains no items thereafter.
void	*Q*.split_at_item(*c_pq_item it*, *concat_pq*& *pq1*, *concat_pq*& *pq2*)	splits *Q* at item *it* into *pq1* and *pq2* such that *it* is the last item of *pq1*. In case *it* = *nil*, *pq2* becomes *Q* and *pq1* becomes empty. The instance *Q* is empty thereafter, unless it is given as one of the arguments.
c_pq_item	*Q*.find_min( )	returns an item with minimal priority (*nil* if *Q* is empty).
P	*Q*.del_min( )	makes the item *it* = *Q.find_min*() irrelevant to *Q* by setting its priority to *infinity*. The former priority is returned.
void	*Q*.del_item(*c_pq_item it*)	makes the item *it* irrelevant to *Q*. *Precondition*: *it* is an item in *Q*.
bool	*Q*.decrease_p(*c_pq_item it*, *P x*)	makes *x* the new priority of item *it*. The function returns *true* iff the operation was successful, i.e. *Q.prio*(*it*) was larger than *x*.
bool	*Q*.increase_p(*c_pq_item it*, *P x*)	makes *x* the new priority of item *it*. The function returns *true* iff the operation was successful, i.e. *Q.prio*(*it*) was smaller than *x*.
int	*Q*.size( )	returns the size of *Q*.
bool	*Q*.empty( )	returns *true*, if *Q* is empty, and *false* otherwise.
void	*Q*.reset( )	makes *Q* the empty priority queue by setting all priorities to *infinity*.
void	*Q*.clear( )	makes *Q* the empty priority queue by deleting all items.
void	*Q*.set_owner(*GenPtr pt*)	sets *owner* of *Q* to the object pointed to by the generic pointer *pt* (type *void*∗).

4. Friend Functions

GenPtr	get_owner(*c_pq_item it*)	returns the generic pointer *owner* of the instance containing item *it*.

5. Iteration

forall_items(it, Q) { "the items of Q are successively assigned to it" }

forall(i, Q) { "the information parts of the items of Q are successively assigned to i" }

6. Implementation

All access operations take time O(1). *concat* and *split_at_item* take time $O(\log n)$, where n is the (maximum) number of elements in the priority queue(s). Operations *clear* and *reset* take time $O(n)$. All other operations take time (at most) $O(\log n)$.

3.3 Single Search Tree Approach

In Chapter 1, we elaborated a generic algorithm (see Algorithm 1.6.3) of Edmonds' blossom–shrinking approach. Most of the details for its realization based on priority queues have been discussed in Chapter 2. In this section, the results established are integrated into a single search tree implementation using priority queues.

A major task in implementing the blossom–shrinking approach is concerned with the representation of blossoms. We will first design a template class *blossom* (type *blossom*<*NT*>) that keeps all necessary information, and turn to the implementation of our algorithm afterwards.

3.3.1 Data Structures

In Chapter 2 we justified extensively the need for each surface blossom to maintain its own concatenable priority queue.

Given the parameterized data type *concat_pq*<*P*, *I*> as specified in the last section, the template class *blossom* can be defined as follows.

⟨*SST.t: data structures*⟩≡
```
template<class NT> class vertex;
template<class NT> class blossom;
⟨class blossom: friend functions — definition⟩
template<class NT>
class blossom : public virtual concat_pq<NT, vertex<NT>*> {
  ⟨class blossom: friend functions — declaration⟩
public:
  ⟨class blossom: data members⟩
```

```
    ⟨class blossom: member functions⟩
    LEDA_MEMORY(blossom<NT>);
  };
```

Class *blossom* inherits all properties of the data type *concat_pq<NT, vertex<NT>* >*. Additional data members and functions will be defined below. The information part of each item points to an object of class *vertex*. Essentially, the template class *vertex* comprises all data associated with a vertex. For the single search tree approach we have:

⟨*SST.t: data structures*⟩+≡

```
  template<class NT> class vertex {
  public:
    NT   pot;
    node my_node;
    node best_adj;
    vertex(NT d, node u) {
      pot = d;
      my_node = u;
      best_adj = nil;
    }
    LEDA_MEMORY(vertex<NT>);
  };
```

Each object of type *vertex<NT>* stores its potential *pot* and its original vertex *my_node*. The way we keep track of the best edge for *my_node* to an even tree vertex is by storing this adjacent vertex in *best_adj*.

Data Members: A blossom is either even labeled, odd labeled or unlabeled. Therefore, a new type *LABEL* is defined.

```
  typedef enum {even, odd, unlabeled} LABEL;
```

Moreover, each blossom maintains its potential *pot* and its offset *offset*.

⟨*class blossom: data members*⟩≡

```
  LABEL label;
  NT    pot;
  NT    offset;
```

The base and mate (if any) vertex of a blossom are stored in *base* and *mate*, respectively. Note that these two vertices represent the endpoints of the matching edge. In order to organize the tree structure of the alternating tree, each odd blossom keeps track of its *discovery* and *predecessor* vertex *disc* and *pred*. *disc* denotes the endpoint of the non–matching tree edge which is contained in the blossom and *pred* refers to the other endpoint (see Figure 3.1). We wish to emphasize that these data will only be kept correctly for surface blossoms.

⟨*class blossom: data members*⟩+≡

```
  node base, mate;
  node disc, pred;
```

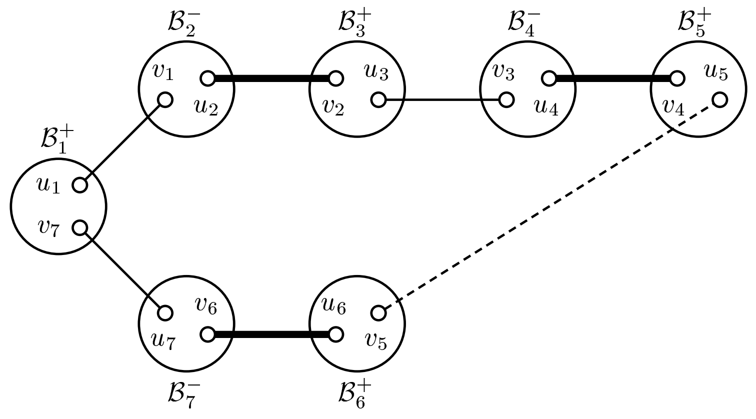

Figure 3.1: Consider the alternating tree T with root blossom $\mathcal{B}_1$ as depicted above. Let *Bi* denote the object of type *blossom<NT>* representing the blossom $\mathcal{B}_i$, $1 \le i \le 7$. For example, the entries of *B3* are set to *base* $= v_2$, *mate* $= u_2$ and *disc* $=$ *pred* $=$ *nil* and for *B4* we have *base* $= u_4$, *mate* $= v_4$, *disc* $= v_3$ and *pred* $= u_3$. The entries of all other blossoms are set accordingly. Using this data, the (surface) tree path from each tree blossom to the root blossom can be identified easily. $\mathcal{B}_1, \ldots, \mathcal{B}_7$ define a new blossom $\mathcal{B}$. Let B be the object corresponding to $\mathcal{B}$. We have: *B.shrink_path* $= <u_1, v_1, u_2, v_2, \ldots, u_7, v_7>$ and *B.subblossom_p* $= <\&B2, \&B3, \ldots, \&B7, \&B1>$.

Additionally, each non–trivial blossom stores its defining surface cycle as a list of vertices in *shrink_path*. All pointers of the immediate subblossom objects are collected in a list *subblossom_p*.

⟨*class blossom: data members*⟩+≡

```
list<node>          shrink_path;
list<blossom<NT>*> subblossom_p;
```

We adopt the following order for the entries of these lists. Let $C = (e_1, e_2, \ldots, e_{2k+1})$ denote the defining surface cycle of a blossom $\mathcal{B}$. Each edge $e_i = (u_i, v_i)$, $1 \le i \le 2k+1$, of C is regarded as being directed such that v_{i-1} and u_i, with $v_0 = v_{2k+1}$, are contained in the same immediate subblossom $\mathcal{B}_i$. Assume further that $\mathcal{B}_1$ refers to the subblossom containing the base. Then, *shrink_path* stores the list of vertices $<u_1, v_1, u_2, v_2, \ldots u_{2k+1}, v_{2k+1}>$ and *subblossom_p* consists of the pointers $<\&B_2, \&B_3, \ldots, \&B_{2k+1}, \&B_1>$, where $\&B_i$ denotes the pointer to the blossom object that represents $\mathcal{B}_i$. Refer to Figure 3.1 for an example.

At the time of shrinking, the *split_item* entry of each (immediate) subblossom is set to the last item of that blossom. This will enable restoration of the concatenable priority queues in an expand step later on.

⟨*class blossom: data members*⟩+≡

```
c_pq_item           split_item;
```

There are some additional data members that will be introduced in the context they are first needed.

Member Functions: We outline only some of the member functions; the remaining functions will be filled in as needed. A function that returns the (stored) reduced cost of the best edge (of all best edges incident to vertices) of a blossom is implemented as follows.

⟨*class blossom: member functions*⟩≡

```
const NT min_prio() const
{ return (find_min() ? prio(find_min()) : INFINITY(NT)); }
```

Here, *INFINITY*(*NT*) simply returns the maximum value of type *NT*. The following three functions return the appropriate entries stored in the information part of an item *it* (type *c_pq_item*).

⟨*class blossom: member functions*⟩+≡

```
const NT   pot_of  (c_pq_item it) const { return inf(it)->pot;      }
const node node_of (c_pq_item it) const { return inf(it)->my_node;  }
const node best_adj(c_pq_item it) const { return inf(it)->best_adj; }
```

We must also provide a function to test whether or not a blossom is trivial: each blossom containing just one item is said to be trivial.

⟨*class blossom: member functions*⟩+≡

```
bool trivial() const { return size() == 1; }
```

Friend Functions: Given the item *it* of a vertex, the following function will return a pointer to the blossom object containing that vertex.

⟨*class blossom: friend functions — declaration*⟩≡

```
friend blossom<NT>* blossom_of<NT>(c_pq_item it);
```

The function will also be used for testing whether or not two vertices are contained in the same blossom.

When a new blossom object is created, we call *set_owner*(*pt*), where *pt* is the generic pointer of the new blossom object. Consequently, we can later cast the pointer returned by *get_owner*(*it*) to a pointer of type *blossom<NT>∗*.

⟨*class blossom: friend functions — definition*⟩≡

```
template<class NT> blossom<NT>* blossom_of(c_pq_item it) {
  return (it ? (blossom<NT>*)(get_owner(it)) : nil);
}
```

Constructor: We are now in a position to define the constructor of the class *blossom*. As an optional argument, the base vertex *b* of the blossom to be created can be given.

⟨*class blossom: member functions*⟩+≡

```
blossom(node b = nil) : concat_pq<NT, vertex<NT>*>() {
  set_owner(leda_cast(this));
  label = even;
```

```
    pot = offset = 0;
    base = b; mate = nil;
    disc = pred = nil;
    marker1 = marker2 = 0;
    item_in_T  = item_in_O = nil;
    item_in_pq = nil;
    split_item = nil;
  }
```

leda_cast(*this*) simply casts the blossom pointer *this* to a generic pointer. The meaning of the missing data members *marker1*, *marker2*, *item_in_T*, *item_in_O* and *item_in_pq* will become clear shortly.

We also define a friend function that provides a more convenient way of constructing and initializing a trivial blossom object. The function

⟨*class blossom: friend functions — declaration*⟩+≡

```
  friend c_pq_item new_blossom<>(NT d, node b, blossom<NT>* &B);
```

creates a new blossom object that consists only of the vertex b having potential d. Afterwards, B points to this new blossom object and the *c_pq_item* of the item corresponding to b is returned.

⟨*class blossom: friend functions — definition*⟩+≡

```
  template<class NT> c_pq_item new_blossom(NT d, node b, blossom<NT>* &B) {
    B = new blossom<NT>(b);
    vertex<NT> *v = new vertex<NT>(d, b);
    return B->init(INFINITY(NT), v);
  }
```

Note that we do not use the data member *pot* of the blossom class to store the potential of b — in fact, that data member is only used to maintain the potential of a non–trivial blossom. The priority of the item corresponding to b is set to $INFINITY(NT)$ indicating that currently no best edge is available.

3.3.2 Algorithm

Let us turn to the implementation of Algorithm 1.6.3 that realizes the ideas outlined in Chapter 2. The algorithm maintains the lower bounds $\delta 1, \ldots, \delta 4$ by the following data structures.

⟨*local variables*⟩≡

```
  NT                          delta1;
  NT                          delta2a;
  p_queue<NT, blossom<NT>*> delta2b;
  p_queue<NT, edge>         delta3;
  p_queue<NT, blossom<NT>*> delta4;

  node resp_d1;
  edge resp_d2a;
```

delta1 keeps track of the minimum (stored) potential of an even tree vertex that has been encountered. *delta1* is only used in the non–perfect matching case. The (actual) value of *delta2a* represents the (actual) reduced cost of the best edge of all best edges from an even non–tree vertex to an even tree vertex. The responsible vertex or edge of *delta1* and *delta2a* is stored in *resp_d1* and *resp_d2a*, respectively.[6] An item $\langle p, pt\rangle$ in *delta2b* represents the best edge of an unlabeled blossom pointed to by pt. The (actual) priority of p equals the (actual) reduced cost of this edge. Each edge e that is a candidate for a shrink step has an item $\langle p, e\rangle$ in *delta3*. The (actual) priority of p equals one half of the (actual) reduced cost of edge e. *delta4* contains one item $\langle p, pt\rangle$ for each non–trivial tree blossom. pt is a pointer to the blossom object and the (actual) priority of p equals one half of the (actual) potential of that blossom.
In Chapter 2 (Section 2.3 and Section 2.4) we have discussed the semantics of each item in any of those priority queues in more detail; we will not repeat that discussion here.

A counter *Delta* is used to accumulate the total sum of dual adjustments that have been performed up to the current stage of the algorithm.

$\langle$*local variables*$\rangle{+}{\equiv}$
```
  NT Delta = 0;
```

We need a mechanism to identify the *pq_item* in *delta2b* or *delta4* corresponding to a blossom object. Therefore, we add the following data member to the *blossom* class.

$\langle$*class blossom: data members*$\rangle{+}{\equiv}$
```
  pq_item item_in_pq;
```

Whenever a blossom sends an item to *delta2b* or *delta4*, the corresponding *pq_item* is stored in *item_in_pq* of that blossom.

When a vertex becomes an even tree vertex, its incident edges will be scanned; it may happen that several vertices become even tree vertices at once. Therefore, all new even tree vertices are collected in a queue Q which is realized by a singly linked list of vertices (type *node_slist*).

$\langle$*local variables*$\rangle{+}{\equiv}$
```
  node_slist Q(G);
```

At the end of a phase the current alternating tree is destroyed and all priorities in each concatenable priority queue need to be reset to *infinity*. We therefore accumulate in T all pointers to surface blossoms that are part of the current alternating tree; and in O the pointers of all unlabeled surface blossoms (outside of T) that are adjacent to any even tree blossom.

$\langle$*local variables*$\rangle{+}{\equiv}$
```
  list<blossom<NT>*> T;
  list<blossom<NT>*> O;
```

[6]In Chapter 2, we assumed that each even tree vertex has an item in a priority queue *delta1*; and that each even non–tree surface blossom has an item in a priority queue *delta2*. However, it is sufficient in fact to keep track of the data as described above: initiating the corresponding step for *resp_d1* or *resp_d2a* will terminate the current phase.

Since we must be able to identify a blossom's item (type *list_item*) in T or O, each blossom object stores this *list_item* in the data member *item_in_T* or *item_in_O*, respectively.

⟨*class blossom: data members*⟩+≡

```
list_item item_in_T;
list_item item_in_O;
```

An array *item_of* (type *node_array<c_pq_item>*) is used to identify the corresponding item (type *c_pq_item*) for each vertex of G.

⟨*local variables*⟩+≡

```
node_array<c_pq_item> item_of(G);
```

Finally, we introduce some variables that are used frequently. M is a list of edges and will be used to represent the resulting matching. The maximum value of number type NT is stored in $INFTY$.

⟨*local variables*⟩+≡

```
edge        e;
node        resp, opst, cur, adj, u, v, r;
blossom<NT> *RESP, *OPST, *CUR, *ADJ, *R;
list<edge>  M;
const NT INFTY = INFINITY(NT);
```

The overall structure of the algorithm computing either a maximum–weight matching or a maximum–weight perfect matching is as follows.

⟨*SST.t: algorithm*⟩≡

```
template<class NT>
list<edge> MWM_SST(const ugraph &G, const edge_array<NT> &w,
                   node_array<NT> &pot, array<two_tuple<NT, int> > &BT,
                   node_array<int> &b, int heur = 1, bool perfect = false) {
  ⟨local variables⟩
  ⟨initialization⟩
  forall_nodes(r, G) {
    R = _BLOSSOM_OF(r);
    if (R->mate) continue;
    ⟨clear priority queues and Q⟩
    R->status_change(even, Delta, T, Q);
    bool terminate = false;
    while (!terminate) {
      ⟨scan all edges of vertices in Q⟩
      ⟨determine lower bounds cand2b, ... , cand4⟩
      if (delta2a == Delta) {
        ⟨augment step using best edge of delta2a⟩
        terminate = true;
      }
      else if (delta1 == Delta) {
          ⟨alternate step using best node of delta1⟩
          terminate = true;
        }
        else if (cand2b == Delta) {
          ⟨grow step using best edge of delta2b⟩
        }
        else if (cand3 == Delta) {
          ⟨shrink step using best edge of delta3⟩
        }
        else if (cand4 == Delta) {
          ⟨expand step using best blossom of delta4⟩
        }
        else {
          ⟨dual adjustment⟩
        }
      }
    }
    ⟨extract matching and checker information⟩
    return M;
  }
```

In a main loop, we iterate over all vertices r of G. When the blossom R of r is unmatched, i.e. the *mate* of R equals *nil*, a phase is initiated. We define the following macro in order to more elegantly retrieve the pointer to the blossom object containing a given vertex.

⟨*SST.t: data structures*⟩+≡

```
#define _BLOSSOM_OF(this_node) \
(this_node ? blossom_of<NT>(item_of[this_node]) : nil)
```

At the beginning of each phase, *delta1*, …, *delta4* are reset and the queue Q is made empty.

⟨*clear priority queues and Q*⟩≡

```
delta1 = delta2a = INFTY;
delta2b.clear(); delta3.clear(); delta4.clear();
Q.clear();
```

R is made an even tree blossom calling the member function

```
void status_change(LABEL l, NT Delta, list<blossom<NT>*> &T, node_slist &Q)
```

the implementation of which will be given shortly. In the inner while loop, we first scan all edges incident to vertices of Q in order to maintain *delta1*, …, *delta4* correctly. Then, the minimum priorities of *delta2b*, *delta3* and *delta4* are determined and stored in *cand2b*, *cand3* and *cand4*, respectively.

⟨*determine lower bounds cand2b, … , cand4*⟩≡

```
NT cand2b = (delta2b.empty() ? INFTY : delta2b.prio(delta2b.find_min()));
NT cand3  = (delta3.empty()  ? INFTY : delta3.prio(delta3.find_min()));
NT cand4  = (delta4.empty()  ? INFTY : delta4.prio(delta4.find_min()));
```

Thereafter, it is checked if any of those lower bounds equals zero.[7] If so, the appropriate step is initiated. Note that between two consecutive dual adjustments, different steps may have to be executed. We adopt the convention that all grow steps precede a single shrink step, and, further, all shrink steps precede any expand step. This seems reasonable to us, since extending the tree in grow steps is rather cheap compared to a shrink step. Moreover, an expand step turned out to be more costly than a shrink step. Finally, when we can choose between either an augment step or an alternate step, we prefer the augment step. An augment step will decrease the number of free vertices by two, whereas an alternate step produces only a decrease by one. When neither of these steps can be executed, a dual adjustment is performed in order to progress.

⟨*dual adjustment*⟩≡

```
NT delta = leda_min(delta1,
                    leda_min(delta2a,
                             leda_min(cand2b,
                                      leda_min(cand3, cand4))));
if ((delta == INFTY) && perfect) return M;   // return empty matching
Delta = delta;   // corresponds to Delta += (delta - Delta)
```

A phase terminates, when either the case *delta1* $==$ *Delta* (this never happens in the perfect matching case) or *delta2a* $==$ *Delta* has occurred; or *delta* can be chosen as *INFTY* and thus no perfect matching exists (in the perfect matching case only). In the latter case, we return the empty matching.

Initialization: Depending on the value assigned to the argument *heur*, the algorithm uses a different method to compute an initial matching and the vertex potentials.

⟨*initialization*⟩≡

```
int free = G.number_of_nodes();
node_array<node> mate(G, nil);
switch(heur) {
  case 0:  { ⟨empty matching⟩ break; }
  case 1:  { ⟨greedy matching⟩ break; }
  default: { ⟨jump start matching⟩ break; }
}
```

All three methods use the *node_array pot* to store the initial potential of each vertex and a *node_array mate* to represent the matching. *free* denotes the number of free vertices.

When the initial matching leaves no vertex unmatched it will be optimal and we can immediately return it.

[7] Remember that the actual priorities are computed by subtracting *Delta* from the stored priorities. Therefore, an actual priority equals zero iff the stored priority equals *Delta*.

⟨*initialization*⟩+≡

```
if (free == 0) {
  ⟨prepare solution⟩
  return M;
}
```

Otherwise, for each vertex u of G a trivial blossom CUR with potential $pot[u]$ is constructed; the corresponding *c_pq_item* is stored in *item_of*$[u]$. When *mate*$[u]$ is different from *nil*, the data member *mate* of CUR is set to the vertex *mate*$[u]$ and CUR gets unlabeled.

⟨*initialization*⟩+≡

```
forall_nodes(u, G) {
  item_of[u] = new_blossom<NT>(pot[u], u, CUR);
  if (mate[u]) {
    CUR->mate = mate[u];
    CUR->label = unlabeled;
  }
}
```

Starting with an empty matching, the potentials are set as outlined in Section 1.6.3.

