Technische Universität Berlin

FACHBEREICH 3
MATHEMATIK

Implementing Weighted b-Matching Algorithms: Towards a Flexible Software Design

by

Matthias Müller–Hannemann
Alexander Schwartz

No. 591/1998

Implementing Weighted b-Matching Algorithms: Towards a Flexible Software Design

Matthias Müller–Hannemann* Alexander Schwartz*

May, 1998, revised December, 1998

Abstract

We present a case study on the design of an implementation of a fundamental combinatorial optimization problem: weighted b-matching. Although this problem is well-understood in theory and efficient algorithms are known, only little experience with implementations is available. This study was motivated by the practical need for an efficient b-matching solver as a subroutine in our approach to a mesh refinement problem in computer-aided design (CAD). The intent of this paper is to demonstrate the importance of flexibility and adaptability in the design of complex algorithms, but also to discuss how such goals can be achieved for matching algorithms by the use of design patterns. Starting from the basis of the famous blossom algorithm we explain how to exploit in different ways the flexibility of our software design which allows an incremental improvement of efficiency by exchanging subalgorithms and data structures.

1 Introduction

Given an undirected graph $G = (V, E)$ with edge weights c_e for each edge $e \in E$ and node capacities b_v for each node $v \in V$, the *b-matching problem* is to find a maximum weight integral vector $x \in \mathbb{N}_0^{|E|}$ satisfying $\sum_{e=(v,w)} x_e \le b_v$ for all $v \in V$. If, in addition, equality is required to hold in these inequalities for all nodes, the b-matching problem is called *perfect*.

Weighted b-matching is a cornerstone problem in combinatorial optimization. Its theoretical importance is due to the fact that it generalizes both ordinary weighted matching (i. e. matching with all node capacities equal to one, 1-matching) and minimum cost flow problems. All these problems belong to a 'well-solved class of integer linear programs' [EJ70] in the sense that they all can be solved in (strongly) polynomial time. There are excellent surveys on matching theory by Gerards [Ger95], Pulleyblank [Pul95], and Lovász & Plummer [LP86].

Applications. Important applications of weighted b-matching include the *T-join problem*, the *Chinese postman problem*, shortest paths in undirected graphs with negative costs (but no negative cycles), the 2-factor relaxation for the symmetric traveling salesman problem (STSP), and capacitated vehicle routing [Mil95]. For numerous other examples of applications of the special cases minimum cost flow and 1-matching we refer to the book of Ahuja, Magnanti, and Orlin [AMO93].

A somewhat surprising new application of weighted b-matching stems from quadrilateral mesh refinement in computer-aided design [MMW97, MM97]. Given a surface description of

*Technische Universität Berlin, Department of Mathematics, Sekr. MA 6–1, Str. d. 17.Juni 136, D 10623 Berlin, Germany, {mhannema,schwartz}@math.tu–berlin.de, http://www.math.tu–berlin.de/~{mhannema,schwartz}.

Both authors were partially supported by the special program "Efficient Algorithms for Discrete Problems and Their Applications" of the Deutsche Forschungsgemeinschaft (DFG) under grants Mo 446/2-2 and Mo 446/2-3.

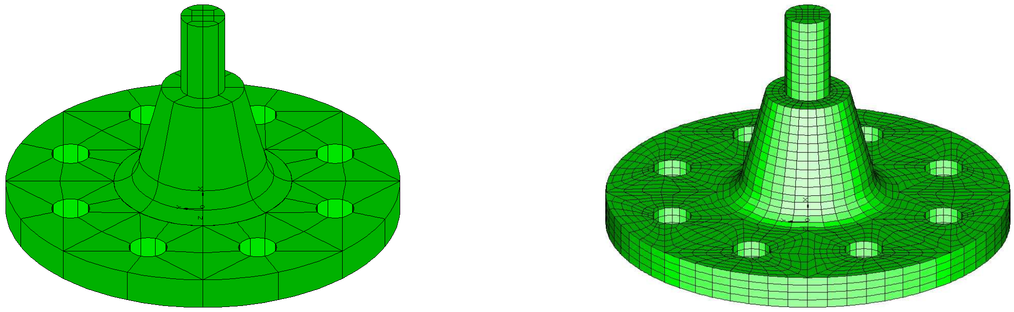

Figure 1: The model of a flange with a shaft and its refinement by our algorithm.