⟨*empty matching*⟩≡

```
forall_nodes(u, G) {
  if (outdeg(u) == 0) { pot[u] = 0; continue; }
  NT max = -INFTY;
  forall_adj_edges(e, u) max = leda_max(w[e], max);
  pot[u] = max/2;
}
```

For each vertex u we determine the value *max* which is maximum along all edge weights of incident edges. The potential of u is set to $max/2$; $pot[u]$ is set to zero, when no incident edge exists.

The construction of a greedy matching or a jump start matching will be the subject of Section 3.5. For now, it shall be sufficient to regard

⟨*greedy matching*⟩≡

```
free = greedy_matching(G, w, pot, mate, perfect);
```

and

⟨*jump start matching*⟩≡

```
free = jump_start(G, w, pot, mate, perfect);
```

as black–boxes that return the appropriate data in *pot*, *mate* and *free*.

In case all vertices are matched after the initial matching has been computed, M can be constructed as shown below.

⟨*prepare solution*⟩≡

```
forall_edges(e, G) {
  u = source(e);
  v = target(e);
  if (mate[u] && (mate[u] == v) &&
      mate[v] && (mate[v] == u))
    M.push(e);
}
```

For each edge e, it is checked whether the endpoints are the mates of each other, or not; if so, e is a matching edge and added to M. Observe that the information needed to check optimality is set correctly: *pot* contains the potential of each vertex; BT is empty and all entries in the *node_array* b are set to -1, indicating that each vertex is a surface blossom.

Performing a Status Change: We next outline the member function of class *blossom* that realizes a status change.

⟨*class blossom: member functions*⟩+≡

```
void status_change(LABEL l, NT Delta, list<blossom<NT>*> &T, node_slist &Q) {
  if (l == unlabeled) {
    ⟨make unlabeled non–tree blossom⟩
  }
  else if (l == odd) {
    ⟨make odd tree blossom⟩
  }
  else if (l == even) {
    ⟨make even tree blossom⟩
  }
  label = l;
}
```

The new status of the blossom object is determined by the given label l. The function needs to adjust the *offset* of the blossom by some *Delta* as developed in Section 2.1. Moreover, T and Q must be maintained correctly.

When a blossom gets unlabeled, its *offset* is adjusted depending on its current label *label* and the list item *item_in_T* of that blossom is deleted from T. Note that only tree blossoms can get unlabeled.

⟨*make unlabeled non–tree blossom*⟩≡

```
assert((label != l) && item_in_T);
offset += (label == odd ? Delta : -Delta);
T.del(item_in_T);
item_in_T = nil;
```

We only allow non–tree blossoms to become odd tree blossoms.[8] The *offset* is decreased by *Delta* and its pointer is added to T.

[8]Why a blossom can change its status from being an even labeled non–tree blossom to an odd labeled tree blossom will become apparent when the realization of an augment step is inspected more closely.

⟨*make odd tree blossom*⟩≡

```
assert((label != l) && !item_in_T);
offset -= Delta;
item_in_T = T.append(this);
```

Consider the case where a blossom becomes an even tree blossom. When its *label* is *odd*, we only need to update the *offset*. Otherwise, the blossom is a non–tree blossom. The *offset* gets adjusted and its pointer is added to T.

⟨*make even tree blossom*⟩≡

```
assert((label != l) || !item_in_T);
if (label == odd) offset += 2*Delta;
else {  // non-tree blossom
  offset += Delta;
  item_in_T = T.append(this);
}
⟨append all vertices to Q⟩
```

There is something more to do here: all vertices contained in the blossom object must be appended to Q. Therefore, we iterate over all items *it* of the blossom and append the vertex of this item (i.e. *node_of*(*it*)) to Q. Some preparatory work is done as well.

⟨*append all vertices to Q*⟩≡

```
c_pq_item it;
forall_items(it, *this) {
  inf(it)->pot += offset;
  Q.append(node_of(it));
}
if (!trivial()) pot -= 2*offset;
offset = 0;
```

The potential of each vertex contained in the blossom and the potential of the blossom itself (if non–trivial) is adjusted such that these actual potentials are computed correctly with respect to the new offset *offset* $= 0$. Since every vertex of the blossom has to be inspected anyway, it is reasonable to use the provident strategy (see Section 2.1.3) at this point.

Scanning New Even Vertices: For all vertices that have been added to Q, i.e. the vertices that have recently become even tree vertices, the incident edges have to be scanned in order to keep the data in the global priority queues *delta1* to *delta4* as well as the reduced cost of edges associated with vertices correctly.

⟨*scan all edges of vertices in Q*⟩≡

```
NT cur_pot, adj_pot, actual_p, stored_p;
while (!Q.empty()) {
  cur     = Q.pop();
  CUR     = _BLOSSOM_OF(cur);
  cur_pot = compute_potential(CUR, Delta, item_of[cur]);
  if (!perfect) {
    ⟨try to improve delta1⟩
    }
    forall_adj_edges(e, cur) {
      adj = opposite(cur, e);
      ADJ = _BLOSSOM_OF(adj);
      ⟨discard dead and tree edges⟩
      adj_pot  = compute_potential(ADJ, Delta, item_of[adj]);
      actual_p = cur_pot + adj_pot - w[e];
      ⟨prune edges⟩
      if ((ADJ->label == even) && !ADJ->item_in_T) {
        ⟨new delta2a edge encountered⟩
      }
      else if (ADJ->label == unlabeled) {
        ⟨new delta2b edge encountered⟩
      }
      else if (ADJ->label == even)  // ADJ is even tree blossom
        delta3.insert(actual_p/2 + Delta, e);
      else if (ADJ->label == odd) {
        stored_p = actual_p - ADJ->offset;
        ADJ->improve_connection(item_of[adj], stored_p, cur);
      }
    }
  }
```

For each vertex *cur* of Q we compute its actual potential *cur_pot* calling the template function *compute_potential*, which is essentially a realization of the Formulae (2.1) and (2.2) of Section 2.1. The function can also be asked to compute the actual potential of a blossom *CUR* by setting *it* to *nil*.

⟨*SST.t: helpers*⟩≡

```
  template<class NT>
  NT compute_potential(blossom<NT> *CUR, NT Delta, c_pq_item it = nil) {
    int a = (it == nil ? -2 : 1);
    int sigma = 0;
    if (CUR->item_in_T) sigma = (CUR->label == even ? -1 : 1);
    NT stored = (it == nil ? CUR->pot : CUR->pot_of(it));
    return stored + a * CUR->offset + a * sigma * Delta;
  }
```

When *cur_pot* is the new minimum potential of all even tree vertices, *delta1* and *resp_d1* are set accordingly (only in non–perfect matching case).

⟨*try to improve delta1*⟩≡

```
  if (cur_pot < delta1 - Delta) {
    delta1 = cur_pot + Delta;
    resp_d1 = cur;
    // if (delta1 == Delta) break;
  }
```

When the new (actual) value of *delta1* equals zero, i.e. *delta1* == *Delta*, we could also immediately break the scan step. The alternate step for *resp_d1* would decrease the number of free vertices by one and terminate the phase. However, we defer this step and complete the scanning procedure instead. The reason for doing so is that *delta2a* might also get decreased to zero and the corresponding augment step will then decrease the number of free vertices by two.

All edges e incident to *cur* are considered. The potential of each adjacent vertex *adj* is computed so as to enable the computation of the actual reduced cost *actual_p* of e. However, dead edges or tree edges are discarded:

⟨*discard dead and tree edges*⟩≡

```
// do not consider edges within a blossom
if (CUR == ADJ) continue;

// do not consider tree edges
if ((ADJ->label == odd) &&
    ((ADJ->base == adj && ADJ->mate == cur) ||
     (ADJ->disc == adj && ADJ->pred == cur))) continue;
```

Moreover, we use a *pruning strategy*. Since a phase terminates when the stored priority of *delta1* (in non–perfect case only) or *delta2a* equals *Delta*, we can discard all edges e whose stored priority exceeds the minimum value of *delta1* and *delta2a*. We will say e is *hopeless*. Note, in the case where e is a candidate edge for a shrink step, its stored priority in *delta3* will equal $actual\_p/2 + Delta$.[9]

⟨*prune edges*⟩≡

```
#if !defined(_NO_PRUNING)
if ((ADJ->label == even) && ADJ->item_in_T) {
  if (actual_p/2 + Delta > leda_min(delta1, delta2a)) continue;
}
else if (actual_p + Delta > leda_min(delta1, delta2a)) continue;
#endif
```

Depending on the status of *ADJ*, four cases are distinguished. First, when *ADJ* is an even non–tree blossom, a new *delta2a* edge has been encountered.

⟨*new delta2a edge encountered*⟩≡

```
if (actual_p < delta2a - Delta) {
  delta2a  = actual_p + Delta;
  resp_d2a = e;
  if (delta2a == Delta) break;
}
```

We check whether e is the new best edge of *delta2a*; if necessary, we update the value of *delta2a* and set *resp_d2a*, accordingly. In case where the new (actual) value of *delta2a* equals zero, we break the scan step immediately.

Second, when *ADJ* is an unlabeled non–tree blossom, a new *delta2b* edge has been encountered.

[9] The user can switch off the pruning strategy by defining the token `_NO_PRUNING` (`#define _NO_PRUNING`) before the file `MWM.t` is included.

⟨*new delta2b edge encountered*⟩≡
```
stored_p = actual_p - ADJ->offset + Delta;
if (ADJ->improve_connection(item_of[adj], stored_p, cur))
  if (ADJ->item_in_pq)
    delta2b.decrease_p(ADJ->item_in_pq, actual_p + Delta);
  else {
    ADJ->item_in_pq = delta2b.insert(actual_p + Delta, ADJ);
    ADJ->item_in_O = O.append(ADJ);
  }
```

The stored reduced cost *stored_p* of e is computed according to (2.5) (as described in Section 2.1.2). We check whether e is the new best edge of *adj* to an even labeled tree vertex and, if so, update the data of the corresponding item by the following member function:

⟨*class blossom: member functions*⟩+≡
```
bool improve_connection(c_pq_item it, NT x, node u) {
  if (!it) return false;
  NT old_min = min_prio();
  if (decrease_p(it, x)) inf(it)->best_adj = u;
  return old_min != min_prio();
}
```

When the new priority x is less than the one currently stored with *it*, i.e. function *decrease_p* returns *true*, the *best_adj* entry is set to u. The function returns *true* iff the minimum priority of the blossom object has changed.

In case e is the new best edge of ADJ, we either decrease the corresponding item in *delta2b* (if there exists any), or insert an appropriate one. In the latter case, ADJ is additionally inserted into O.

The last two cases are easy. For an even tree blossom ADJ, an appropriate item is simply inserted into *delta3*; and for an odd tree blossom ADJ, we call the member function *improve_connection* as described above.

Alternate Step: We come to the alternate step which is initiated when the actual value of the minimum item in *delta1* equals zero. Remember that this will never happen in the perfect matching case.

⟨*alternate step using best node of delta1*⟩≡
```
RESP = _BLOSSOM_OF(resp_d1);
RESP->base = resp_d1;
alternate_path(RESP, item_of);
⟨destroy alternating tree T⟩
RESP->label = even;
```

First, the surface blossom $RESP$ of *resp_d1* is retrieved. The alive edges along the even length path from $RESP$ to the root blossom of the tree are alternately unmatched and matched. $RESP$ will become free. The *base* of $RESP$ is set to *resp_d1*, since that vertex has (actual) potential zero and thus is allowed to stay unmatched. The current alternating tree gets destroyed (as will be described below). Thereafter, $RESP$ will be an unlabeled non–tree blossom. Remember, however, that free non–tree blossoms are supposed to be even. Therefore, *RESP*'s label is corrected to even (we do not call *status_change*).

Next, we describe the function *alternate_path* which alternates the alive edges along the tree path from *RESP* to the root blossom of the tree. More precisely, each matching edge along this path will become non–matching and each non–matching edge becomes matching. Recall that matching edges are represented by means of the data members *base* and *mate* of class blossom. The function will be reused in the augment step below.

⟨*SST.t: helpers*⟩+≡

```
template<class NT>
void alternate_path(blossom<NT>* RESP, node_array<c_pq_item> &item_of) {
  if (!RESP) return;
  blossom<NT> *CUR = RESP;
  node pred = RESP->base, disc = nil, mate;
  while (CUR) {
    if (CUR->label == even) {
      mate = CUR->mate;
      CUR->mate = disc;
      CUR->base = pred;
      CUR = _BLOSSOM_OF(mate);
    }
    else {  // CUR->label == odd
      pred = CUR->pred;
      disc = CUR->disc;
      CUR->mate = pred;
      CUR->base = disc;
      CUR = _BLOSSOM_OF(pred);
    }
  }
}
```

Starting at $CUR = RESP$, we follow the tree path towards the root. We keep the following invariants: *pred* and *disc* denote the predecessor and discovery vertex, respectively, of the odd blossom which has been considered most recently; initially, *pred* is set to the base of *RESP* and *disc* is set to *nil*. For an even labeled tree blossom *CUR*, we store the former mate in *mate* and set its data members *mate* and *base* to *disc* and *pred*, respectively. After this, the blossom to be inspected next is retrieved by using the former mate information stored in *mate*. When *CUR* is an odd tree blossom, *pred* and *disc* are set, and the *mate* and *base* data members of *CUR* are set accordingly. The next blossom to consider is the blossom of *pred*.

Augment Step: When the actual reduced cost of *resp_d2a* equals zero, the current (surface) matching is augmented. We need to determine the two surface blossoms *RESP* and *OPST* of the endpoints of edge $e = resp\_d2a$.

⟨*determine RESP and OPST of e*⟩≡

```
resp = source(e);
opst = target(e);
RESP = _BLOSSOM_OF(resp);
OPST = _BLOSSOM_OF(opst);
if (!OPST->item_in_T) {
  leda_swap(resp, opst);
  leda_swap(RESP, OPST);
}
// invariant: OPST is tree blossom
```

OPST denotes the blossom that is contained in the alternating tree T. First, the even non–tree blossom *RESP* is made an odd tree blossom (here we need to allow an even non–tree blossom to become an odd tree blossom); its *pred* and *disc* entries are set appropriately.

⟨*augment step using best edge of delta2a*⟩≡

```
e = resp_d2a;
⟨determine RESP and OPST of e⟩
RESP->status_change(odd, Delta, T, Q);
RESP->pred = opst;
RESP->disc = resp;
alternate_path(RESP, item_of);
⟨destroy alternating tree T⟩
```

Then, the edges along the tree path from *RESP* (traversing *OPST*) towards the root blossom of T are alternated calling *alternate_path*. Finally, the alternating tree T gets destroyed, as described next.

Destroy Tree: T stores all pointers to the surface blossoms contained in the alternating tree. For each such blossom *CUR*, we reset the priorities of all items to *infinity* by calling the member function *reset* (see Section 3.2), and perform a status change: *CUR* gets unlabeled.

⟨*destroy alternating tree T*⟩≡

```
forall(CUR, T) {
  if (CUR->label == odd) {
    CUR->disc = CUR->pred = nil;
    CUR->item_in_pq = nil;
  }
  CUR->reset();
  CUR->status_change(unlabeled, Delta, T, Q);
}
T.clear();
```

For an odd tree blossom *CUR*, the data members *disc* and *pred* as well as *item_in_pq* have to be set to *nil*. Finally, T is made empty.

Every unlabeled non–tree blossom that is adjacent to any even tree blossom contains at least one item whose priority differs from *infinity*. All those items need to be reset (to *infinity*).

⟨*destroy alternating tree T*⟩+≡

```
forall(CUR, O) {
  CUR->reset();
  CUR->item_in_pq = nil;
  CUR->item_in_O = nil;
}
O.clear();
```

The pointers of all unlabeled surface blossoms adjacent to any even tree blossom have been collected in O. Therefore, we call the member function *reset* for each surface blossom pointed to by an entry *CUR* of O; *nil* gets assigned to *CUR*'s data members *item_in_pq* and *item_in_O*. Alternatively, one could also delete each connection from a vertex contained in *CUR* to an even tree vertex separately. However, it turned out that calling *reset* once for each such surface blossom is more efficient. O is afterwards made empty.

Grow Step: The implementation of a grow step is as follows. First, we retrieve the unlabeled non–tree blossom *RESP* having actual priority zero in *delta2b*.

⟨*grow step using best edge of delta2b*⟩≡

```
RESP = delta2b.inf(delta2b.find_min());
delta2b.del_item(RESP->item_in_pq);
RESP->item_in_pq = nil;
```

The item of *RESP* in *delta2b* is deleted and *item_in_pq* is set to *nil*. The best edge of *RESP* is stored with the minimum item. Using the member functions of class *blossom*, it is not difficult to obtain *resp* and *opst*, the two endpoints of that edge.

⟨*grow step using best edge of delta2b*⟩+≡

```
c_pq_item best = RESP->find_min();
resp = RESP->node_of(best);
opst = RESP->best_adj(best);
```

The vertex *resp* is contained in *RESP*, and *opst* denotes the even labeled vertex in the alternating tree. *RESP* becomes an odd tree vertex having *opst* and *resp* as predecessor and discovery vertex, respectively.

⟨*grow step using best edge of delta2b*⟩+≡

```
RESP->status_change(odd, Delta, T, Q);
RESP->pred = opst;
RESP->disc = resp;
```

RESP is deleted from O and its data member *item_in_O* is set to *nil*, since it is no longer an unlabeled non–tree blossom; notice that *RESP* must have an item in O.

⟨*grow step using best edge of delta2b*⟩+≡

```
O.del_item(RESP->item_in_O);
RESP->item_in_O = nil;
```

We do not need to delete the connection stored with *resp* from *RESP*. This will be done when the tree gets destroyed later on.

Finally, when $RESP$ is a non–trivial blossom, an item representing $RESP$ and one half of the value of its potential is inserted into *delta4*.

⟨*grow step using best edge of delta2b*⟩+≡

```
if (!RESP->trivial())
  RESP->item_in_pq =
    delta4.insert(compute_potential(RESP, Delta)/2 + Delta, RESP);
```

The mate blossom $MATE$ of $RESP$ is also added to T. $MATE$ becomes an even tree blossom.

⟨*grow step using best edge of delta2b*⟩+≡

```
node mate = RESP->mate;
blossom<NT> *MATE = _BLOSSOM_OF(mate);
MATE->status_change(even, Delta, T, Q);
if (MATE->item_in_pq) {
  delta2b.del_item(MATE->item_in_pq);
  MATE->item_in_pq = nil;
  O.del_item(MATE->item_in_O);
  RESP->item_in_O = nil;
}
```

When $MATE$ has an item in *delta2b* it must be removed; we also delete its item from O.

Shrink Step: A shrink step is more complex. The minimum item in *delta3* containing the new tight edge e is deleted and the blossoms $RESP$ and $OPST$ containing the endpoints *resp* and *opst* of e are determined (as described above).

⟨*shrink step using best edge of delta3*⟩≡

```
e = delta3.inf(delta3.find_min());
delta3.del_min();
⟨determine RESP and OPST of e⟩
if (RESP == OPST) continue;  // dead edge encountered;
```

In case e is dead, i.e. $RESP$ and $OPST$ refer to the same blossom, we simply discard e and continue with the main algorithm. Otherwise, we have to determine the lowest common ancestor blossom LCA of $RESP$ and $OPST$ as well as the shrink path, i.e. the defining odd length surface cycle, of the new blossom.

⟨*shrink step using best edge of delta3*⟩+≡

```
blossom<NT>       *LCA;
list<node>         P1, P2;
list<blossom<NT>*> sub1, sub2;
⟨determine LCA and shrink path of RESP and OPST⟩
```

The code chunk which implements this will be presented shortly. For the time being, assume *sub1* and *P1* correspond to the lists *subblossom_p* and *shrink_path* of the new blossom as described in Section 3.3.1. We construct a new surface blossom $SUPER$ whose base and mate equal those of LCA. Note that the priority queue of $SUPER$

is empty. Its actual potential is set to zero (the stored potential must hence equal $-2Delta$); and *P1* is assigned to its data member *shrink_path*.

⟨*shrink step using best edge of delta3*⟩+≡

```
blossom<NT> *SUPER = new blossom<NT>(LCA->base);
SUPER->pot         = -2*Delta;
SUPER->mate        = LCA->mate;
SUPER->shrink_path = P1;
```

Subsequently, the priority queues of all subblossoms *CUR* of *SUPER* are concatenated one after another, calling the member function *append_subblossom* discussed below. When *CUR* is an odd tree blossom and has sent an item to *delta4*, this item is deleted. Finally, *SUPER* is added to the list of *T*.

⟨*shrink step using best edge of delta3*⟩+≡

```
forall(CUR, sub1) {
  if (CUR->item_in_pq) {
    delta4.del_item(CUR->item_in_pq);
    CUR->item_in_pq = nil;
  }
  SUPER->append_subblossom(CUR, Delta, T, Q);
}
SUPER->item_in_T = T.append(SUPER);
```

We next need to fill in details of the member function *append_subblossom* which helps to concatenate the subblossoms.

⟨*class blossom: member functions*⟩+≡

```
void append_subblossom(blossom<NT> *CUR, NT Delta,
                       list<blossom<NT>*> &T, node_slist &Q) {
  if (CUR->label == odd)
    CUR->status_change(even, Delta, T, Q);
  if (!CUR->trivial())
    CUR->pot += -2*CUR->offset + 2*Delta;
  T.del(CUR->item_in_T);
  CUR->item_in_T = nil;
  concat(*CUR);
  CUR->split_item = last_item();
  subblossom_p.append(CUR);
}
```

Each odd subblossom is made even by calling the member function *status_change*. In case *CUR* is non–trivial, its potential gets frozen as explained (in Section 2.1.3). *CUR* is deleted from *T*, since it is no longer a surface blossom. The priority queue of *CUR* gets concatenated to that of the blossom object by calling the inherited function *concat*. *split_item* of *CUR* is set to the last item of the resulting priority queue (which is the last item of *CUR*), and *CUR* is appended to the *subblossom_p* list of the current blossom object.

Determination of the Lowest Common Ancestor: We will determine the lowest common ancestor blossom of *RESP* and *OPST* by traversing the two tree paths towards the root in a *lock–step* fashion.[10]

We introduce an additional counter *lock*, which is initially set to zero and will be incremented each time a lowest common ancestor has to be determined; since *lock* might get incremented up to n^2 times, type *double* has been chosen (in order to prevent an overflow as might occur for type *int*).

⟨*local variables*⟩+≡

```
double lock = 0;
```

Moreover, each blossom occupies two markers called *marker1* and *marker2*.

⟨*class blossom: data members*⟩+≡

```
double marker1, marker2;
```

The way we determine the lowest common ancestor is as follows. We traverse the tree paths from *RESP* and *OPST* towards the root. For each even blossom *CUR1* on the first path (starting with *RESP*), we set *marker1* to *lock*; and for each even tree blossom *CUR2* on the second path (starting with *OPST*), we set *marker2* to *lock*. The lowest common ancestor blossom is encountered when either *marker2* of *CUR1* or *marker1* of *CUR2* equals *lock*.

⟨*determine LCA and shrink path of RESP and OPST*⟩≡

```
blossom<NT> *CUR1 = RESP, *CUR2 = OPST;
CUR1->marker1 = CUR2->marker2 = ++lock;

P1.push(resp); P2.push(opst);

while (CUR1->marker2 != lock && CUR2->marker1 != lock &&
       (CUR1->mate != nil || CUR2->mate != nil)) {
  if (CUR1->mate) {
    sub1.push(CUR1);
    P1.push(CUR1->base); P1.push(CUR1->mate);
    CUR1 = _BLOSSOM_OF(CUR1->mate);
    sub1.push(CUR1);
    P1.push(CUR1->disc); P1.push(CUR1->pred);
    CUR1 = _BLOSSOM_OF(CUR1->pred);
    CUR1->marker1 = lock;
  }
  if (CUR2->mate) {
    sub2.push(CUR2);
    P2.push(CUR2->base); P2.push(CUR2->mate);
    CUR2 = _BLOSSOM_OF(CUR2->mate);
    sub2.push(CUR2);
    P2.push(CUR2->disc); P2.push(CUR2->pred);
    CUR2 = _BLOSSOM_OF(CUR2->pred);
```

[10] A trivial method to determine the lowest common ancestor of *RESP* and *OPST* is as follows. Starting at *RESP* we trace the tree path up to the root, marking each traversed blossom. After this, following the tree path from *OPST*, the first marked blossom we meet will be the lowest common ancestor. However, that method uses time $O(n)$ per determination and thus would not comply with our worst–case bound of $O(m \log n)$ per phase.

```
      CUR2->marker2 = lock;
    }
  }
  sub1.push(CUR1); sub2.push(CUR2);
```

While we are tracing the paths towards the lowest common ancestor, we keep track of the subblossoms and edges traversed on either path. The lists *sub1* and *sub2* (type *list<blossom<NT>*>*) contain the pointers of all traversed surface blossoms from *RESP* and *OPST* to *CUR1* and *CUR2* in reversed order, respectively. *P1* and *P2* (type *list<node>*) consist of all vertex pairs representing the (directed) alive path from *RESP* and *OPST* to *CUR1* and *CUR2* in reversed order, respectively.

Assume the while loop above is left, since *marker1* of *CUR2* equals *lock*. *CUR2* then denotes the lowest common ancestor blossom *LCA*. We correct *sub1* and *P1* such that the head of *sub1* equals *LCA* and the first vertex pair of *P1* corresponds to the first (directed) edge on the reversed tree path from *RESP* to *LCA*. The case where *CUR1* equals the lowest common ancestor blossom *LCA* is treated analogously.

⟨*determine LCA and shrink path of RESP and OPST*⟩+≡

```
  if (CUR2->marker1 == lock) { // CUR2 is LCA
    while (sub1.head() != CUR2) {
      sub1.pop(); sub1.pop();
      P1.pop(); P1.pop();
      P1.pop(); P1.pop();
    }
  }
  else if (CUR1->marker2 == lock) { // CUR1 is LCA
    while (sub2.head() != CUR1) {
      sub2.pop(); sub2.pop();
      P2.pop(); P2.pop();
      P2.pop(); P2.pop();
    }
  }
  // sub1.head() == sub2.head() == LCA
  LCA = sub1.pop();
  sub2.reverse(); sub1.conc(sub2);
  P2.reverse(); P1.conc(P2);
```

Finally, the concatenation of *sub1* and *sub2* (the first element of *sub1* is popped and *sub2* is reversed beforehand) yields the desired list *sub1* corresponding to the list *subblossom_p* of the new blossom object as specified previously. Analogously, the concatenation of *P1* with the reversed path *P2* corresponds to the *shrink_path* of the new blossom.

Expand Step: The responsible blossom *RESP* which is going to be expanded can easily be obtained from *delta4*.

⟨*expand step using best blossom of delta4*⟩≡

```
  RESP = delta4.inf(delta4.find_min());
  delta4.del_item(RESP->item_in_pq);
```

Next, we need to recover the data for each (immediate) subblossom of *RESP*. Therefore, we define a new member function *expand* which restores the priority queue for each subblossom of *RESP* and unfreezes the potential if necessary. *RESP* is deleted from *T* and the pointers of all subblossoms are added to *T*.

⟨*expand step using best blossom of delta4*⟩+≡

```
RESP->expand(Delta);
forall(CUR, RESP->subblossom_p)
  CUR->item_in_T = T.append(CUR);
T.del(RESP->item_in_T);
```

The details of the member function *expand* are discussed next. Later on, we will also use that member function to expand an even or unlabeled non–tree blossom. We iterate over all (immediate) subblossoms of the blossom object stored in the *subblossom_p* list. For each subblossom *CUR* we split the current priority queue of the blossom object at *CUR*'s *split_item* into two. The first of which gets assigned to *CUR* and the remaining becomes the new current priority queue of the blossom object (which will be split in the next iteration). At the end, all subblossom priority queues are restored and the priority queue of the blossom object is empty.

⟨*class blossom: member functions*⟩+≡

```
void expand(NT Delta) {
  blossom<NT> *CUR;
  forall(CUR, subblossom_p) {
    split_at_item(CUR->split_item, *CUR, *this);
    CUR->offset = offset;
    CUR->label = label;
    if (!CUR->trivial() && label == odd)
      CUR->pot += 2*offset + 2*Delta;
    else if (!CUR->trivial()) {
      assert(!CUR->item_in_T);
      assert(CUR->label == even || CUR->label == unlabeled);
      CUR->pot += 2*offset;
    }
  }
}
```

Moreover, the *offset* of each subblossom *CUR* is set to the *offset* value of the blossom object. Recall that at the time of shrinking, we arranged that each subblossom is labeled even. However, the actual potential of each vertex, and the reduced cost associated with it, is computed correctly with respect to the status of the blossom object containing that vertex. Therefore, each subblossom is labeled according to the blossom object (calling *status_change* would be wrong). Finally, the potential of each non–trivial subblossom *CUR* gets unfrozen (by the formula given in Section 2.1.3).

We can now determine the base blossom *BASE* and the discovery blossom *DISC* of *RESP* as follows.

⟨*expand step using best blossom of delta4*⟩+≡

```
blossom<NT> *BASE = _BLOSSOM_OF(RESP->base);
blossom<NT> *DISC = _BLOSSOM_OF(RESP->disc);
int dist = RESP->restore_matching(BASE, DISC);
⟨extend alternating tree⟩
delete RESP;
```

The matching needs to be restored for the subblossoms. We do so by calling the member function *restore_matching* which will be the subject of the next paragraph. In the code chunk to extend the alternating tree, we will set up some additional data for the subblossoms along the even length (alive) path from *BASE* to *DISC*, and, moreover, remove the remaining subblossoms from *T*. Finally, we can destroy the blossom object pointed to by *RESP*.

Restoring the Matching: The member function *restore_matching* restores the matching data for all subblossoms of the blossom object.

⟨*class blossom: member functions*⟩+≡

```
int restore_matching(blossom<NT> *BASE, blossom<NT> *DISC) {
  ⟨cyclically rotate subblossom_p and shrink_path list⟩
  ⟨alternately match/unmatch subblossoms along subblossom_p⟩
  return dist;
}
```

The idea is simple. We start at the base blossom *BASE* and alternately unmatch and match the edges along the odd length (alive) cycle (represented by *shrink_path*). First of all, we need to cyclically rotate the lists *subblossom_p* and *shrink_path* until the base blossom *BASE* occurs at the end of *subblossom_p*:

⟨*cyclically rotate subblossom_p and shrink_path list*⟩≡

```
while (subblossom_p.tail() != BASE) {
  subblossom_p.append(subblossom_p.pop());
  shrink_path.append(shrink_path.pop());
  shrink_path.append(shrink_path.pop());
}
```

Note that the *i*–th vertex pair of *shrink_path* corresponds to the incoming edge of the *i*–th subblossom on *subblossom_p*. The *mate* and *base* entries of the *BASE* blossom are set to *mate* and *base* of the blossom object, respectively.

⟨*alternately match/unmatch subblossoms along subblossom_p*⟩≡

```
BASE->mate = mate;
BASE->base = base;
```

Then, the subblossoms along the *subblossom_p* list are matched pairwise. In the process, we keep track of the position *dist* of *DISC* in *subblossom_p*; we start counting with 1.

⟨*alternately match/unmatch subblossoms along subblossom_p*⟩+≡

```
node b, m;
int dist, pos = 1;
list_item p_it   = shrink_path.first();
list_item sub_it = subblossom_p.first();
blossom<NT> *CUR = subblossom_p.inf(sub_it), *ADJ;
while (CUR != BASE) {
  if (CUR == DISC) dist = pos;
  sub_it = subblossom_p.succ(sub_it); pos++;
  ADJ    = subblossom_p.inf(sub_it);
  if (ADJ == DISC) dist = pos;
  p_it = shrink_path.succ(p_it);
  p_it = shrink_path.succ(p_it); b = shrink_path.inf(p_it);
  p_it = shrink_path.succ(p_it); m = shrink_path.inf(p_it);
  CUR->base = b; CUR->mate = m;
  ADJ->base = m; ADJ->mate = b;
  sub_it = subblossom_p.succ(sub_it); pos++;
  CUR    = subblossom_p.inf(sub_it);
  p_it   = shrink_path.succ(p_it);
}
if (CUR == DISC) dist = pos;
```

Extending the Alternating Tree: We need to set up some additional data, such as the *pred* and *disc* pointers etc., for the subblossoms of *RESP* lying on the even length (alive) path from *BASE* to *DISC*. Furthermore, all remaining subblossoms must leave T. The way we achieve the desired result is by another traversal of the blossom cycle. We start at the base blossom *BASE* and follow the even length path to *DISC*, setting up the necessary data for each tree blossom on this path. After this, all remaining subblossoms on the blossom cycle become unlabeled and leave T.

Remember that *dist* stores the position of *DISC* in *subblossom_p* and that *BASE* is the last element in this list. Moreover, we know that the number of elements in *subblossom_p* is odd. When *dist* is odd, the reversal of *subblossom_p* contains all subblossoms of the even length path from *BASE* to *DISC* followed by all subblossoms that leave T. Otherwise, when *dist* is even, we move *BASE* to the head of *subblossom_p*. Again, *subblossom_p* then consists of the subblossoms of the even length path from *BASE* to *DISC* followed by the subblossoms leaving T.

⟨*extend alternating tree*⟩≡

```
if (dist % 2) {
  RESP->subblossom_p.reverse();
  RESP->shrink_path.reverse();
}
else RESP->subblossom_p.push(RESP->subblossom_p.Pop());
```

We establish the following invariant for the vertex pairs along *shrink_path*. The i–th vertex pair of *shrink_path* corresponds to the outgoing edge of the i-th subblossom in *subblossom_p*.

Next, we turn to the set up of the data for the subblossoms staying in T. First, the discovery and predecessor vertices of *DISC* are set accordingly.

⟨*extend alternating tree*⟩+≡
```
DISC->disc = RESP->disc;
DISC->pred = RESP->pred;
```

Next, the first two elements *CUR* and *ADJ* are popped from the *subblossom_p* list; *CUR* corresponds to an odd blossom and *ADJ* to an even blossom. We set the *pred* and *disc* entries for *CUR*; and in case *CUR* is non–trivial, insert an item in *delta4*. *ADJ* is made even. This process is repeated until the current blossom *CUR* equals *DISC*.

⟨*extend alternating tree*⟩+≡
```
CUR = RESP->subblossom_p.pop();
while (CUR != DISC) {
  ADJ = RESP->subblossom_p.pop();
  cur = RESP->shrink_path.pop();
  adj = RESP->shrink_path.pop();
  CUR->pred = adj; CUR->disc = cur;
  if (!CUR->trivial())
    CUR->item_in_pq =
      delta4.insert(compute_potential(CUR, Delta)/2 + Delta, CUR);
  ADJ->status_change(even, Delta, T, Q);
  RESP->shrink_path.pop();
  RESP->shrink_path.pop();
  CUR = RESP->subblossom_p.pop();
}
// send item for DISC also
if (!CUR->trivial())
  CUR->item_in_pq =
    delta4.insert(compute_potential(CUR, Delta)/2 + Delta, CUR);
```

Finally, each remaining blossom *CUR* in *subblossom_p* gets unlabeled (and is thereby removed from T). When the priority queue of *CUR* is not empty, its best edge is sent to *delta2b* and *CUR* is inserted into O. Moreover, the *pred* and *disc* entries of *CUR* need to be set to *nil*.

⟨*extend alternating tree*⟩+≡
```
while (!RESP->subblossom_p.empty()) {
  CUR = RESP->subblossom_p.pop();
  CUR->status_change(unlabeled, Delta, T, Q);
  if (!CUR->empty()) {
    CUR->item_in_pq = delta2b.insert(CUR->min_prio() + CUR->offset, CUR);
    CUR->item_in_O = O.append(CUR);
  }
  CUR->pred = CUR->disc = nil;
}
```

This concludes the discussion of all details concerned with an expand step.

Extracting Matching and Checker Information: The algorithm terminates with a surface matching. We have to extract the original matching M (type *list<edge>*) by expanding all non–trivial blossoms (which are either labeled even or unlabeled).

⟨*extract matching and checker information*⟩≡

```
int k = 0;
forall_nodes(v, G)
  unpack_blossom(_BLOSSOM_OF(v), item_of, pot, b, BT, k, -1, Delta);
if (k != 0) BT.resize(k);

forall_edges(e, G)
  if (_BLOSSOM_OF(source(e))->mate == target(e)) M.push(e);
```

We use a function *unpack_blossom* which recursively expands all subblossoms to a given blossom; simultaneously, the information needed by the checker will be constructed. A surface blossom is expanded completely, the first time when one of its vertices is considered. When all blossoms are expanded, we have to reset the index range of BT to $[0, \dots, k-1]$, where k will refer to the number of non–trivial surface blossoms. Finally, each matching edge is added to M.

We turn to the description of the function *unpack_blossom*:

⟨*SST.t: helpers*⟩+≡

```
template<class NT>
void unpack_blossom(blossom<NT> *RESP, const node_array<c_pq_item> &item_of,
                    node_array<NT> &pot, node_array<int> &b,
                    array<two_tuple<NT, int> > &BT,
                    int &k, int parent, NT Delta) {

  if (RESP->trivial()) {
    ⟨set up checker data for trivial blossom⟩
  }
  else {
    ⟨set up checker data for non–trivial blossom⟩

    RESP->expand(Delta);
    blossom<NT> *BASE = _BLOSSOM_OF(RESP->base);
    blossom<NT> *DISC = nil;
    RESP->restore_matching(BASE, DISC);

    blossom<NT>* CUR;
    forall(CUR, RESP->subblossom_p)
      unpack_blossom(CUR, item_of, pot, b, BT, k, idx, Delta);

    delete RESP;
  }
}
```

The function creates the data *pot*, *b* and BT needed for the checker. We discussed the semantics of these arrays in Section 3.1 and will not repeat the discussion here. k denotes the index that is used to store the next non–trivial blossom data in BT. We use *parent* to pass the parent index of a non–trivial surface blossom to its (immediate) subblossoms.

First, assume *RESP* is a trivial blossom containing only the vertex *cur*. When *RESP* has already been expanded, i.e. $b[cur]$!= -1, we immediately leave *unpack_blossom*.

Otherwise, we simply set up its checker data:

⟨*set up checker data for trivial blossom*⟩≡

```
node cur = RESP->node_of(RESP->first_item());
if (b[cur] != -1) return;
pot[cur] = compute_potential(RESP, Delta, item_of[cur]);
b[cur] = parent;
```

The actual potential of the vertex *cur* is computed and entered in *pot*[*cur*]. *b*[*cur*] is set to the parent index *parent* (of the smallest non–trivial blossom containing *cur*; or -1).

When *RESP* is non–trivial, its checker data is set up as follows.

⟨*set up checker data for non–trivial blossom*⟩≡

```
if (k > BT.high()) BT.resize(2*k+1);
BT[k].first()  = compute_potential(RESP, Delta);
BT[k].second() = parent;
int idx = k++;
```

We double the size of BT whenever k exceeds the highest index of BT. The actual potential of *RESP* is computed and stored in the first component of $BT[k]$. *RESP*'s parent index is stored in the second component. We keep the current value of k in *idx* (which will be used as the parent index for the recursive calls) and increment k.

Subsequently, the subblossoms are expanded and the matching is restored for the (immediate) subblossoms of *RESP*. The functions needed to achieve this were discussed for the expand step (see above). Each immediate subblossom gets expanded recursively, by calling *unpack_blossom* for it. The parent index for the recursive calls is set to the index value *idx* of *RESP* in BT.

3.4 Multiple Search Tree Approach

The efficiency of a priority queue based implementation of Edmonds' blossom–shrinking approach is substantially improved when several trees are grown simultaneously. We next sketch the basic ideas underlying our multiple search tree approach. The implementation details will be presented in the subsequent sections.

An alternating tree T_i is rooted at each free vertex r_i. Each tree T_i is extended as in the single search tree approach. That is, we perform alternate, grow, shrink and expand steps as before. However, an augment step is performed differently: when a tight edge uv, $u^+ \in T_i$ and $v^+ \in T_j$ with $T_i \neq T_j$ exists, the current matching is augmented along the two tree paths (from u and v to their roots), and u and v get matched. A dual adjustment by δ changes the potentials of all vertices and surface blossoms as explained in Section 1.6.3. As before, the value of δ is determined by the lower bounds $\delta_1, \ldots, \delta_4$; δ_1 is only taken into account in the non–perfect matching case. But note that the definition of δ_3 needs to be refined now. The reduced cost of an edge uv, with $u^+ \in T_i$, $v^+ \in T_j$ and $T_i \neq T_j$, decreases by 2δ for a dual adjustment. Therefore, the reduced costs of those edges have to be taken into consideration as well. More precisely, we redefine δ_3 as follows:

$$\delta_3 \quad = \quad \min_{uv \in E} \quad \{\pi_{uv}/2 \; : \; u^+ \in T_i, \; v^+ \in T_j\},$$

where T_i and T_j refer to any of the alternating trees (different or equal).

In the multiple search tree approach, each vertex u keeps its *best connection* to an alternating tree T_i. An edge uv_i incident to u is called a *best connection* from u to T_i, when

(1) v_i is an even tree vertex in T_i, and

(2) the (stored) reduced cost π_{uv_i} is minimal along all other (stored) reduced costs π_{uv_j}, with $v_j^+ \in T_i$, i.e. $\pi_{uv_i} = \min_{uv_j \in E}\{\pi_{uv_j} : v_j^+ \in T_i\}$.

When several best connections from u to a fixed tree T_i exist, *the* best connection from u to T_i will refer to any of those.

For each vertex u we have a priority queue P_u which stores the best connections from u to *all* existing alternating trees. However, even tree vertices form an exception: when $u^+ \in T_i$ is an even tree vertex contained in an alternating tree T_i, we do not keep the best connection from u to its own tree T_i in P_u. When uv_i is the best connection from u to T_i, the corresponding item in P_u equals $\langle \pi_{uv_i}, v_i \rangle$, where π_{uv_i} denotes the (stored) reduced cost of that edge.
As before, each surface blossom $\mathcal{B}$ (trivial or non–trivial) is associated with a concatenable priority queue $P_{\mathcal{B}}$. Each vertex $u \in \mathcal{B}$ has a representative item in $P_{\mathcal{B}}$. The representative item of a vertex u in $P_{\mathcal{B}}$ corresponds to the minimum of all best connections of u (with regard to the reduced costs). Thus, the minimum item in $P_{\mathcal{B}}$ represents the best connection of $\mathcal{B}$.
In Section 2.1 we presented a strategy to handle the varying priorities for each of these priority queues.

An alternating tree T_i collects all edges uv that are candidates for a shrink step, i.e. $u^+, v^+ \in T_i$, in a priority queue P_{T_i}. The priority stored with each such edge uv corresponds to the (stored) reduced cost of that edge. In the non–perfect matching case, T_i knows its even vertex $u_i^+ \in T_i$ whose (stored) potential is minimum along all even tree vertices in T_i.

The way we will use the data associated with each surface blossom or alternating tree is as follows. Again, the lower bounds $\delta_1, \ldots, \delta_4$ that determine the value of δ are realized by means of the priority queues *delta1* to *delta4*.

In the non–perfect matching case, each alternating tree T_i has a corresponding item $\langle y_{u_i}, u_i \rangle$ in *delta1*. $u_i^+ \in T_i$ denotes the even vertex stored with T_i as introduced before; and y_{u_i} equals the (stored) potential of u_i.
Each non–tree blossom $B^{\{\varnothing|+\}}$ sends its best connection to *delta2*. An even labeled non–tree blossom will only occur in the non–perfect matching case. The (actual) priority of each item in *delta2* equals the (actual) reduced cost of the represented edge.
δ_3 is realized by two priority queues *delta3a* and *delta3b*. Generally speaking, *delta3a* keeps best connections that can be used for an augment step, and *delta3b* collects candidate edges for a shrink step. More precisely, for each best connection of an even

tree blossom $\mathcal{B}^+ \in T_i$, we have an appropriate item in *delta3a*.[11] An alternating tree T_i sends its best candidate edge uv from P_{T_i} to *delta3b*. In both priority queues, the actual priorities will correspond to one half of the actual reduced cost of the corresponding edges.
Finally, in *delta4* we collect all odd tree blossoms. The actual priority of each item equals one half of the actual potential of the corresponding blossom.

The ideas outlined should suffice for the moment. All remaining details will become clear in the rest of this section, where we discuss our implementation of a multiple search tree approach. Many particulars have been presented for the single search tree approach in the preceding section. We will therefore focus on the ensuing modifications and extensions.

3.4.1 Data Structures

We come to the data structures of our implementation.

⟨*MST.t: data structures*⟩≡

```
  template<class NT> class blossom;
  template<class NT> class vertex;
  template<class NT> class tree;
  ⟨class blossom⟩
  ⟨class vertex⟩
  ⟨class tree⟩
```

As before, blossoms are represented by an object of class *blossom*. All data members and most of the member functions that have been introduced for that class in the preceding section will be reused.

Definition of the Additional Class *vertex*:

We define a class *vertex* which keeps all data associated with a vertex. Its overall structure is given below.

⟨*class vertex*⟩≡

```
  template<class NT>
  class vertex : public virtual p_queue<NT, node> {
  public:
    NT   pot;
    node my_node;
    h_array<node, pq_item> ITEM_OF;
    ⟨class vertex: member functions⟩
    LEDA_MEMORY(vertex<NT>);
  };
```

[11] Here, we need to have each even tree vertex keep track of its best connections to other, i.e. different, alternating trees.

Class *vertex* inherits all properties of a priority queue (type *p_queue*<*NT*, *node*>). The priorities are of type *NT* and the information part refers to a vertex. Each best connection uv_i of a vertex u to an alternating tree T_i is represented by an item $\langle \pi_{uv_i}, v_i \rangle$ in the priority queue. As before, *pot* is set to the (stored) potential u and *my_node* denotes the vertex u itself. We will need to identify the item corresponding to u's best connection to a given tree T_i. Therefore, we use a hashing array *ITEM_OF* (type *h_array*<*node*, *pq_item*>) which maps the root vertex r_i of a tree T_i to the appropriate item (type *pq_item*). *h_array* is a dynamic data type provided by LEDA. It is implemented by hashing with chaining. All access operations take expected time $O(1)$. The operations of this data type that are used will be explained briefly at the time they are first needed.

Constructor: The constructor of class *vertex* is trivial. It simply creates a new vertex object for a vertex u having potential d.

⟨*class vertex: member functions*⟩≡

```
vertex(NT d, node u) : p_queue<NT, node>() { pot = d; my_node = u; }
```

The object is initialized with the empty priority queue, and *ITEM_OF* is undefined for all vertices.

Member Functions: We come to some standard access functions. *min_prio* returns the priority of the minimum item; and *min_inf* the information part.

⟨*class vertex: member functions*⟩+≡

```
NT   min_prio() const
{ return (find_min() ? prio(find_min()) : INFINITY(NT)); }
node min_inf() const
{ return (find_min() ? inf(find_min()) : nil); }
```

As before, *INFINITY*(*NT*) or *nil* is returned, respectively, when the priority queue is empty.

We also need to redefine the member function *best_adj* of class blossom:

⟨*class blossom: member functions*⟩+≡

```
const node best_adj(c_pq_item it) const { return inf(it)->min_inf(); }
```

This function returns the vertex stored with the minimum item in the priority queue of a vertex object (*inf*(*it*)).

The following member function tries to improve the best connection of a vertex object to a tree, say T, rooted at r.

⟨*class vertex: member functions*⟩+≡

```
bool decrease_p(node u, NT x, node r) {
  NT old_min = min_prio();
  if (!ITEM_OF.defined(r))
```

```
      ITEM_OF[r] = insert(x, u);
    else {
      pq_item it = ITEM_OF[r];
      if (prio(it) > x) {
        p_queue<NT, node>::decrease_p(it, x);
        p_queue<NT, node>::change_inf(it, u);
      }
    }
    return old_min != min_prio();
  }
```

u denotes the even tree vertex in T, and x will correspond to the (stored) reduced cost of the edge from *my_node* to u. First, we check whether the vertex object stores an item representing a best connection to T. We do so by means of a *defined* operation provided by the data type *h_array*. *defined*(r) returns *true*, iff an item has been set for r. In the case where r is not defined for *ITEM_OF*, we insert a new item $\langle x, u \rangle$ representing the best connection to T. *ITEM_OF*$[r]$ is set to the corresponding *pq_item* (and henceforth defined for r). Otherwise, we can retrieve the item *it* of the current best connection to T by an access operation *ITEM_OF*$[r]$. When x is smaller than the priority currently stored with *it*, the priority of *it* is decreased to x and the information is changed to u. The function returns *true*, iff the minimum priority of the vertex object has changed.

We will need a member function to delete the best connection of a vertex object to a tree rooted at r.

⟨*class vertex: member functions*⟩+≡

```
  bool del(node r) {
    if (!ITEM_OF.defined(r)) return false;
    NT old_min = min_prio();
    del_item(ITEM_OF[r]);
    ITEM_OF.undefine(r);
    return old_min != min_prio();
  }
```

Given the root vertex r, we can look up its item using the access operation *ITEM_OF*$[r]$; when *ITEM_OF* is not defined for r nothing has to be done. This item is deleted from the priority queue and *ITEM_OF* becomes undefined for r by calling *undefine*(r). If the minimum priority has changed due to the deletion, the function returns *true*; otherwise *false*.

Definition of the Additional Class *tree*:

We define a new class *tree* to maintain the necessary data for the alternating trees.

⟨*class tree*⟩≡

```
  ⟨class tree: friend functions — definition⟩
  template<class NT> class tree {
    ⟨class tree: friend functions — declaration⟩
  public:
    node root;
    node d1_node;
    list<blossom<NT>*> my_blossoms;
    p_queue<NT, edge>  d3b_edges;
    pq_item            item_in_d3b;
    ⟨class tree: member functions⟩
    LEDA_MEMORY(tree<NT>);
  };
```

An object T of class *tree* (type *tree*<*NT*>) stores its root vertex in *root*. The pointers of all surface blossoms contained in T are collected in a list *my_blossoms*. *d1_node* denotes an even vertex of T having minimum potential (along all even vertices of T); this entry will be used in the non–perfect matching case only. Additionally, each alternating tree T has its own priority queue *d3b_edges*. An item $\langle p, e\rangle$ in *d3b_edges* represents an edge e having (stored) reduced cost p; moreover, e is a candidate for a shrink step, i.e. $e = uv$ with $u^+, v^+ \in T$. The minimum item of *d3b_edges* is sent as a representative to a global priority queue *delta3b*. *item_in_d3b* enables the identification of this item in *delta3b*.

We briefly explain why we decided to keep a separate priority queue for each alternating tree. One could, alternatively, simply insert all these edges in the global priority queue *delta3b*. But when an alternating tree T is destroyed after an augment step, we would need a mechanism to identify all candidate edges of T in *delta3b*. Each such edge would have to be deleted separately from *delta3b* consuming time $O(\log m)$; or $O(m \log m)$ in total.
In our strategy, however, we simply delete the representative of T (accessible by *item_in_d3b*) and make the priority queue *d3b_edges* empty. This will take total time $O(\log m + m)$.

Each blossom stores a pointer to its alternating tree. That is, we add the following data member to the blossom class:

⟨*class blossom: data members*⟩+≡

```
  tree<NT> *my_tree;
```

When a new blossom object is constructed, *my_tree* is set to *nil*. Furthermore, an access operation *tree_root* is defined to return the root vertex of the alternating tree containing the blossom.

⟨*class blossom: member functions*⟩+≡

```
  const node tree_root() const { return (my_tree ? my_tree->root : nil); }
```

Constructor: The construction of a tree object is trivial. *d1_node* and *item_in_d3b* are set to *nil*. A root vertex r for the tree object to be created can be given as an optional argument.

⟨*class tree: member functions*⟩≡

```
  tree(node r = nil) { root = r; d1_node = nil; item_in_d3b = nil; }
```

Initially, *my_blossoms* is empty and *d3b_edges* contains no items.

Member Functions: At this, we present only some basic member functions. The remaining ones will be introduced when required.

A blossom object (pointed to by) B is added to a tree as follows:

⟨*class tree: member functions*⟩+≡

```
  void add(blossom<NT> *B)
  { B->item_in_T = my_blossoms.append(B); B->my_tree = this; }
```

B is appended to the list *my_blossom* of the alternating tree. The item (type *list_item*) of B in *my_blossoms* is stored in the data member *item_in_T* of B; and *my_tree* of B is set to the current tree object.

Conversely, the removal of a blossom B from an alternating tree is realized by *remove*:

⟨*class tree: member functions*⟩+≡

```
  void remove(blossom<NT> *B)
  { my_blossoms.del(B->item_in_T); B->item_in_T = nil; B->my_tree = nil; }
```

The operations needed to retrieve the priority or information part of the minimum item in *d3b_edges* are given below.

⟨*class tree: member functions*⟩+≡

```
  const NT min_prio() const {
    return (d3b_edges.find_min() ? \
            d3b_edges.prio(d3b_edges.find_min()) : INFINITY(NT)); }
  const edge min_inf() const
  { return (d3b_edges.find_min() ? \
            d3b_edges.inf(d3b_edges.find_min()) : nil); }
```

We define an operation *ins*: it inserts an item $\langle x, e\rangle$ for an edge e having (stored) reduced cost x into the priority queue *d3b_edges*.

⟨*class tree: member functions*⟩+≡

```
  bool ins(NT x, edge e) {
    pq_item old_min = d3b_edges.find_min();
    d3b_edges.insert(x, e);
    return (old_min != d3b_edges.find_min());
  }
```

The function returns *true*, iff the minimum item in *d3b_edges* has changed.

Friend Functions: We declare a function *new_tree* that allows us to create a new tree object more comfortably.

⟨*class tree: friend functions — declaration*⟩≡

```
  friend tree<NT>* new_tree<>(node r, blossom<NT>* &B);
```

It constructs a new tree object that represents an alternating tree rooted at r. B is the only blossom contained in this tree. The function returns a pointer to the new tree object.

⟨*class tree: friend functions — definition*⟩≡

```
template<class NT> tree<NT>* new_tree(node r, blossom<NT>* &B) {
  tree<NT>* T = new tree<NT>(r);
  B->item_in_T = T->my_blossoms.append(B);
  return T;
}
```

3.4.2 Algorithm

The data structures introduced above will suffice for our multiple search tree algorithm. We now proceed to present the implementation details of the algorithm. Altogether, five priority queues will be needed:

⟨*local variables*⟩+≡

```
node_pq<NT> delta1(G);
```

delta1 is a specialized priority queue of type *node_pq<NT>*. A *node_pq* is realized more efficiently than a priority queue of type *p_queue<NT, node>*. However, it can only be used with the restriction that each vertex occurs in at most one *node_pq*. The data type suits our purposes perfectly. For each tree we set the priority of its *d1_node* in *delta1* to the (stored) potential of the vertex. Note that *delta1* will only be used, however, in the non–perfect case.

⟨*local variables*⟩+≡

```
p_queue<NT, blossom<NT>*> delta2;
```

delta2 contains an item $\langle p, pt\rangle$ for each non–tree blossom. The actual priority of p equals the actual reduced cost of the best connection of the blossom pointed to by pt. In the perfect matching case, each such non–tree blossom will be unlabeled; however, in the non–perfect matching case also even labeled non–tree blossoms will occur.

⟨*local variables*⟩+≡

```
p_queue<NT, blossom<NT>*> delta3a;
p_queue<NT, tree<NT>*>    delta3b;
```

In *delta3a*, each item $\langle p, pt\rangle$ refers to an even tree blossom (pointed to by pt). The actual priority of p equals one half of the actual reduced cost of the best connection of this blossom.
Each alternating tree T sends an item $\langle p, pt\rangle$ to *delta3b*. pt is a pointer to T. The actual priority of p corresponds to one half of the actual reduced cost of the best candidate edge for a shrink step in T (stored in *d3b_edges*).

⟨*local variables*⟩+≡

```
p_queue<NT, blossom<NT>*> delta4;
```

An item $\langle p, pt\rangle$ in *delta4* represents an odd tree blossom (pointed to by pt) having actual potential equal to one half of the actual priority of p.

Many local variables, as introduced for the single search tree algorithm, are needed here as well. For instance, the *node_array item_of*, the singly linked list of nodes Q (type *node_slist*), the global counter *Delta*, the list of matching edges M (type *list*<*edge*>), etc. We will not discuss their meaning again, but refer to the description in the preceding section.

The overall structure of the algorithm changes slightly.

⟨*MST.t: algorithm*⟩≡

```
template<class NT>
list<edge> MWM_MST(const ugraph &G, const edge_array<NT> &w,
                   node_array<NT> &pot, array<two_tuple<NT, int> > &BT,
                   node_array<int> &b, int heur = 1, bool perfect = false) {
  ⟨local variables⟩
  int free = G.number_of_nodes();
  ⟨initialization⟩
  while (free) {
    ⟨scan all edges of vertices in Q⟩
    ⟨determine lower bounds cand1, ... , cand4⟩
    if (cand3a == Delta) {
      ⟨augment step using best connection of blossom in delta3a⟩
      free -= 2;
    }
    else if (cand1 == Delta) {
      ⟨alternate step using best node of delta1⟩
      free -= 1;
    }
    else if (cand2 == Delta) {
      ⟨grow or augment step using best connection of blossom in delta2⟩
    }
    else if (cand3b == Delta) {
      ⟨shrink step using best edge in delta3b⟩
    }
    else if (cand4 == Delta) {
      ⟨expand step using best blossom of delta4⟩
    }
    else {
      ⟨dual adjustment⟩
    }
  }
  ⟨extract matching and checker information⟩
  return M;
}
```

The counter *free* has to be interpreted as follows. In the perfect matching case, *free* simply refers to the number of free vertices. But in the non–perfect matching case, *free* denotes the number of free vertices having (actual) potential larger than zero, i.e. the number of vertices that violate (CS)(2) (see Section 1.6.1). An alternate step will decrease *free* by 1, whereas an augment step decreases *free* by 2.

We determine the minimum value *cand1*, *cand2*, *cand3a*, *cand3b* and *cand4* of each priority queue *delta1*, *delta2*, *delta3a*, *delta3b* and *delta4*, respectively:

⟨*determine lower bounds cand1, ... , cand4*⟩≡

```
NT cand1  = (delta1.empty()  ? INFTY : delta1.prio(delta1.find_min()));
NT cand2  = (delta2.empty()  ? INFTY : delta2.prio(delta2.find_min()));
NT cand3a = (delta3a.empty() ? INFTY : delta3a.prio(delta3a.find_min()));
NT cand3b = (delta3b.empty() ? INFTY : delta3b.prio(delta3b.find_min()));
NT cand4  = (delta4.empty()  ? INFTY : delta4.prio(delta4.find_min()));
```

When any of these values equals *Delta* (and hence the actual priority equals zero), the appropriate step is initiated. Regarding the specific order of these steps, the same arguments apply as were given for the single search tree approach. The realization of each step will be discussed below.

We perform a dual adjustment as follows. The code is similar to the one discussed for the single search tree algorithm.

⟨*dual adjustment*⟩≡

```
NT delta = leda_min(cand1,
                    leda_min(cand2,
                             leda_min(cand3a,
                                      leda_min(cand3b, cand4))));
if ((delta == INFTY) && perfect) return M;   // return empty matching
Delta = delta;   // corresponds to Delta += (delta - Delta)
```

When the value of *free* drops to zero, the algorithm terminates. We extract the matching and checker information in exactly the same way as has been described for the single search tree approach; therefore, the code realizing this will not be repeated here.

Initialization: The initialization differs only slightly. As before, depending on the value of *heur* we construct either an empty matching, a greedy matching or a jump start matching. Remember that the *node_arrays* *mate* and *pot* represent the constructed matching and the vertex potentials. What differs is the way we set up the data for each blossom:

⟨*initialization*⟩+≡

```
forall_nodes(u, G) {
  item_of[u] = new_blossom<NT>(pot[u], u, CUR);
  if (mate[u]) {
    CUR->mate = mate[u];
    CUR->label = unlabeled;
  }
  else {
    CUR->my_tree = new_tree<NT>(u, CUR);
    Q.append(u);
  }
}
```

For each vertex u of G we construct a trivial blossom *CUR* consisting of u only. The potential of u is set to $pot[u]$. When u is matched, its mate is stored in the data member *mate* of *CUR*, and *CUR* gets unlabeled. Otherwise, we construct a new alternating tree which is rooted at u. *CUR* is the only blossom of this tree object; we let *my_tree* of *CUR* point to the object. u is added to Q such that the priority queue data for each adjacent vertex of u will be set up correctly when all edges incident to any vertex in Q are scanned for the first time.

Performing a Status Change: We next revise the member function of class blossom that performs a status change. The overall structure remains the same.

⟨*class blossom: member functions*⟩+≡

```
void status_change(LABEL l, NT Delta, node_slist &Q) {
  if (l == unlabeled) {
    assert((label != l) && item_in_T);
    offset += (label == odd ? Delta : -Delta);
    my_tree->remove(this);
  }
  else if (l == odd) {
    assert((label != l) && !item_in_T);
    offset -= Delta;
    my_tree->add(this);
  }
  else if (l == even) {
    assert((label != l) || !item_in_T);
    if (label == odd) offset += 2*Delta;
    else {  // non-tree blossom
      offset += Delta;
      my_tree->add(this);
    }
    ⟨append all vertices to Q⟩
  }
  label = l;
}
```

We adjust the *offset* value of the blossom object as outlined for the single search tree approach. The blossom object is added to or removed from the alternating tree using the member functions *add* and *remove* of class *tree*, respectively. What differs, however, is the action to be taken when a blossom becomes an even tree blossom.

⟨*append all vertices to Q*⟩≡

```
c_pq_item it;
forall_items(it, *this) {
  Q.append(node_of(it));
  delete_connection(it, my_tree->root);
  ⟨adjust vertex potential and priorities (in provident case)⟩
}
⟨adjust blossom potential and offset (in provident case)⟩
```

We add each vertex of the blossom object to the list Q and also delete the best connection for each such vertex to the current tree; the member function *delete_connection* will be discussed below. We do so in order to comply with the convention that each even tree vertex keeps its best connections to every different tree.

Another difference is that we do not use the provident strategy (see Section 2.1.3) as

in the case of the single search tree approach. That is, the potential and priorities associated with each vertex of the blossom are not adjusted so as to compute their actual value with respect to the offset *offset* = 0. Instead, we implement the non–provident strategy. We will come back to this point when the implementation of a shrink step is considered more closely.
We have experimented with both strategies for the multiple search tree approach. The non–provident strategy seems to be slightly more efficient in practice and is thus used by default.

However, we briefly state all additional details for the implementation of the provident strategy.[12]

⟨*adjust vertex potential and priorities (in provident case)*⟩≡

```
#ifdef _PROVIDENT
if (offset != 0) {
  inf(it)->pot += offset;
  if (inf(it)->empty()) continue;
  inf(it)->adjust_priorities(offset);
  increase_p(it, prio(it) + offset);
}
#endif
```

As before, we iterate over all items *it* of the blossom object. When the blossom offset differs from zero, the potential of each vertex (*inf*(*it*)) contained in the current blossom object is adjusted as described in Section 2.1.3. Moreover, we need to increase all priorities stored with each vertex object by *offset*. This is achieved by calling the member function *adjust_priorities* of class vertex. Its implementation will be discussed for the shrink step, later on. The priority of item *it* is also increased by *offset*, calling the inherited function *increase_p*.

⟨*adjust blossom potential and offset (in provident case)*⟩≡

```
#ifdef _PROVIDENT
if (!trivial()) pot -= 2*offset;
offset = 0;
#endif
```

Finally, the blossom potential is adjusted (when non–trivial) and *offset* is set to zero.

What remains to be presented is the member function *delete_connection* of class blossom:

⟨*class blossom: member functions*⟩+≡

```
bool delete_connection(c_pq_item it, node r) {
  if (!it) return false;
  NT old_min = min_prio();
  if (inf(it)->del(r))
    if (inf(it)->empty())
      del_item(it);
    else
      increase_p(it, inf(it)->min_prio());
  return old_min != min_prio();
}
```

[12]Defining the token _PROVIDENT (#define _PROVIDENT), before the file MWM.t is included, forces the algorithm to use the provident instead of the non–provident strategy.

For a given item *it* (type *c_pq_item*), the function deletes the best connection from the corresponding vertex object (pointed to by *inf*(*it*)) to the tree rooted at r. The realization is simple: we use the member function *del* of class vertex to delete the corresponding item in the priority queue of the vertex object. When *del* returns *true*, i.e. when the minimum item has been changed due to this operation, we need to update the priority of *it* in the concatenable priority queue. Two cases are distinguished: when the priority queue of the vertex object is empty, *it* is deleted (its priority is set to *infinity*); otherwise, the priority of *it* is increased to the new minimum priority stored in the priority queue of the vertex object. The function returns *true* iff the minimum priority of the blossom has changed.

Scanning New Even Vertices: We next give some details of the scanning procedure. All edges e incident to a vertex *cur* in Q are inspected in order to correctly maintain the priority queues *delta1* to *delta4* as well as the priorities associated with each vertex, tree or blossom. Most of the details are similar to those discussed for the single search tree approach.

⟨*scan all edges of vertices in Q*⟩≡
```
NT cur_pot, adj_pot, actual_p, stored_p;
while (!Q.empty()) {
  cur     = Q.pop();
  CUR     = _BLOSSOM_OF(cur);
  cur_pot = compute_potential(CUR, Delta, item_of[cur]);
  if (!perfect) {
    ⟨try to improve delta1⟩
  }
  forall_adj_edges(e, cur) {
    adj = opposite(cur, e);
    ADJ = _BLOSSOM_OF(adj);
    ⟨discard dead and tree edges⟩
    adj_pot  = compute_potential(ADJ, Delta, item_of[adj]);
    actual_p = cur_pot + adj_pot - w[e];
    if (!ADJ->item_in_T) {
      ⟨new delta2 edge encountered⟩
    }
    else if ((ADJ->label == even) && (ADJ->my_tree != CUR->my_tree)) {
      ⟨new delta3a edge encountered⟩
    }
    else if ((ADJ->label == even) && (ADJ->my_tree == CUR->my_tree)) {
      ⟨new delta3b edge encountered⟩
    }
    else if (ADJ->label == odd) {
      stored_p = actual_p - ADJ->offset;
      ADJ->improve_connection(item_of[adj], stored_p, cur, CUR->tree_root());
    }
  }
}
```

In the non–perfect matching case, we need to keep the even tree vertex *d1_node* for each alternating tree. Recall that the vertex *d1_node* is supposed to denote the vertex that has minimum potential along all even tree vertices contained in the tree object.

⟨*try to improve delta1*⟩≡

```
if (!CUR->my_tree->d1_node) {
  delta1.insert(cur, cur_pot + Delta);
  CUR->my_tree->d1_node = cur;
}
else if (cur_pot < delta1.prio(CUR->my_tree->d1_node) - Delta) {
  delta1.del(CUR->my_tree->d1_node);
  delta1.insert(cur, cur_pot + Delta);
  CUR->my_tree->d1_node = cur;
}
```

When no vertex is stored in *d1_node* of the tree object containing *CUR*, we simply set this data member to *cur* and insert an appropriate item into *delta1*. Otherwise, we look up the current stored potential of *d1_node* (in *delta1*). When the actual potential of *d1_node* is larger than the actual potential *cur_pot* of *cur*, we proceed as follows. *d1_node* is deleted from *delta1* and the new vertex *cur* is inserted with its stored potential $cur\_pot + Delta$. Moreover, the vertex stored in *d1_node* is replaced by *cur*.

When *ADJ* is a non–tree blossom, we have possibly discovered a new best connection from *adj* to the alternating tree containing *CUR*; let *T* denote the object representing this alternating tree.

⟨*new delta2 edge encountered*⟩≡

```
stored_p = actual_p - ADJ->offset + Delta;
if (ADJ->improve_connection(item_of[adj], stored_p, cur, CUR->tree_root()))
  if (ADJ->item_in_pq)
    delta2.decrease_p(ADJ->item_in_pq, actual_p + Delta);
  else
    ADJ->item_in_pq = delta2.insert(actual_p + Delta, ADJ);
```

We compute the stored reduced cost *stored_p* of that edge and try to improve the connection from *adj* to *T* by calling *improve_connection*, which will be discussed shortly. The function returns *true* if the minimum priority of *ADJ* has changed, i.e. the currently inspected edge is the new best connection of *ADJ*. If so, we either decrease the corresponding priority in *delta2* (when *ADJ* has an item in *delta2*), or insert a new item into *delta2*.

We proceed in a similar way when *ADJ* represents an even tree blossom contained in a different tree:

⟨*new delta3a edge encountered*⟩≡

```
stored_p = actual_p - ADJ->offset + 2*Delta;
if (ADJ->improve_connection(item_of[adj], stored_p, cur, CUR->tree_root())) {
  if (ADJ->item_in_pq)
    delta3a.decrease_p(ADJ->item_in_pq, actual_p/2 + Delta);
  else
    ADJ->item_in_pq = delta3a.insert(actual_p/2 + Delta, ADJ);
}
```

The member function *improve_connection* of class blossom is implemented as follows.

⟨*class blossom: member functions*⟩+≡

```
bool improve_connection(c_pq_item it, NT x, node u, node r) {
  if (!it) return false;
  NT old_min = min_prio();
  if (inf(it)->decrease_p(u, x, r)) decrease_p(it, x);
  return old_min != min_prio();
}
```

For a given item *it* (type *c_pq_item*), we try to improve the best connection from the corresponding vertex (pointed to by *inf*(*it*)) to the tree rooted at *r*. *x* denotes the (stored) reduced cost of the newly discoverd connection, and *u* refers to an even vertex contained in the tree rooted at *r*. We use the member function *decrease_p* of class *vertex*. Its implementation has been described before. When this connection is the new minimum item of the priority queue of the vertex, i.e. *decrease_p* returns *true*, the priority of item *it* is decreased to *x* as well. The function returns *true* iff the minimum priority of the blossom object has changed.

We next discuss the case where *ADJ* is an even tree blossom contained in the same tree as *CUR*.

⟨*new delta3b edge encountered*⟩≡

```
tree<NT> *T = CUR->my_tree;
if (T->ins(actual_p/2 + Delta, e))
  if (T->item_in_d3b)
    delta3b.decrease_p(T->item_in_d3b, actual_p/2 + Delta);
  else
    T->item_in_d3b = delta3b.insert(actual_p/2 + Delta, T);
```

Using the member function *ins* of class tree, we insert the new candidate edge *e* into the priority queue *d3b_edges* of *T*. When *e* is the new minimum edge of this tree (*ins* returns *true*), we update *T*'s item in *delta3b* accordingly.

Alternate Step: An alternate step will only be initiated in the non–perfect matching case. The responsible vertex *resp* which attains the minimum in *delta1* is retrieved. *RESP* denotes the surface blossom of *resp*.

⟨*alternate step using best node of delta1*⟩≡

```
resp = delta1.del_min();
RESP = _BLOSSOM_OF(resp);
RESP->base = resp;
alternate_path(RESP, item_of);
```

The edges along the tree path are alternated starting from *RESP*. *RESP* will become free, and hence we must set the *base* of *RESP* to *resp*; *resp*'s actual potential equals zero and is thus allowed to stay free. The function *alternate_path* has been given in the preceding section.

⟨*alternate step using best node of delta1*⟩+≡

```
  slist<blossom<NT>*> correct;
  RESP->my_tree->destroy_tree(correct, delta1, delta3a,
                              delta3b, delta4, Delta, Q, item_of);
  correct_pqs(correct, delta2, delta3a);
  RESP->label = even;
```

The tree object containing *RESP* is destroyed by calling the member function *destroy_tree*, which will be the subject of the next paragraph. Destroying an alternating tree object is more complicated than in the single search tree approach: we need to remove all best connections to this tree. As a consequence, the minimum item of some non–tree blossoms or even labeled tree blossoms may change, and thus their corresponding items in *delta2* and *delta3a* need to be adjusted. *destroy_tree* will return these blossoms (represented by their pointers) in a list *correct*. Calling *correct_pqs* for this list will achieve the desired result. Finally, we have to set the *label* of *RESP* to *even* (*destroy_tree* makes *RESP* unlabeled).

Destroy Tree: When a tree object T is going to be destroyed, it is not sufficient to delete the corresponding items of each even or odd tree blossom from *delta3a* or *delta4*; we also have to delete all best connections to this tree. We tried two different strategies to achieve the latter goal.
One of the strategies is as follows: we keep all vertices that store a best connection to T in a list. When T gets destroyed, we traverse this list and simply delete each such connection. The time needed to do so is $O(n \log n)$, since there can be at most n vertices.
Another possibility is to inspect each edge uv incident to any even vertex $u^+ \in T$. When v (still) stores a best connection to T, it gets deleted.[13] The time required by this method is $O(\deg(T) + n \log n)$, where $\deg(T)$ refers to the total number of edges incident to all even vertices contained in T. Obviously, $\deg(T)$ is bounded by m, the number of edges.
Although the first strategy looks better with respect to the theoretical running–time, the latter turned out to be more efficient in practice. We therefore decided to use the latter strategy.

⟨*class tree: member functions*⟩+≡

```
  void destroy_tree(slist<blossom<NT>*> &correct,
                    node_pq<NT> &delta1,
                    p_queue<NT, blossom<NT>*> &delta3a,
                    p_queue<NT, tree<NT>*> &delta3b,
                    p_queue<NT, blossom<NT>*> &delta4,
                    NT Delta, node_slist &Q, node_array<c_pq_item> &item_of) {
    blossom<NT>* CUR;
    forall(CUR, my_blossoms) {
      if (CUR->label == even)
        CUR->delete_all_connections(item_of, correct);
      ⟨delete item of CUR from delta3a or delta4⟩
```

[13]Note that a vertex v may be considered several times due to the existence of different edges $u_1v, u_2v, \ldots$ where $u_1^+, u_2^+, \ldots \in T$.

```
      if (!CUR->min_changed) {
        correct.push(CUR);
        CUR->min_changed = true;
      }
      CUR->pred = CUR->disc = nil;
      CUR->status_change(unlabeled, Delta, Q);
    }
    ⟨delete item of tree from delta1 and delta3b⟩
    delete this;
  }
```

We iterate over all blossoms *CUR* contained in the tree. When *CUR* is even, the best connection from each adjacent vertex of *CUR* to the tree is deleted. The way we achieve this is by calling the member function *delete_all_connections* of class *vertex*. We will come back to the realization of this member function shortly.

When *CUR* has sent an item to *delta3a* (in the case where *CUR* is even) or an item to *delta4* (in the case where *CUR* is odd) we delete that item.

⟨*delete item of CUR from delta3a or delta4*⟩≡

```
  if (CUR->item_in_pq) {
    if (CUR->label == even)
      delta3a.del_item(CUR->item_in_pq);
    else
      delta4.del_item(CUR->item_in_pq);
    CUR->item_in_pq = nil;
  }
```

In *correct* (type *slist<blossom<NT>*>*), we collect all non–tree blossoms or even labeled tree blossoms whose corresponding item in *delta2* or *delta3a* needs to be adjusted. Each tree blossom *CUR* will become an unlabeled non–tree blossom and thus we add *CUR* to *correct*. Since we want each such blossom to occur only once in this list, we introduce a new data member for class blossom:

⟨*class blossom: data members*⟩+≡

```
  bool min_changed;
```

Initially, *min_changed* is set to *false*. Whenever a blossom object is stored in *correct*, *min_changed* will be set to *true*. The *pred* and *disc* entries of *CUR* are set to *nil* and the status of *CUR* is changed to unlabeled.

Finally, the priority stored for *d1_node* in *delta1* has to be removed. Moreover, when the alternating tree has an item in *delta3b*, we delete this item as well.

⟨*delete item of tree from delta1 and delta3b*⟩≡

```
  if (d1_node)
    delta1.del(d1_node);
  if (item_in_d3b)
    delta3b.del_item(item_in_d3b);
```

We now discuss the member function *delete_all_connections* of class blossom.

⟨*class blossom: member functions*⟩+≡

```
void delete_all_connections(const node_array<c_pq_item> &item_of,
                            slist<blossom<NT>*> &correct) {
  edge e;
  c_pq_item it;
  node cur, adj;
  blossom<NT> *ADJ;
  forall_items(it, *this) {
    cur = node_of(it);
    forall_adj_edges(e, cur) {
      adj = opposite(cur, e);
      ADJ = blossom_of<NT>(item_of[adj]);
      bool min_changed = ADJ->delete_connection(item_of[adj], tree_root());
      if (min_changed && !ADJ->min_changed && ADJ->label != odd) {
        correct.append(ADJ);
        ADJ->min_changed = true;
      }
    }
  }
}
```

For each vertex *cur* contained in the blossom object, we inspect each incident edge *e*. *adj* denotes the vertex which is adjacent to *cur* with respect to *e*. The blossom containing *adj* is pointed to by *ADJ*. We delete the best connection from *adj* to the tree containing the current blossom by calling *delete_connection*. The implementation details for this function have already been given above. When the minimum of *ADJ* has changed and *ADJ* is either labeled even or unlabeled we add *ADJ* to *correct*. However, this will be done only when *ADJ* is not already contained in *correct*.

Correcting Global Priority Queues: We come to the corrections that are necessary for the blossoms stored in the list *correct*. Note that each blossom in *correct* is either a non–tree blossom or an even tree blossom.

⟨*MST.t: helpers*⟩+≡

```
template<class NT>
void correct_pqs(slist<blossom<NT>*> &correct,
                 p_queue<NT, blossom<NT>*> &delta2,
                 p_queue<NT, blossom<NT>*> &delta3a) {
  blossom<NT> *CUR;
  forall(CUR, correct) {
    if (CUR->item_in_pq) {
      ⟨delete item of CUR from delta2 or delta3a⟩
    }
    if (!CUR->empty()) {
      ⟨insert item for CUR in delta2 or delta3a⟩
    }
    CUR->min_changed = false;
  }
  correct.clear();
}
```

For each blossom *CUR*, we first delete its item (if any) from *delta2* or *delta3a*:

⟨*delete item of CUR from delta2 or delta3a*⟩≡

```
if (!CUR->item_in_T)
  delta2.del_item(CUR->item_in_pq);
else
  delta3a.del_item(CUR->item_in_pq);
CUR->item_in_pq = nil;
```

and then insert a new item (if necessary) into *delta2* or *delta3a*:

⟨*insert item for CUR in delta2 or delta3a*⟩≡

```
if (!CUR->item_in_T)
  CUR->item_in_pq = delta2.insert(CUR->min_prio() + CUR->offset, CUR);
else
  CUR->item_in_pq = delta3a.insert((CUR->min_prio() + CUR->offset)/2, CUR);
```

The data member *min_changed* of *CUR* is set to *false*, and, eventually, *correct* is made empty.

Augment Step: When the actual priority of the minimum item in *delta3a* equals zero, an augment step is initiated. Most of the details given previously suffice for the discussion of the implementation details of this step.

⟨*augment step using best connection of blossom in delta3a*⟩≡

```
RESP = delta3a.inf(delta3a.find_min());
delta3a.del_item(RESP->item_in_pq);
RESP->item_in_pq = nil;
```

We retrieve the even tree blossom *RESP* stored in the information part of the minimum item and then delete this item from *delta3a*.

The best connection of *RESP* corresponds to the new tight edge that we will use to augment the matching. We define a member function *best_edge* for class blossom as follows:

⟨*class blossom: member functions*⟩+≡

```
void best_edge(node &resp, node &opst) const {
  resp = node_of(find_min());
  opst = best_adj(find_min());
}
```

This member function allows us to determine the endpoints of the best connection to a given blossom more elegantly.

⟨*augment step using best connection of blossom in delta3a*⟩+≡

```
RESP->best_edge(resp, opst);
OPST = _BLOSSOM_OF(opst);
```

resp corresponds to the vertex contained in the blossom *RESP* and *opst* denotes the other endpoint contained in *OPST*. *OPST* represents an even tree blossom. Note that the trees containing *RESP* and *OPST* are distinct. The two tree paths from *RESP* and *OPST* to their roots are alternated, calling the function *alternate_path* for each blossom.

⟨*augment step using best connection of blossom in delta3a*⟩+≡

```
alternate_path(RESP, item_of);
alternate_path(OPST, item_of);
RESP->base = OPST->mate = resp;
RESP->mate = OPST->base = opst;
```

After this, we match *RESP* and *OPST* with each other by setting their *base* and *mate* entries appropriately. What remains to be done is to delete the two trees containing *RESP* and *OPST*. The function used to achieve this has been discussed above.

⟨*augment step using best connection of blossom in delta3a*⟩+≡

```
slist<blossom<NT>*> correct;
RESP->my_tree->destroy_tree(correct, delta1, delta3a,
                            delta3b, delta4, Delta, Q, item_of);
OPST->my_tree->destroy_tree(correct, delta1, delta3a,
                            delta3b, delta4, Delta, Q, item_of);
correct_pqs(correct, delta2, delta3a);
```

Finally, we update the items in *delta2* and *delta3a* for blossoms collected in *correct*.

Grow or Augment Step: The priority queue *delta2* keeps all best connections of non–tree blossoms. In the perfect matching case, each such blossom will be unlabeled and thus its best connection can be used for a grow step. However, since alternate steps might occur in the non–perfect matching case, non–tree blossoms can also be labeled even. We will use the best connection of an even non–tree blossom to augment the matching.

⟨*grow or augment step using best connection of blossom in delta2*⟩≡

```
RESP = delta2.inf(delta2.find_min());
delta2.del_item(RESP->item_in_pq);
RESP->item_in_pq = nil;
if (RESP->label == even) {
  ⟨augment step using best connection of RESP⟩
}
else {
  ⟨grow step using best connection of RESP⟩
}
```

The blossom object *RESP* is retrieved from *delta2* and the minimum item is deleted from *delta2*. If *RESP* is labeled even, an augment step for the best connection of *RESP* is initiated; otherwise, we use the best connection of *RESP* for a grow step. Let us consider the augment step first.

⟨*augment step using best connection of RESP*⟩≡

```
RESP->best_edge(resp, opst);
OPST = _BLOSSOM_OF(opst);
```

We extract the vertices *resp* and *opst*. *resp* is part of the blossom *RESP* and the blossom containing *opst* is denoted by *OPST*. The blossom object *OPST* represents an even tree blossom.

⟨*augment step using best connection of RESP*⟩+≡

```
alternate_path(OPST, item_of);
RESP->base = OPST->mate = resp;
RESP->mate = OPST->base = opst;
RESP->label = unlabeled;
```

The way we augment the matching is as follows. We call *alternate_path* for *OPST*. All edges along the tree path from *OPST* to the root are alternated; *OPST* becomes free. We then match *RESP* and *OPST* and set the label of *RESP* to *unlabeled*.

The tree of *OPST* is destroyed and the priority queues *delta2* and *delta3a* are corrected as discussed before. *free* is decreased by 1 (not by 2), since the number of free vertices with potential larger than zero has been decreased by 1.[14]

⟨*augment step using best connection of RESP*⟩+≡

```
slist<blossom<NT>*> correct;
OPST->my_tree->destroy_tree(correct, delta1, delta3a,
                            delta3b, delta4, Delta, Q, item_of);
correct_pqs(correct, delta2, delta3a);
free -= 1;
```

We come to the grow step. The best connection stored with the unlabeled blossom *RESP* is retrieved.

⟨*grow step using best connection of RESP*⟩≡

```
RESP->best_edge(resp, opst);
OPST = _BLOSSOM_OF(opst);
```

OPST denotes an even tree blossom. We make *RESP* an odd tree blossom of the tree that contains *OPST*.

⟨*grow step using best connection of RESP*⟩+≡

```
RESP->my_tree = OPST->my_tree;
RESP->status_change(odd, Delta, Q);
RESP->pred = opst;
RESP->disc = resp;
if (!RESP->trivial())
  RESP->item_in_pq =
    delta4.insert(compute_potential(RESP, Delta)/2 + Delta, RESP);
```

When *RESP* is non–trivial, we insert a representative item for *RESP* into *delta4*.

The mate blossom *MATE* of *RESP* is also added to the alternating tree. *MATE* becomes an even tree blossom.

[14]Note that all vertices in *RESP* already satisfied the complementary slackness condition (CS)(2) before the augment step, i.e. we have decreased *free* for each of these vertices in some earlier step.

⟨*grow step using best connection of RESP*⟩+≡

```
  node mate = RESP->mate;
  blossom<NT> *MATE = _BLOSSOM_OF(mate);
  MATE->my_tree = OPST->my_tree;
  MATE->status_change(even, Delta, Q);
  if (MATE->item_in_pq) {
    delta2.del_item(MATE->item_in_pq);
    if (!MATE->empty())
      MATE->item_in_pq =
        delta3a.insert((MATE->min_prio() + MATE->offset)/2, MATE);
    else MATE->item_in_pq = nil;
  }
```

When *MATE* has an item in *delta2*, we delete that item. The best connection (if any) of *MATE* to another (distinct) tree is inserted into *delta3a*.

Shrink Step: We next describe the realization of a shrink step. Each item in *delta3b* represents the best candidate edge (for a shrink step) of an alternating tree. First of all, we determine the tree object T whose best candidate edge has (actual) reduced cost zero.

⟨*shrink step using best edge in delta3b*⟩≡

```
  tree<NT> *T = delta3b.inf(delta3b.find_min());
  delta3b.del_item(T->item_in_d3b);
  T->item_in_d3b = nil;
```

The new tight edge e itself is stored in the information part of the minimum item of T's priority queue *d3b_edges*.

⟨*shrink step using best edge in delta3b*⟩+≡

```
  e = T->min_inf();
  T->d3b_edges.del_min();
  resp = source(e); RESP = _BLOSSOM_OF(resp);
  opst = target(e); OPST = _BLOSSOM_OF(opst);
```

resp and *opst* refer to the endpoints of e. We let *RESP* and *OPST* denote the blossoms containing these endpoints. The lowest common ancestor blossom *LCA* and the shrink path *P1* now have to be determined.

⟨*shrink step using best edge in delta3b*⟩+≡

```
  blossom<NT>       *LCA;
  list<node>         P1, P2;
  list<blossom<NT>*> sub1, sub2;
  ⟨determine LCA and shrink path of RESP and OPST⟩
```

The code realizing this has been discussed in detail for the single search tree approach; it is not repeated here. A new blossom object *SUPER* is created and some of its data members are set appropriately.

⟨*shrink step using best edge in delta3b*⟩+≡

```
blossom<NT> *SUPER = new blossom<NT>(LCA->base);
SUPER->mate        = LCA->mate;
SUPER->my_tree     = T;
SUPER->shrink_path = P1;
```

Recall that the immediate subblossom objects are collected in the list *sub1*. For each such object *CUR*, we delete its item (if any) from *delta4* or *delta3a* depending on the status of *CUR*.

⟨*shrink step using best edge in delta3b*⟩+≡

```
forall(CUR, sub1) {
  if (CUR->item_in_pq) {
    if (CUR->label == odd)
      delta4.del_item(CUR->item_in_pq);
    else
      delta3a.del_item(CUR->item_in_pq);
    CUR->item_in_pq = nil;
  }
  SUPER->append_subblossom(CUR, Delta, Q);
}
```

CUR is made a subblossom of the new blossom *SUPER* by calling the member function *append_subblossom*. The implementation of *append_subblossom* differs from the one presented for the single search tree approach. It realizes the non–provident strategy (see Section 2.1.3) as will be discussed below.

⟨*shrink step using best edge in delta3b*⟩+≡

```
SUPER->pot = 2*(SUPER->offset - Delta);
T->add(SUPER);
if (!SUPER->empty())
  SUPER->item_in_pq =
    delta3a.insert((SUPER->min_prio() + SUPER->offset)/2, SUPER);
```

Finally, the stored potential of *SUPER* is set such that its actual potential equals zero. We need to add *SUPER* to *T* and (possibly) insert an item that represents its best connection into *delta3a*. Finally, we delete all dead edges contained in the priority queue *d3b_edges* of *T* as shown below and (if necessary) insert a new representative into *delta3b*.

⟨*shrink step using best edge in delta3b*⟩+≡

```
T->del_dead_edges(item_of);
if (!T->d3b_edges.empty())
  T->item_in_d3b = delta3b.insert(T->min_prio(), T);
```

Deleting all dead (minimum) edges from *d3b_edges* of a given tree object is simple. We simply delete the minimum item from *d3b_edges* until its edge e is alive ($CUR\ != ADJ$), or *d3b_edges* is empty.

⟨*class tree: member functions*⟩+≡

```
void del_dead_edges(const node_array<c_pq_item> &item_of) {
  blossom<NT> *CUR, *ADJ;
  while (!d3b_edges.empty()) {
    edge e = min_inf();
    CUR = blossom_of<NT>(item_of[source(e)]);
    ADJ = blossom_of<NT>(item_of[target(e)]);
    if (CUR != ADJ) break;
    else d3b_edges.del_min();
  }
}
```

The member function *append_subblossom* is realized as follows. Each call makes *CUR* a subblossom of the blossom object.

⟨*class blossom: member functions*⟩+≡

```
void append_subblossom(blossom<NT>* CUR, NT Delta, node_slist &Q) {
  if (CUR->label == odd)
    CUR->status_change(even, Delta, Q);
  if (!CUR->trivial())
    CUR->pot += -2*CUR->offset + 2*Delta;
  if (offset != CUR->offset) {
    ⟨adjust potentials and priorities of smaller group⟩
  }
  CUR->my_tree->remove(CUR);
  concat(*CUR);
  CUR->split_item = last_item();
  subblossom_p.append(CUR);
}
```

As in the single search tree approach, *CUR* is made even, when it refers to an odd subblossom. Moreover, the potential of a non–trivial subblossom is frozen, as explained before. When the *offset* currently assigned to the blossom object differs from the *offset* value of *CUR* we adjust the vertex potentials and associated reduced costs of the smaller group. After that, all actual values (potentials and reduced costs) are computed with respect to the same offset value *offset*. We can, therefore, concatenate the priority queue of *CUR* to the priority queue of the blossom object and append *CUR* to the *subblossom_p* list. *CUR* is removed from its alternating tree.

The ideas underlying the unification of different offset values of two blossoms have been given in Section 2.1.3. We now proceed to present our realization.
The actual potentials and priorities associated with each vertex contained in the current blossom object are computed with respect to the value *offset*. Correspondingly, the actual potentials and priorities of a vertex contained in the subblossom *CUR* are computed with regard to the offset value of *CUR*. First of all, we determine the blossom *SMALL_B* that contains fewer vertices. The other blossom is referred to as *LARGE_B*. The difference of their offset values is stored in *adjustment*.

⟨*adjust potentials and priorities of smaller group*⟩≡

```
blossom<NT>* SMALL_B = (size() < CUR->size() ? this : CUR);
blossom<NT>* LARGE_B = (size() < CUR->size() ? CUR : this);
NT adjustment = SMALL_B->offset - LARGE_B->offset;
```

Next, we iterate over all items of the smaller blossom *SMALL_B*.

⟨*adjust potentials and priorities of smaller group*⟩+≡

```
c_pq_item it;
forall_items(it, *SMALL_B) {
  SMALL_B->inf(it)->pot += adjustment;
  if (SMALL_B->inf(it)->empty()) continue;
  SMALL_B->inf(it)->adjust_priorities(adjustment);
  NT cur_prio = SMALL_B->prio(it);
  if (adjustment < 0)
    SMALL_B->decrease_p(it, cur_prio + adjustment);
  else
    SMALL_B->increase_p(it, cur_prio + adjustment);
}
offset = LARGE_B->offset;
```

For each item *it*, we adjust the potential of the corresponding vertex by *adjustment*. When the priority queue associated with this vertex is not empty, we also need to adjust all priorities contained in this queue. The member function *adjust_priorities* (which will be discussed next) of class *vertex* has been implemented to achieve this. Finally, the priority of *it* is decreased or increased by *adjustment* as well.

The priorities in a priority queue of a vertex object are adjusted by a value *adjustment* as follows.

⟨*class vertex: member functions*⟩+≡

```
void adjust_priorities(NT adjustment) {
  if (adjustment == 0) return;
  node r;
  pq_item it;
  NT cur_prio;
  forall_defined(r, ITEM_OF) {
    it = ITEM_OF[r];
    cur_prio = prio(it);
    if (adjustment < 0)
      p_queue<NT, node>::decrease_p(it, cur_prio + adjustment);
    else {  // simulate increase_p
      node v = inf(it);
      del_item(it);
      ITEM_OF[r] = insert(cur_prio + adjustment, v);
    }
  }
}
```

We iterate over all root vertices r for which an item *it* (type *pq_item*) has been defined. When the value of *adjustment* is smaller than zero, we simply decrease the current priority of *it* by *adjustment*, calling operation *decrease_p*. Otherwise, we simulate an *increase_p* operation by deleting *it* and then inserting *it* with the new priority again. The new item (type *pq_item*) needs to be set for *ITEM_OF*[*r*]. Note that we do in fact need the functionality of the *forall_defined* iterator provided by the data type class *h_array*.

This concludes our description of the implementation details for the shrink step. We next consider the expand step.

Expand Step: Only a few minor changes ensue for the expansion of a blossom. Most details are exactly the same as for the single search tree approach.

⟨*expand step using best blossom of delta4*⟩≡

```
RESP = delta4.inf(delta4.find_min());
delta4.del_item(RESP->item_in_pq);
```

The responsible blossom *RESP* is retrieved from *delta4* and the corresponding item is deleted. After that, *RESP* is expanded by calling the member function *expand*. All details of this function have been discussed for the expand step in the single search tree approach.

⟨*expand step using best blossom of delta4*⟩+≡

```
RESP->expand(Delta);
forall(CUR, RESP->subblossom_p)
  RESP->my_tree->add(CUR);
RESP->my_tree->remove(RESP);
```

What differs here is the way we add each subblossom *CUR* to the tree containing *RESP*, and the way we subsequently remove *RESP* from its tree: we do so by using the member functions *add* and *remove*, respectively.

We restore the matching for the immediate subblossoms, extend the alternating tree and, finally, destroy the blossom object *RESP*. Again, the code realizing this is exactly the same as before.

⟨*expand step using best blossom of delta4*⟩+≡

```
blossom<NT> *BASE = _BLOSSOM_OF(RESP->base);
blossom<NT> *DISC = _BLOSSOM_OF(RESP->disc);
int dist = RESP->restore_matching(BASE, DISC);
⟨extend alternating tree⟩
delete RESP;
```

3.5 Constructing Better Initial Solutions

The performance of both algorithms is considerably improved when a heuristic is used to construct an initial matching and the vertex potentials. We will discuss two heuristics in this section: a *greedy heuristic* and a *fractional matching heuristic*.
The greedy heuristic will set the initial vertex potentials as in the empty matching case and then choose a matching within the tight edges in a greedy fashion. The time

required by this heuristic will be $O(n+m)$.
The fractional matching heuristic first solves the *fractional matching problem*; the fractional matching problem only comprises of constraints (1) and (3) of (WM) or (WPM) (see Section 1.4), respectively. The solution to this problem will be half–integral and, moreover, the edges with value $\frac{1}{2}$ will form vertex disjoint odd length cycles. The initial matching will then consist of all edges having value 1 and of $\lfloor |C|/2 \rfloor$ edges from every odd length cycle C. Constructing an initial matching and the vertex potentials in this way will take time $O(n(m+n\log n))$.

The function

```
int greedy_matching(const ugraph &G, const edge_array<NT> &w,
                    node_array<NT> &pot, node_array<node> &mate,
                    bool perfect);
```

realizes the greedy heuristic and

```
int jump_start(const ugraph &G, const edge_array<NT> &w,
               node_array<NT> &pot, node_array<node> &mate,
               bool perfect);
```

implements the fractional matching heuristic. Given an undirected graph G and a weight function w, either function constructs an initial matching of G and, moreover, returns the vertex potentials in a *node_array pot*. The matching is represented by a *node_array mate*: an edge $e = uv$ is a matching edge iff the endpoints u and v are mates of each other, i.e. $mate[u] == v$ and $mate[v] == u$. The function returns the number of free vertices.

The computed matching and the vertex potentials will satisfy the following conditions:

(C1) the reduced cost of each edge is non–negative,

(C2) each matching edge is tight, and

(C3) when *perfect* is set to *false*: each potential is non–negative.

We present the implementation details of each function in the subsequent sections.

3.5.1 Greedy Heuristic

The idea underlying the construction of a greedy matching is simple. We compute the initial potential $pot[u]$ of each vertex u as for the empty matching, i.e. we set the potential $pot[u]$ to one half of the weight of the heaviest incident edge: $pot[u] = \max\{w_e/2 : e \in \delta(u)\}$. When u is an isolated vertex, we set $pot[u] = 0$, since it will never be matched. The reduced costs of all edges will then satisfy (C1) and, moreover, (C3) is also satisfied.[15]

[15](C3) only holds under the assumption that all edge weights are non–negative. We may make this assumption here, since the weighted matching problem is not affected when a positive constant $c = \max\{|w_e| : e \in E\}$ is added to all edge weights.

⟨*greedy.t: initialize vertex potentials*⟩≡

```
edge e;
node u, v;
pot.init(G, -INFINITY(NT));
forall_nodes(u, G)
  if (outdeg(u) == 0) pot[u] = 0;
forall_edges(e, G) {
  u = source(e);
  v = target(e);
  pot[u] = leda_max(pot[u], (w[e]/2));
  pot[v] = leda_max(pot[v], (w[e]/2));
}
```

After this, we inspect each edge $e = uv$ of G: when e is tight and, moreover, neither u nor v is matched, we make e a matching edge (u is made a mate of v and vice versa). Note that (C2) is met. The number of free vertices is kept in *free*.

⟨*greedy.t: construct greedy matching*⟩≡

```
int free = G.number_of_nodes();
mate.init(G, nil);
forall_edges(e, G) {
  u = source(e);
  v = target(e);
  if ((pot[u] + pot[v] == w[e]) &&
       (mate[u] == nil) && (mate[v] == nil)) {
    mate[v] = u;
    mate[u] = v;
    free -= 2;
  }
}
```

In the non–perfect matching case, the vertex potentials are not restricted to being non–negative. We can therefore tighten the reduced costs of edges that are incident to free vertices.

⟨*greedy.t: adjust vertex potentials in non–perfect case*⟩≡

```
if (perfect) {
  forall_nodes(u, G) {
    if (!mate[u]) {
      NT slack = INFINITY(NT);
      forall_adj_edges(e, u) {
        v = opposite(u, e);
        slack = leda_min(pot[u] + pot[v] - w[e], slack);
      }
      pot[u] -= slack;
    }
  }
}
```

We inspect all edges uv incident to a free vertex u and determine the value *slack*, which refers to the minimum reduced cost of these edges. The reduced cost of each such edge will also stay non–negative when we decrease the value of $pot[u]$ by *slack*.

The complete greedy algorithm to compute an initial matching and the vertex potentials satisfying (C1) to (C3) now reduces to:

⟨*greedy.t: algorithm*⟩≡

```
template<class NT>
int greedy_matching(const ugraph &G, const edge_array<NT> &w,
                    node_array<NT> &pot, node_array<node> &mate,
                    bool perfect) {
  ⟨greedy.t: initialize vertex potentials⟩
  ⟨greedy.t: construct greedy matching⟩
  ⟨greedy.t: adjust vertex potentials in non–perfect case⟩
  return free;
}
```

Obviously, the time required by this function will be $O(n + m)$.

3.5.2 Fractional Matching Problem

Let us consider the linear programming formulation (FWPM) of the so–called *fractional (perfect) matching problem* to a given instance $G = (V, E, w)$.[16] (FWPM) is the linear programming relaxation of (IWPM) presented in Section 1.4.2.

$$\text{(FWPM)} \quad \begin{array}{lrcll} \text{maximize} & w^T x & & & \\ \text{subject to} & x(\delta(u)) & = & 1 & \text{for all } u \in V, \quad (1) \\ & x_{uv} & \geq & 0 & \text{for all } uv \in E. \quad (2) \end{array}$$

The following theorem states that an optimal solution to (FWPM) meets certain requirements.

Theorem 3.5.1 (Half–Integrality of Fractional Matching Problem) Let x be an optimal solution to (FWPM) and let $\mathcal{P}^{(\text{FWPM})}$ denote the convex hull defined by the incidence vectors of (FWPM). Then, x is half–integral, i.e. $x_e \in \{0, \frac{1}{2}, 1\}$ for all $e \in E$. Moreover, the edges e for which $x_e = \frac{1}{2}$ form vertex disjoint odd length cycles if x is a vertex of $\mathcal{P}^{(\text{FWPM})}$.[17]

⌜ We sketch a constructive proof of Theorem 3.5.1. ⌝
First, we show that every optimal solution x to (FWPM) must be half–integral. As mentioned previously (see Section 1.4), Birkhoff [Bir46] proved that every optimal solution to the fractional matching problem is integral when G is restricted to being bipartite. We construct a bipartite

[16] There also exists a *fractional non–perfect matching problem*: (FWPM)(1) is replaced by $x(\delta(u)) \leq 1$ for all $u \in V$. However, we will concentrate on the perfect matching case here. All results to come can easily be transferred to the non–perfect case using the reduction presented in Section 1.5.

[17] At this point we assume that the reader is familiar with certain concepts and results from the field of polyhedral combinatorics. We briefly summarize the two results needed here (for a more extensive discussion see Cook et al. [CCPS98]). (1) A vector v of a polyhedron $\mathcal{P}^{(\text{LP})}$ is a *vertex* of $\mathcal{P}^{(\text{LP})}$ iff v cannot be written as a convex combination of vectors in $\mathcal{P}^{(\text{LP})} \setminus v$. (2) If an optimal solution to a linear program (LP) exists, then (LP) has also an optimal solution x, which is a vertex of the corresponding polyhedron $\mathcal{P}^{(\text{LP})}$.

graph $G' = (A\dot{\cup}B)$ as follows. For each vertex v in G we have a vertex $v' \in A$ and a vertex $v'' \in B$. Each edge $e = uv$ in G corresponds to two edges $e' = u'v''$ and $e'' = u''v'$ in G'. The weight of each edge e' and e'' in G' equals the weight of the corresponding edge e in G. An optimal solution x' to (FWPM) for $G' = (A\dot{\cup}B, E', w')$ will be integral. Thus, choosing $x_e = \frac{1}{2}(x'_{e'} + x'_{e''})$ gives us a half–integral solution which is optimal for the fractional matching problem for $G = (V, E, w)$, as desired.
We now proceed to prove that all edges e with $x_e = \frac{1}{2}$ form vertex disjoint odd length cycles if x is a vertex of $\mathcal{P}^{(\text{FWPM})}$. Clearly, every edge e with $x_e = \frac{1}{2}$ must be part of a cycle, since $x(\delta(u)) = 1$ for all $u \in V$. Moreover, all cycles are vertex disjoint. Let x be an optimal solution and assume there exists an even length cycle C with $x_e = \frac{1}{2}$ for each edge $e \in C$. We show that x is not a vertex of the convex hull $\mathcal{P}^{(\text{FWPM})}$ defined by the incidence vectors of (FWPM). We define a vector d as follows: $d_e = 0$ for all $e \notin C$ and d_e is alternately set to $\frac{1}{2}$ and $-\frac{1}{2}$ for the edges e along C. Then, $x + d$ as well as $x - d$ are feasible solutions to (FWPM) (and at least one of those has objective value larger or equal to that of x). Since x can be written as a convex combination $x = \frac{1}{2}(x + d) + \frac{1}{2}(x - d)$, x cannot be a vertex of $\mathcal{P}^{(\text{FWPM})}$. ⌟

Theorem 3.5.1 gives rise to the idea that one can use an optimal solution of (FWPM) to construct an initial matching M. This idea was put forward by Derigs and Metz [DM86]. We proceed as follows. First, we compute an optimal (vertex) solution x to (FWPM) using a primal–dual method which is similar to (but considerably simpler than) the one developed in Section 1.6. The computed solutions (primal and dual) will meet the following conditions:

(I1) each edge e with $x_e > 0$ is tight,

(I2) the reduced cost of each edge is non–negative,

(I3) in the non–perfect matching case: all vertex potentials are non–negative.

The initial matching M is then constructed as follows. Each edge e with $x_e = 1$ is added to M. Moreover, we add $\lfloor |C|/2 \rfloor$ edges of every odd length cycle C to M. Due to the feasibility of (I1) to (I3), the invariants (C1) to (C3) will hold for M and the computed vertex potentials.

A realization of a primal–dual method for the fractional matching problem is as follows. We describe a single search tree approach. The algorithm starts with an initial matching M ($x_e \in \{0, 1\}$) and vertex potentials such that (I1) to (I3) are met. Initially, every matched vertex is unlabeled and every free vertex is labeled even. The algorithm proceeds in phases. In each phase an alternating tree T is grown from a free vertex r; a vertex r is said to be free in this context, when $x(\delta(r)) = 0$. Only tight edges are used by the algorithm. The details for an alternate step (in the non–perfect case), a grow step and an augment step are identical to those given for the blossom–shrinking approach. However, when a tight edge uv with $u^+ \in T$ and $v^+ \in T$ exists, we proceed differently. x_e is set to $\frac{1}{2}$ for all edges along the encountered odd length cycle C, and the edges along the tree path from the lowest common ancestor of u and v to the root r get alternately unmatched and matched (r becomes matched). After this, all vertices in T are unlabeled and T is destroyed. When a tight edge uv with $u^+ \in T$ and $v^\varnothing \notin T$ is encountered and v is moreover part of a half–valued odd length cycle, the edges along the odd length cycle get alternately unmatched and matched starting in v (v becomes free) and then all edges along the path $p = (v, u, \ldots, r)$ get alternately matched and unmatched (v and r become matched). Following this, all vertices in T get unlabeled and T is destroyed.

A dual adjustment is performed as in the blossom–shrinking approach: each potential of an even tree vertex is decreased by δ, each potential of an odd tree vertex is increased by δ and all other vertex potentials stay the same. The value of δ is only determined by the lower bounds δ_1, δ_2 and δ_3 (see Section 1.6.3).

Implementation: We now come to our implementation. The algorithm can be asked to solve either the fractional perfect matching problem or the fractional non–perfect matching problem (depending on the argument *perfect*). It guarantees a worst–case running–time of $O(n(m + n \log n))$. As before, priority queues are used to determine the value of δ and to identify new tight edges. Most of the ideas presented in the preceding sections are reused.

Besides some standard variables, we have two additional *node_array*s: *label*, which stores the label to each vertex, and *pred*, which stores the predecessor vertex of each odd vertex u in the alternating tree.

⟨*fractional.t: local variables*⟩≡

```
edge e;
node u, v, r;

node_array<int>  label(G);
node_array<node> pred(G, nil);
```

The value of δ is determined by means of the following data structures:

⟨*fractional.t: local variables*⟩+≡

```
NT               delta1;
NT               delta2a;
node_pq<NT>      delta2b(G);

node             resp_d1;
edge             resp_d2a;
node_array<edge> resp_d2b(G);

NT               Delta = 0;
```

delta1 stores the minimum (stored) potential of an even tree vertex *resp_d1*. By *resp_d2a* and *delta2a* we keep track of the best edge that will terminate the current phase. More precisely, *resp_d2a* may denote an edge uv with $u^+ \in T$ and $v^{\{\varnothing|+\}} \notin T$; v will lie on a half–valued cycle if $v^\varnothing \notin T$. The actual value of *delta2a* corresponds to the actual reduced cost of uv. Otherwise, *resp_d2a* refers to an edge uv with $u^+ \in T$ and $v^+ \in T$. Then, the actual value of *delta2a* then equals one half of the actual reduced cost of uv. We use a *node_array resp_d2b* and a *node_pq delta2b* to maintain the best edge uv with $u^+ \in T$ of each vertex $v^\varnothing \notin T$; $v^\varnothing$ is not part of an odd length cycle. *resp_d2b*$[v]$ stores the edge uv and the (actual) priority of *delta2b*$[v]$ refers to the (actual) reduced cost of uv. As before, we accumulate the total amount of dual adjustments in *Delta*:

⟨*fractional.t: local variables*⟩+≡

```
slist<edge>        tight;
slist<node>        Q;
node_slist         T(G);
```

A list *tight* is used to collect all edges that have recently become tight and can thus be used by the algorithm. Q and T are essentially used as in the single search tree algorithm of the blossom–shrinking approach: Q stores all new even vertices, and T keeps all vertices that are part of the alternating tree.

The overall structure of our algorithm is as follows.

⟨*fractional.t: algorithm*⟩≡

```
template<class NT>
int jump_start(const ugraph &G, const edge_array<NT> &w,
               node_array<NT> &pot, node_array<node> &mate,
               bool perfect) {
  ⟨fractional.t: local variables⟩
  ⟨fractional.t: initialization⟩
  forall_nodes(r, G) {
    if (mate[r] || pred[r]) continue;
    ⟨clear priority queues, Q and tight⟩
    pot[r] += Delta;
    T.append(r); Q.append(r);
    bool terminate = false;
    while (!terminate) {
      ⟨scan all edges of vertices in Q⟩
      if (delta1 == Delta) {
        ⟨alternate step using best node of delta1⟩
      }
      else if (!tight.empty()) {
        ⟨use all tight edges⟩
      }
      else {
        ⟨dual adjustment⟩
        ⟨extract tight edges⟩
      }
    }
  }
  ⟨match all odd length cycles⟩
  return free;
}
```

The initialization is simple: we compute a greedy matching and label all vertices appropriately.

⟨*fractional.t: initialization*⟩≡

```
int free = greedy_matching(G, w, pot, mate, perfect);
if (free == 0) return free;
forall_nodes(u, G)
  label[u] = (mate[u] ? unlabeled : even);
```

Next, a phase is initiated for each free vertex r. We use the following convention to determine the value $x(\delta(u))$ to a given vertex u:

$$x(\delta(u)) = \begin{cases} 0 & \text{when } \mathit{mate}[u] = \mathit{nil} \text{ and } \mathit{pred}[u] = \mathit{nil}, \\ \frac{1}{2} & \text{when } \mathit{mate}[u] = \mathit{nil} \text{ and } \mathit{pred}[u] \neq \mathit{nil}, \\ 1 & \text{when } \mathit{mate}[u] \neq \mathit{nil}. \end{cases}$$

For a half–valued odd length cycle C we will set the $\mathit{pred}[u]$ entry of each vertex $u \in C$ such that the cycle can be traversed following these entries, i.e. $C = u, \mathit{pred}[u], \mathit{pred}[\mathit{pred}[u]], \ldots$.

At the beginning of each phase, *delta1* and *delta2a* are reset and *delta2b*, Q and T are made empty.

⟨*clear priority queues, Q and tight*⟩≡

```
delta1 = delta2a = INFINITY(NT);
delta2b.clear();
Q.clear(); tight.clear();
```

The free vertex r is added to T and entered into Q. Due to the status change of r, we have to adjust its potential by $+\mathit{Delta}$ (see formula (2.3), Section 2.1); we do not maintain an offset for each vertex, but instead adjust its potential when a status change occurs. In a while loop, all edges incident to vertices in Q are scanned as will be explained below. Afterwards, we initiate an alternate step when the actual value of *delta1* equals zero (this will only happen in the non–perfect case), or use the tight edges collected in *tight* to extend T. When neither case applies, a dual adjustment is performed.

⟨*dual adjustment*⟩≡

```
NT cand2b = (delta2b.empty() ? \
             INFINITY(NT) : delta2b.prio(delta2b.find_min()));
NT delta = leda_min(delta1, leda_min(delta2a, cand2b));
if (delta == INFINITY(NT) && perfect) {
  mate.init(G, nil);
  return 0;
}
Delta = delta;  // corresponds to Delta += (delta - Delta)
```

When the actual value of *delta1* equals zero, we immediately resume the while loop. When *delta2a* has actual value zero, the responsible edge *resp_d2a* is appended to *tight* (*resp_d2a* is the only element). The next step will terminate the phase; note that Q is empty. Otherwise, all new tight edges are retrieved from *delta2b* and added to *tight*.

⟨*extract tight edges*⟩≡

```
if (delta1 == Delta)
  continue;
else if (delta2a == Delta) {
  tight.append(resp_d2a);
  resp_d2a = nil;
}
else {
  while (!delta2b.empty() &&
          (delta2b.prio(delta2b.find_min()) == Delta)) {
      u = delta2b.del_min();
      tight.append(resp_d2b[u]);
      resp_d2b[u] = nil;
    }
  }
```

Finally, the algorithm terminates with an optimal solution to the fractional matching problem. We alternately match and unmatch ($x_e \in \{0,1\}$) the edges along all existing odd length cycles to obtain the final matching. Exactly one vertex per cycle (which is u below) will become free.

⟨*match all odd length cycles*⟩≡

```
  forall_nodes(u, G)
    if (pred[u]) {
      alternate_cycle(u, mate, pred);
      free++;
    }
```

The function *alternate_cycle* is easily defined as follows.

⟨*fractional.t: helpers*⟩≡

```
  void alternate_cycle(node u, node_array<node> &mate,
                       node_array<node> &pred) {
    node cur1 = pred[u];
    while (cur1 != u) {
      mate[pred[cur1]] = cur1;
      mate[cur1] = pred[cur1];
      node h = pred[cur1];
      pred[cur1] = nil;
      cur1 = h;
      h = pred[cur1];
      pred[cur1] = nil;
      cur1 = h;
    }
    pred[u] = nil;
  }
```

Starting with *cur1* $=$ *pred*$[u]$, we traverse the odd length cycle, alternately matching and unmatching the edges along this cycle by setting the *mate* and *pred* entries appropriately.
All remaining details will be filled in subsequently.

Scanning New Even Vertices: As in the blossom–shrinking approach, all edges incident to any vertex that has recently become an even tree vertex need to be inspected. This is necessary so as to maintain *delta1*, *delta2a* as well as *delta2b* correctly.

⟨*scan all edges of vertices in Q*⟩≡

```
  while (!Q.empty()) {
    u = Q.pop();
    NT pot_u = pot[u] - Delta;
    if (!perfect) {
      ⟨try to improve delta1⟩
    }
    forall_adj_edges(e, u) {
      v = opposite(u, e);
      if (label[v] == odd) continue;
      NT pot_v = pot[v] - (((label[v] == even) && T.member(v)) ? Delta : 0);
      NT pi = pot_u + pot_v - w[e];
      if (pi == 0) {
        ⟨add edge e to tight⟩
      }
      else {
        ⟨prune edges⟩
        if ((label[v] == unlabeled) && mate[v]) {
          ⟨new delta2b edge encountered⟩
        }
        else {
          ⟨new delta2a edge encountered⟩
        }
      }
    }
  }
```

We compute the actual potential *pot_u* for each even tree vertex u in Q. The actual potential of a vertex u will be determined as stated in Section 2.1 (formula (2.1)); the only difference is that no offset exists.
delta1 is only maintained in the non–perfect matching case.

⟨*try to improve delta1*⟩≡

```
  if (pot_u < leda_min(delta1, delta2a) - Delta) {
    delta1 = pot_u + Delta;
    resp_d1 = u;
    if (delta1 == Delta) break;
  }
```

All edges $e = uv$ incident to u are considered. When the adjacent vertex v of u is odd we simply continue, since nothing has to be done. Otherwise, we compute the actual potential *pot_v* of v and the actual reduced cost *pi* of uv. When e is tight, i.e. *pi* equals zero, we add e to the list *tight*.

⟨*add edge e to tight*⟩≡

```
  if ((label[v] == unlabeled) && mate[v]) tight.append(e);
  else {
    tight.clear(); Q.clear();
    tight.append(e);
    break;
  }
```

If we have encountered a tight edge that will terminate the current phase, we proceed as follows. Q and *tight* are emptied and e becomes the only element of *tight*; we break the scanning procedure.

Otherwise, the actual reduced cost *pi* is larger than zero. As for our single search tree algorithm of the blossom–shrinking approach, we prune hopeless edges; i.e. edges whose stored priority in *delta2a* or *delta2b* exceeds the minimum value of *delta1* and *delta2a*.[18]

⟨*prune edges*⟩≡

```
#if !defined(_NO_PRUNING)
if (label[v] == even && T.member(v)) {
  if (pi/2 + Delta >= leda_min(delta1, delta2a)) continue;
}
else if (pi + Delta >= leda_min(delta1, delta2a)) continue;
#endif
```

When v is an unlabeled vertex and does not lie on a half–valued cycle, we check whether or not e is the new best edge for v. If it is, we set *resp_d2b*[v] to e and store the (stored) reduced cost of e in *delta2b*.

⟨*new delta2b edge encountered*⟩≡

```
if (delta2b.member(v)) {
  if (pi < delta2b.prio(v) - Delta) {
    delta2b.decrease_p(v, pi + Delta);
    resp_d2b[v] = e;
  }
}
else {
  delta2b.insert(v, pi + Delta);
  resp_d2b[v] = e;
}
```

When v is not of the kind above, we have discovered a new edge for *delta2a*. A check is performed to determine whether e is the new best edge of *delta2a*; if it is, *delta2a* and *resp_d2a* are set appropriately. Note that *pi* must be halved in the case where v is an even labeled tree vertex.

⟨*new delta2a edge encountered*⟩≡

```
if ((label[v] == even) && T.member(v)) pi /= 2;
if (pi < delta2a - Delta) {
  delta2a = pi + Delta;
  resp_d2a = e;
}
```

Alternate Step: Let us consider the alternate step. The edges along the (even length) tree path from *resp_d1* towards the root r get alternately unmatched and matched.

⟨*alternate step using best node of delta1*⟩≡

```
alternate_path(resp_d1, label, mate, pred);
destroy_tree(T, label, pot, mate, pred, Delta);
label[resp_d1] = even;
terminate = true;
```

[18]Define the token _NO_PRUNING (`#define _NO_PRUNING`) to switch off this strategy.

After this, T is destroyed and the phase terminates. Note that *destroy_tree* will set the label of *resp_d1* to *unlabeled*. We therefore need to correct it to *even*.

We give the implementation details of the function *alternate_path*, which alternates the edge along the tree path starting with the given vertex u.

⟨*fractional.t: helpers*⟩+≡

```
void alternate_path(node u, node_array<int> &label,
                    node_array<node> &mate, node_array<node> &pred) {
  node cur = u;
  node pre = nil, nxt;
  while (cur) {
    if (label[cur] == even) {
      nxt = mate[cur];
      mate[cur] = pre;
      cur = nxt;
    }
    else {
      pre = cur;
      mate[cur] = pred[cur];
      nxt = pred[cur];
      pred[cur] = nil;
      cur = nxt;
    }
  }
}
```

Following the path from u towards the root, the *mate* and *pred* entries are set appropriately for each vertex on the path.

The current alternating tree T is easily destroyed.

⟨*fractional.t: helpers*⟩+≡

```
template<class NT>
void destroy_tree(node_slist &T, node_array<int> &label, node_array<NT> &pot,
                  node_array<node> &mate, node_array<node> &pred, NT Delta) {
  node v;
  while (!T.empty()) {
    v = T.pop();
    if (label[v] == even) pot[v] -= Delta;
    else {
      pot[v] += Delta;
      if (mate[v]) pred[v] = nil;  // only for vertices not on cycle
    }
    label[v] = unlabeled;
  }
}
```

Each vertex v is removed from T. Depending on the status of v, its potential $pot[v]$ needs to be adjusted as stated in formula (2.3). Moreover, when u is an odd vertex, $pred[u]$ is set to *nil*; however, it is crucial that $pred[u]$ is not set to *nil* when u is part of a half–valued cycle.[19]

Using Tight Edges: All edges that can be used by the algorithm are collected in *tight*. In a while loop, we retrieve each such edge $e = uv$ and act accordingly. We ensure that u always denotes a tree vertex.

⟨*use all tight edges*⟩≡

```
while (!tight.empty()) {
  e = tight.pop();
  u = (T.member(source(e)) ? source(e) : target(e));
  v = opposite(u, e);
  if (label[v] == odd || label[u] == odd) continue;
  if (label[v] == unlabeled) {
    if (mate[v]) {
      ⟨grow step using edge e⟩
    }
    else {  // v on half-valued cycle
      ⟨alternate cycle and tree path using edge e⟩
      terminate = true;
      break;
    }
  }
  else {  // label[v] == even
    if (T.member(v)) {
      ⟨construct half valued cycle⟩
      terminate = true;
      break;
    }
    else {
      ⟨augment step using edge e⟩
      terminate = true;
      break;
    }
  }
}
```

It may happen that *tight* stores two edges $e = uv$ and $e' = u'v$ to the same unlabeled vertex v (not lying on a half–valued cycle). Assume e' is used before e. Then, v will be labeled odd, when e is considered later on; e is of no use. We therefore continue with the next tight edge when either of the endpoints v or u is odd.

Grow Step: We turn to the description of a grow step using a tight edge $e = uv$; u is an even tree vertex. v becomes an odd tree vertex with predecessor vertex u.

[19]That we need to take this case into account will become clear later on, when the construction of a half–valued cycle is discussed.

⟨*grow step using edge e*⟩≡

```
label[v] = odd;
pred[v] = u;
pot[v] -= Delta;
T.append(v);
delta2b.del(v);
resp_d2b[v] = nil;
```

The potential of v is adjusted according to its status change. We delete the best edge data for v from *delta2b* and *resp_d2b*.

The mate m of v is also added to T. m becomes an even tree vertex and is added to Q. Its entry in *delta2b* as well as in *resp_d2b* is deleted.

⟨*grow step using edge e*⟩+≡

```
node m = mate[v];
label[m] = even;
pot[m] += Delta;
T.append(m);
Q.append(m);
delta2b.del(m);
resp_d2b[m] = nil;
```

Alternate Cycle and Tree Path: We next consider the case where $e = uv$ is a tight edge with $u^+ \in T$, $v^\varnothing \notin T$ and v lies on a half–valued cycle C. We alternately unmatch and match the edges along C starting at v; the function *alternate_cycle* to achieve this has already been presented. v will afterwards be free.

⟨*alternate cycle and tree path using edge e*⟩≡

```
alternate_cycle(v, mate, pred);
alternate_path(u, label, mate, pred);
mate[u] = v;
mate[v] = u;
destroy_tree(T, label, pot, mate, pred, Delta);
free--;
```

We alternately unmatch and match the edges along the tree path from u to r calling *alternate_path* and finally match u and v with each other. T is subsequently destroyed.

Constructing a Half–Valued Cycle: When a tight edge $e = uv$ is of the kind $u^+ \in T$ and $v^+ \in T$, we proceed as follows. We determine the lowest common ancestor vertex *lca* of u and v by calling the function *seek_lca*. Starting in *lca* the edges along the tree path are alternately unmatched and matched by the function *alternate_path* (*lca* becomes free). Then, a new half–valued cycle $C = (lca, \dots, u, v, \dots, lca)$ is constructed, calling *construct_cycle*. Finally, the tree T is destroyed and *free* is decreased by one, since the root r is now matched.

⟨*construct half valued cycle*⟩≡

```
node lca;
seek_lca(u, v, lca, mate, pred, P1, P2, lock);
alternate_path(lca, label, mate, pred);
construct_cycle(u, v, lca, mate, pred);
destroy_tree(T, label, pot, mate, pred, Delta);
free--;
```

The lowest common ancestor vertex is determined in lock–step fashion as discussed before. We therefore additionally introduce the following local data structures.

⟨*fractional.t: local variables*⟩+≡

```
double              lock = 0;
node_array<double> P1(G, 0);
node_array<double> P2(G, 0);
```

The determination of the lowest common ancestor vertex is achieved as follows.

⟨*fractional.t: helpers*⟩+≡

```
void seek_lca(node u, node v, node &lca,
              node_array<node> &mate, node_array<node> &pred,
              node_array<double> &P1, node_array<double> &P2,
              double &lock) {
  node cur1 = u, cur2 = v;
  P1[cur1] = P2[cur2] = ++lock;
  while ((P1[cur2] != lock) && (P2[cur1] != lock) &&
         (mate[cur1] || mate[cur2])) {
    if (mate[cur1]) {
      cur1 = pred[mate[cur1]];
      P1[cur1] = lock;
    }
    if (mate[cur2]) {
      cur2 = pred[mate[cur2]];
      P2[cur2] = lock;
    }
  }
  if (P1[cur2] == lock)  // cur2 is lca
    lca = cur2;
  else if (P1[cur1] == lock)  // cur1 is lca
    lca = cur1;
  else lca = nil;
}
```

We follow the two tree paths from u and v towards the root r. All even vertices on the path from u to r are marked by *lock* using the *node_array P1* and all even vertices on the path from v to r are marked by *lock* using the *node_array P2*. When either *P1*[*cur2*] or *P2*[*cur1*] equals *lock*, the lowest common ancestor *lca* has been found.

We now discuss the details of the half–valued cycle construction. The idea is simple. Let p_u and p_v denote the two tree paths from u and v to the *lca* vertex, respectively. First, the *pred* entries of all vertices along p_u are set such that they represent the reversed path of p_u. Then, the *pred* entries of all vertices along p_v are set such that they represent p_v itself. Finally, we set *pred*$[u]$ to v and obtain the representation of C as desired.

⟨*fractional.t: helpers*⟩+≡
```
void construct_cycle(node u, node v, node lca,
                     node_array<node> &mate, node_array<node> &pred) {
  node cur1 = u, cur2 = v;
  while (cur1 != lca) {
    // set pred data to reversed tree path; delete mate entries
    node h = mate[cur1];
    mate[cur1] = nil;
    cur1 = pred[h];
    pred[h] = mate[h];
    mate[h] = nil;
    pred[cur1] = h;
  }
  while (cur2 != lca) {
    // set pred data to tree path; delete mate entries
    node h = mate[cur2];
    pred[cur2] = mate[cur2];
    mate[cur2] = nil;
    mate[h] = nil;
    cur2 = pred[h];
  }
  pred[u] = v;
}
```

Augment Step: The only detail that has not been presented yet is how to perform an augment step for a tight edge $e = uv$ with $u^+ \in T$ and $v^+ \notin T$. We first alternately unmatch and match the edges along the tree path from u to the root vertex r (u becomes free). Thereafter, u and v are matched with each other (v becomes unlabeled), the tree T is destroyed and *free* is decreased by 2.

⟨*augment step using edge e*⟩≡
```
alternate_path(u, label, mate, pred);
mate[u] = v;
mate[v] = u;
label[v] = unlabeled;
destroy_tree(T, label, pot, mate, pred, Delta);
free -= 2;
```

This concludes the discussion of all the details involved in the construction of an initial matching M and the vertex potentials by solving the fractional matching problem.

3.6 Experimental Results

We performed several experiments in order to rate the practical efficiency of our algorithms. The comparisons we made are as follows.

(1) Comparison of different strategies: single search tree approach with and without pruning strategy, multiple search tree approach with and without provident strategy.

(2) Comparison of the single search tree approach with the multiple search tree approach, and the effect of using different heuristics.

(3) Comparison of our multiple search tree approach with other matching algorithms available in LEDA.

(4) Comparison of our multiple search tree approach with the currently most efficient algorithm, Blossom IV, of Cook and Rohe [CR97].

In this section we will discuss the results of these comparisons. In summary, they reveal the efficiency of our algorithms in practice. We wish to state that our multiple search tree approach is (at least) competitive to Blossom IV: so far, we have not encountered an instance on which our algorithm is inferior (if the comparison is *fair*, as we are about to explain). However, we would like to leave the decision on whether or not our algorithm is superior to Blossom IV to the reader. We decided so, due to the fact that Blossom IV uses a so–called *price and repair strategy* for complete geometric instances that we have not yet implemented for our algorithm. The price and repair strategy significantly improves the running–time of Blossom IV on these instances. Due to the lack of a similar strategy for our algorithm, the comparisons on complete geometric instances are regarded to be not quite *fair*.

Experimental Setting: We experimented with three kinds of instances: Delaunay instances, (sparse and dense) random instances and complete geometric instances.

For the Delaunay instances we chose n random points in the unit square and computed the Delaunay triangulation (using the LEDA Delaunay implementation). The edge weights correspond to the Euclidean distances scaled to integers in the range $[0, \ldots, 2^{16})$. Delaunay graphs are known to contain perfect matchings (see Dillencourt [Dil90]).

For the random instances we created random graphs with n vertices. The number of edges for sparse graphs was chosen as $m = \alpha n$ for small values of α, $\alpha \leq 10$.
The number of edges for dense graphs is about 20%, 40% and 60% of the density of a complete graph, i.e. $m = dn(n-1)/2$, with $d \in \{0.2, 0.4, 0.6\}$.

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

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

Different Strategies We discuss the influence of the usage of different strategies for each approach. The comparisons were made on sparse random graphs with n vertices and a fixed $\alpha = 10$. Both algorithms computed a maximum–weight matching; the greedy heuristic was used.

The single search tree approach (SST) has been implemented to use a pruning strategy by default. Table 3.1 implies this to be reasonable.

n	α	$\text{SST}^{+}_{\text{pru}^-}$	$\text{SST}^{+}_{\text{pru}^+}$	t
10000	6	37.42	28.17	5
20000	6	125.95	99.60	5
40000	6	428.78	364.67	5

Table 3.1: Effect of pruning strategy for single search tree algorithm (SST).

We compared the single search tree algorithm using the pruning strategy ($\text{SST}_{\text{pru}^+}$) with the single search tree algorithm not using the pruning strategy ($\text{SST}_{\text{pru}^-}$). The running–time of the single search tree algorithm is considerably improved using the pruning strategy. Recall that the pruning strategy is also implemented for the fractional matching heuristic.

For the multiple search tree approach, the user may choose between the provident and the non–provident strategy. As mentioned previously, the non–provident strategy ($\text{MST}_{\text{pro}^-}$) seems to us to be slightly superior to the provident strategy ($\text{MST}_{\text{pro}^+}$). However, the differences are negligible, as indicated in Table 3.2.

n	α	$\text{MST}^{+}_{\text{pro}^+}$	$\text{MST}^{+}_{\text{pro}^-}$	t
10000	6	13.14	12.89	5
20000	6	29.20	28.59	5
40000	6	67.02	66.01	5

Table 3.2: Effect of non–provident strategy for multiple search tree algorithm (MST).

In the subsequent comparisons, we will always use the strategies that are chosen by default. That is, the single search tree approach as well as the fractional matching heuristic use the pruning strategy, and the multiple search tree approach implements the non–provident strategy.

Single Search Tree vs. Multiple Search Tree Approach: We compared the single search tree approach (SST) to the multiple search tree approach (MST) using different heuristics. The results are given in Table 3.3.

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

Table 3.3: SST vs. MST algorithm and effect of greedy and fractional matching heuristics.

Both algorithms computed a maximum–weight perfect matching on Delaunay instances with n vertices. Either no heuristic ($^-$), the greedy heuristic ($^+$) or the fractional matching heuristic (*) was used. The time needed to construct a greedy or a fractional matching is given in columns GY and FM, respectively.

The fractional matching heuristic is computationally more intensive than the greedy heuristic. However, the fractional matching heuristic improves the overall running–time of both algorithms significantly. We draw attention to the fact that the difference between the two heuristics is more pronounced for the single search tree approach. The multiple search tree approach is superior to the single search tree approach.

We will attempt to give an interpretation for the better running–time performance of the multiple search tree approach. We take a closer look at the number of dual adjustments that were performed during the course of the algorithms. Our algorithms can be asked to output certain statistical information (not documented in the preceding sections). We give a sample output below.

```
MST
------------------------------------------------------
INIT:             0.28 sec.
MATCHING:        14.40 sec.
EXTRACT:          0.14 sec.
CHECKER:          0.23 sec.
------------------------------------------------------
ADJUSTMENTS:      7663
SCAN:            29295    5.91 sec. (avg. 0.20 msec.)
GROW:            16593    0.44 sec. (avg. 0.03 msec.)
SHRINK:              8    0.00 sec. (avg. 0.00 msec.)
EXPAND:              5    0.00 sec. (avg. 0.00 msec.)
ALTERNATE:          43    0.00 sec. (avg. 0.00 msec.)
AUGMENT:          4983    7.40 sec. (avg. 1.49 msec.)
DESTROY TREE:    10000    6.54 sec. (avg. 0.65 msec.)
------------------------------------------------------
TOTAL TIME (without checking):   14.84 sec.
```

```
SST
------------------------------------------------------
INIT:             0.19 sec.
MATCHING:        40.07 sec.
EXTRACT:          0.12 sec.
CHECKER:          0.22 sec.
------------------------------------------------------
ADJUSTMENTS:     13428
SCAN:            81237    4.93 sec. (avg. 0.06 msec.)
GROW:            62265    0.86 sec. (avg. 0.01 msec.)
SHRINK:            155    0.11 sec. (avg. 0.71 msec.)
EXPAND:            148    0.16 sec. (avg. 1.08 msec.)
ALTERNATE:         258    0.21 sec. (avg. 0.81 msec.)
AUGMENT:          4983    0.44 sec. (avg. 0.09 msec.)
DESTROY TREE:     5241    0.57 sec. (avg. 0.11 msec.)
------------------------------------------------------
TOTAL TIME (without checking):   40.39 sec.
```

Both algorithms computed a maximum–weight matching on the same random instance with $n = 10000$, $m = 60000$ and edge weights in the range $[0, \ldots, 2^{16})$. No heuristic was used.

We first of all observe that the multiple search tree approach needs to perform fewer dual adjustments than the single search tree approach. This is to be expected; we consider the rate of change Δf of the dual objective value. For the single search tree approach we observed that $\Delta f = -\delta$, when a dual adjustment is performed by δ (cf. discussion on page 30). In the multiple search tree approach, however, we have a decrease by δ for each existing tree. That is, $\Delta f = -t\delta$, where t refers to the number of alternating trees that currently exist when a dual adjustment is performed.

Further, it seems to us that the single search tree approach needs to initiate *needless* steps, since it is forced to search from a fixed free vertex. Note, for example, that the average number of scan, alternate, grow, shrink and expand steps per augmentation differs drastically. However, this statement is vague.

In the comparisons that follow, we chose the multiple search tree approach using the fractional matching heuristic (MST*) as the canonical implementation.

Comparisons to Matching Algorithms in LEDA: LEDA provides an algorithm for each of the four variants of the matching problem introduced in Chapter 1: a maximum–cardinality bipartite matching algorithm (BCM), a maximum–cardinality matching algorithm (GCM), a maximum–weight bipartite matching algorithm (BWM) and a maximum–weight matching algorithm (GWM). The theoretical running–time of the algorithms are as follows: $O(\sqrt{n}m)$ for BCM, $O(nm\alpha(n,m))$ for GCM (α denotes the inverse Ackermann function), $O(n(m + n\log n))$ for BWM and $O(n^3)$ for GWM. For a detailed description of the underlying algorithms and their implementations see the book by Mehlhorn and Näher [MN99, Chapter 7].

We compared our MST algorithm to each of the algorithms. The tests were performed on sparse random graphs with n vertices and αn edges. In the cardinality cases, unit weights ($w_e = 1$) were used by our algorithm. The results can be seen in Table 3.4. Due to the time intensity of GWM, the comparisons were made on small instances only with $n = 10000$.

n	α	BCM	MST*	GCM	MST*	BWM	MST*	GWM	MST*	t
10000	4	0.73	1.14	0.55	1.94	3.67	1.57	585.43	1.44	5
10000	6	0.92	0.67	0.50	0.91	7.81	3.95	883.66	3.96	5
10000	8	1.28	0.74	0.45	1.18	9.82	6.62	897.81	6.25	5
20000	4	1.81	2.69	1.42	5.19	9.30	3.31	–	3.42	5
20000	6	2.23	1.61	1.42	5.06	29.64	11.05	–	10.05	5
20000	8	3.06	1.65	1.29	2.56	35.62	18.05	–	18.53	5
40000	4	5.52	8.04	4.26	9.19	24.45	8.39	–	8.23	5
40000	6	5.92	4.54	3.86	14.10	109.35	32.50	–	30.22	5
40000	8	7.41	4.31	3.66	9.69	128.66	51.13	–	56.46	5

Table 3.4: Comparison of our MST algorithm to the matching algorithms available in LEDA.

We draw attention to the fact that, for bipartite instances, our algorithm is competitive with the specialized algorithms in LEDA. In the bipartite case, the fractional matching heuristic will always compute an optimal matching. That is, MST essentially reduces to the fractional matching algorithm discussed in the preceding section.

Blossom IV: The Blossom IV algorithm of Cook and Rohe [CR97] is the most efficient code currently available for weighted perfect matchings in general graphs. The efficiency of Blossom IV is revealed in two papers:

(1) In [CR97] Blossom IV is compared to the implementation of Applegate and Cook [App93]. It is shown that Blossom IV is substantially faster.

(2) In [App93] the Applegate and Cook implementation is compared to other implementations. The authors show that their code is superior to all other codes.

Blossom IV can be asked to run either a single search tree approach, a multiple search tree approach or a refinement of the multiple search tree approach called the *variable δ approach.* In the variable δ approach, each alternating tree T_{r_i} chooses its own dual adjustment value δ_{r_i} so as to maximize the decrease in the dual objective value. A heuristic is used to make these choices, since the determination of optimum δ_{r_i}'s would be too costly. The experiments in [CR97] show that the variable δ approach is superior to the other approaches in practice.

We compared our MST algorithm to the multiple search tree approach (B4) as well as to the variable δ approach ($B4_{\text{var}}$) of Blossom IV. Blossom IV (B4 and $B4_{\text{var}}$) also uses a fractional matching heuristic to compute an initial matching (indicated by *).

Delaunay Instances: We give the experiments on Delaunay instances with n vertices in Table 3.5.

n	B4*	B4^*_{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

Table 3.5: MST algorithm vs. Blossom IV (B4 and B4_{var}) on Delaunay instances.

Observe that the variable δ approach (B4_{var}) is significantly faster than the multiple search tree approach (B4). Our MST algorithm is competitive to the variable δ approach B4_{var}.

Asymptotics: In Table 3.6 we compared Blossom IV to our MST approach on random instances; we varied n for a fixed $\alpha = 6$.

n	α	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

Table 3.6: MST algorithm vs. Blossom IV (B4 and B4_{var}) on random instances.

Both algorithms, B4_{var} and MST, seem to grow less than quadratically as a function in n. B4_{var} takes about six times as long as our multiple search tree approach MST. In Table 3.7 we additionally varied α.

n	α	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

Table 3.7: MST algorithm vs. Blossom IV (B4 and B4_{var}) on random instances.

A log–log plot indicating the asymptotics of Blossom IV (B4_{var}) and our MST algorithm on random instances ($\alpha = 6$) is depicted in Figure 3.2.

Influence of Edge Weights: Table 3.8 shows the influence of edge weights on the running–time. We took random instances with $m = 4n$ edges and random edge weights in the range $[1, \ldots, b]$ and varied b.

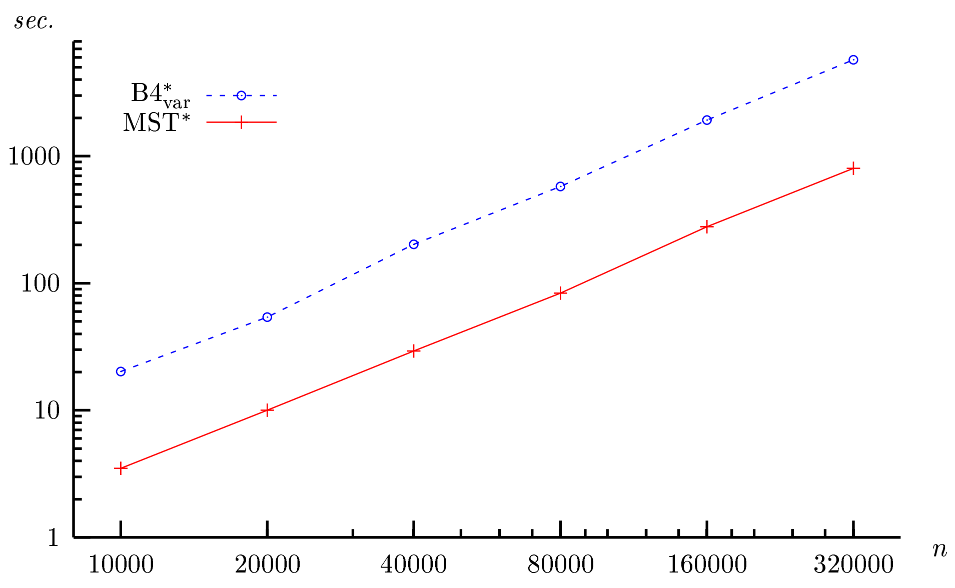

Figure 3.2: Asymptotics of MST algorithm and Blossom IV (B4$_{\text{var}}$) on random instances.

n	α	b	B4*	B4$^*_{\text{var}}$	MST*	t
10000	40000	1	3.98	3.99	0.85	1
10000	40000	10	2.49	3.03	2.31	1
10000	40000	100	3.09	3.10	2.58	1
10000	40000	1000	17.41	8.40	2.91	1
10000	40000	10000	13.69	11.91	2.78	1
10000	40000	100000	12.06	11.20	2.69	1

Table 3.8: MST algorithm vs. Blossom IV (B4 and B4$_{\text{var}}$). Influence of edge weights.

Both B4 and B4$_{\text{var}}$ are sensitive to different edge weights. Their running–time significantly depends on the range of chosen edge weights. The running–time of our MST algorithm is stable (except for the unweighted case ($b = 1$), which is simpler).

We attempt an explanation. When the range of edge weights is small, a single dual adjustment is more likely to produce more than one tight edge. In addition to or as a consequence of this, it seems to us that the number of dual adjustments needed to compute an optimal matching is smaller. Table 3.9 indicates that this assumption is in fact true. We recorded the number of dual adjustments needed by Blossom IV (B4$_{\text{var}}$) and our MST algorithm.

n	α	b	B4$^*_{\text{var}}$	MST*	t
10000	4	1	0	0	1
10000	4	10	2094	2076	1
10000	4	100	5269	5101	1
10000	4	1000	7487	7091	1
10000	4	10000	8063	7877	1
10000	4	100000	8491	8134	1

Table 3.9: MST algorithm vs. Blossom IV (B4 and B4$_{\text{var}}$). Number of dual adjustments.

Since Blossom IV needs time $O(n)$ to perform a dual adjustment, whereas our implementation needs time $O(m \log n)$ for all dual adjustments in a phase, our MST algorithm is less harmed when the edge weights are chosen from a large range.

Observe that, although the variable δ approach ($B4_{\mathrm{var}}$) of Blossom IV was used, our algorithm needs less dual adjustments.

Variance: Table 3.10 gives information about the variance in running–time of Blossom IV ($B4_{\mathrm{var}}$) and our MST algorithm. For each algorithm the best, worst and average time of five random instances, with n vertices and $\alpha = 6$, is given. The fluctuation seems to be about the same for the $B4_{\mathrm{var}}$ and the MST algorithm.

n	α	$B4^*_{\mathrm{var}}$			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

Table 3.10: MST algorithm vs. Blossom IV ($B4_{\mathrm{var}}$) on random instances. Variance.

Dense Random Instances: The experiments suggest that our MST algorithm is superior to $B4_{\mathrm{var}}$ on sparse instances. Table 3.11 shows the running–time on dense instances with n vertices and about 20%, 40% and 60% density. Our MST algorithm is competitive to $B4_{\mathrm{var}}$ on these instances as well.

n	m	B4*	$B4^*_{\mathrm{var}}$	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

Table 3.11: Comparison of MST algorithm to Blossom IV (B4 and $B4_{\mathrm{var}}$). Dense graphs.

Complete Random Instances and Price and Repair: Blossom IV provides a so–called *price and repair heuristic* for complete geometric instances. The instances are implicitly represented by a set of points in an $n \times n$ square (the edge weights correspond to the Euclidean distance). Using the price and repair strategy significantly improves the running–time of Blossom IV on these instances. We have not yet implemented such a heuristic for our algorithm. We compared our MST algorithm to Blossom IV on complete geometric instances ($B4_{\mathrm{var}}$ did not and $B4^{\mathrm{par}}_{\mathrm{var}}$ did use the price and repair strategy). Our algorithm requires an explicit representation of the underlying graph and we thus were only able to experiment with rather small instances. The results are presented in Table 3.12.

n	$B4^*_{\text{var}}$	$B4^{*\text{par}}_{\text{var}}$	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

Table 3.12: MST algorithm vs. Blossom IV ($B4_{\text{var}}$ and $B4^{\text{par}}_{\text{var}}$). Effect of price and repair.

The idea underlying the price and repair heuristic is simple. Instead of running the algorithm on the complete set of edges, the price and repair heuristic starts with a sparse subgraph. Once an optimum weighted matching is computed for the sparse subgraph a check is performed to determine whether or not the computed matching is also optimum for the complete graph. This is what is called *pricing*. Some of the edges having negative reduced cost are added to the current graph, with the matching and the potentials being modified such that all preconditions of the matching algorithm are satisfied. The algorithm is resumed so as to *repair* the matching for the current graph. This process is repeated until the obtained matching is optimum for the complete graph.

There are several natural choices for the selection of the sparse subgraph. For example, a minimum–weight matching will have a natural tendency to avoid heavy edges. Thus, taking the k lightest edges incident to any vertex seems to be a reasonable choice. In fact, this was the way Applegate and Cook [App93] constructed the initial subgraph (they called it the *k–nearest neighbour graph*). Another choice, proposed by Cook and Rohe [CR97], is to use the Delaunay triangulation of the point set as the initial subgraph. For more extensive sources related to the price and repair strategy see Derigs and Metz [DM91], Applegate and Cook [App93] and Cook and Rohe [CR97].

'Worse–case' Instances for Blossom IV: A demanding task would be to implement a generator, which constructs instances that force either algorithm, i.e. our MST or the Blossom IV algorithm, into its worst case. Random graphs tend to be rather simple instances; during the performance of our experiments, for example, many random instances occurred that had been solved almost optimal by the fractional matching heuristic. So far, we have not been able to generate worst–case instances for either algorithm. However, we wish to conclude this section 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 equals 0). Blossom IV ($B4_{\text{var}}$) 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 3.13 shows the results.

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

Table 3.13: Comparison of MST algorithm to Blossom IV ($B4_{\text{var}}$) on chains.

The running–time of Blossom IV 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 optimum 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 $O(n)$. The idea of testing both algorithms on this kind of chains is due to Kurt Mehlhorn (personal communication).

Another 'worse–case' instance for Blossom IV occurred in VLSI–Design having $n = 151780$ vertices and $m = 881317$ edges. Kindly, Andreas Rohe made this instance available to us. We compared the Blossom IV algorithms (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 Table 3.14.

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

Table 3.14: Comparison of MST algorithm to Blossom IV ($\mathrm{B4}_{\mathrm{var}}$) on `boese.edg` instance.

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 optimum 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).

Open Problems

We have described a priority queue based $O(nm \log n)$ algorithm of Edmonds' blossom–shrinking approach. Two implementations, a single search tree and a multiple search tree algorithm, were presented. The additional programming expenditure for the multiple search tree algorithm turned out to be well worth the effort when efficiency in practice is considered.
Our multiple search tree algorithm is competitive with the most efficient known implementation, Blossom IV, due to Cook and Rohe [CR97]. Blossom IV implements a refinement of a multiple search tree approach, called the variable δ approach, and only requires simple data structures. We can thus provide an affirmative answer to the question whether or not sophisticated data structures such as concatenable priority queues help in practice.

Our research raises several questions. (1) The variable δ algorithm is substantially faster than the other algorithms of Blossom IV. Would it be possible to integrate the variable δ approach into a priority queue based $O(nm \log n)$ algorithm? Moreover, it would be interesting to see if an $O(nm \log n)$ variable δ algorithm will improve the practical efficiency as dramatically as for Blossom IV. (2) A price and repair strategy is worth considering for the $O(nm \log n)$ algorithm as well. We expect that such a strategy will improve the running–time of our algorithm on dense and complete instances tremendously. (3) As previously mentioned, a generator of instances forcing either algorithm into its worst case would be of use. (4) Recently, Stefan Näher (personal communication) observed that using a static variant of the *graph* data structure in LEDA (currently, we use a dynamic graph data structure) improves the overall running–time of other graph algorithms by a factor of about two. Most likely, a similar effect can be achieved for our algorithm too. (5) Our fractional matching heuristic also uses priority queue data structures. So far, however, it only implements a single search tree approach. We believe that a fractional matching heuristic based on the multiple search tree approach would further improve the running–time. Possibly, this would also result in a more efficient algorithm for bipartite matching problems. (6) At the end of Chapter 1, we (very roughly) sketched the ideas underlying an $O(n(m + n \log n))$ approach. Although we doubt that an efficient implementation of this approach is possible, it is worth attempting to falsify our hypothesis.

Bibliography

[ADK$^+$00] E. Althaus, D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel. An efficient algorithm for the dominance problem. Submitted, 2000.

[AE85] J. Aráoz and J. Edmonds. A case of non–convergent pivoting in assignment problems. *Discrete Appl. Math.*, 11:95–102, 1985.

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

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows : theory, algorithms and applications.* Prentice Hall, 1993.

[App93] W. Applegate, D. and Cook. Solving large-scale matching problems. In David S. Johnson and Catherine C. McGeoch, editors, *Network Flows and Matchings*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 557–576, Providence, RI, 1993. American Mathematical Society.

[BBD83] M. O. Ball, L. D. Bodin, and R. Dial. A matching based heuristic for scheduling mass transit crews and vehicles. *Transportation Science*, 17:4–31, 1983.

[Bel94] C. E. Bell. Weighted matching with vertex weights: an applicaton to scheduling training sessions in NASA space shuttle cockpit simulators. *European Journal of Operational Research*, 73:443–449, 1994.

[Ber57] C. Berge. Two theorems in graph theory. In *Proceedings of the National Academy of Sciences*, volume 43, pages 842–844, USA, 1957.

[Ber58] C. Berge. Sur le couplage maximum dún graphe. *C. R. Acad. Sci. Paris Sér. I Math.*, 247:258–259, 1958.

[Bir46] G. Birkhoff. Tres observaciones sobre el algebra lineal. *Rev. Fac. Ci. Exactas, Puras y Aplicadas Univ. Nac. Tucuman*, Ser. A 5:147–151, 1946.

[BS00] R. Beier and J. Sibeyn. A powerful heuristic for telephone gossiping. Technical report, Max–Planck Institut für Informatik, 2000.

[BT97] D. Bertsimas and J. N. Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena Scientific, 1997.

[CCPS98] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial optimization.* Wiley-interscience series in discrete mathematics and optimization. Wiley, 1st edition, 1998.

[Chr76] N. Christofides. Worst–case analysis of a new heuristic for the travelling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie–Mellon University, Pittsburgh, PA, 1976.

[Chv83] V. Chvátal. *Linear programming.* W. H. Freeman and Co., 1983.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* The MIT electrical engineering and computer science series. MIT Press, 6th printing edition, 1992.

[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.

[DM92] U. Derigs and A. Metz. A matching–based approach for solving delivery/pickup vehicle routing problem with time constraints. *Operations Research Spektrum*, 14:91–106, 1992.

[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 of Mathematics*, 17:449–467, 1965.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34, 1987.

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

[Gab85] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *26th Annual Symposium on Foundations of Computer Science*, pages 90–100. IEEE Computer Society Press, October 1985.

[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.

[GGS89] H. N. Gabow, Z. Galil, and T. H. Spencer. Efficient implementation of graph algorithms using contraction. *Journal of the ACM*, 36, 1989.

[GKT99] H. N. Gabow, H. Kaplan, and R. E. Tarjan. Unique maximum matching algorithms. To appear in STOC, 1999.

[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.

[GT91] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph matching problems. *Journal of the ACM*, 38:815–853, 1991.

[Kön16] D. König. Über Graphen und ihre Anwendung auf Determinatentheorie und Mengenlehre. *Math. Ann.*, 77:453–465, 1916.

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

[LP86] L. Lovász and M. D. Plummer. *Matching theory*, volume 121/29 of *North–Holland mathematics studies.* North–Holland, 1986.

[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. *LEDA : a platform for combinatorial and geometric computing.* Cambridge University Press, 1999.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization — algorithms and complexity.* Prentice-Hall, 1982.

[Pul95] W. R. Pulleyblank. Matchings and extensions. In Ronald L. Graham, Martin Grötschel, and László Lovasz, editors, *Handbook of combinatorics*, volume 1, pages 179–232, Amsterdam, 1995. Elsevier.

[RT81] E. M. Reingold and R. E. Tarjan. On a greedy heuristic for complete matching. *SIAM Journal of Computing*, 10:676–681, 1981.

[Tut47] W. T. Tutte. The factorisation of linear graphs. *J. London Math. Soc.*, 22:107–111, 1947.