some workpiece in three-dimensional space as a collection of polygons (for example, a model of a flange with a shaft, see Fig. 1), the task to refine the coarse input mesh into an all-quadrilateral mesh can be modeled as a weighted perfect b-matching problem (or, equivalently, as a bidirected flow problem). This class of problem instances is of particular interest because unlike the previous examples, the usually occurring node capacities b_v are quite large (in principle, not even bounded in $\mathcal{O}(|V|)$) and change widely between nodes.

Both authors have been engaged in a research project together with partners in industry where this approach to mesh refinement has been developed. To the best of our knowledge, there is no publicly available code for weighted b-matching problems. Therefore, we first took the obvious formulation of weighted perfect b-matching problem as an integer linear program (ILP) and used CPLEX and its general purpose branch & and bound facilities to solve our instances. The corresponding experiences allowed us to conclude that the modeling of the mesh refinement problem as a weighted perfect b-matching problem captures the requirements of a high-quality mesh refinement very successfully [MH97]. The drawback, however, of such an ad-hoc solution was manifold: first, we encountered instances which could not be solved with CPLEX to optimality within reasonable time and space limits; second, we found the average running time too large for a convenient use in an interactive CAD-system, and third, our partners in industry prefer a software solution independent from commercial third-party products like CPLEX. Hence, the practical need to solve the mesh refinement problems in an efficient and robust way led us to work on our own implementation of weighted b-matching.

Design goals and main features of our implementation. As mentioned above, matching problems are well-understood in theory. Nevertheless the implementation of an algorithm for weighted b-matching is a real 'challenge'.

An efficient and adaptable implementation requires a sophisticated software design. A recent systematic discussion of this design problem in the general context of graph algorithms can be found in [Wei98]. In his paper, Weihe uses Dijkstra's algorithm as a running example to demonstrate what flexible adaptability of an algorithm component really means. The complications which arise in our context are due to the fact that we have to perform quite complex graph operations, namely shrinking and expanding of subgraphs, the famous "blossoms."

Design patterns capture elegant solutions to specific design problems in object-oriented software design which support reuse and flexibility. See the book of Gamma et al. [GHJV95] for an excellent introduction to design patterns. Our approach uses the design patterns *strategy, observer* and *iterator* which are well-known from [GHJV95] as well as *data accessor, loop kernel,* and *adjacency iterator* introduced by Kühl & Weihe [KW96, KW97]. The present paper serves as an empirical case study in the application of design principles.

As there are many promising variants of b-matching algorithms, but not too much practical experience with them, we decided to develop a general framework which captures all of these variants. This framework enabled us to do a lot of experiments to improve the performance of our code incrementally by exchanging subalgorithms and data structures. We thereby got an efficient code which solves all instances from our mesh refinement application very well, but seems to be also fast on other classes of instances. Details can be found in an accompanying computational study [MS99].

Previous work. Most work on matching problems is based on the pioneering work of Edmonds [Edm65b, Edm65a, Edm67]. "Blossom I" by Edmonds, Johnson, and Lockhart [EJL69] was the first implementation for the *bidirected flow problem* (which is, as mentioned above, equivalent to the b-matching problem).

Pulleyblank [Pul73] worked out the details of a blossom-based algorithm for a mixed version of the perfect and imperfect b-matching in his Ph. D. thesis and gave a PL1 implementation, "Blossom II". His algorithm has a complexity of $\mathcal{O}(|V||E|B)$ with $B = \sum_{v \in V} b_v$, and is therefore only pseudo-polynomial.

Cunningham & Marsh [Mar79] used scaling to obtain the first polynomial bounded algorithm for b-matching. In principle, b-matching problems may be reduced to 1-matching problems by standard transformations (see for example Gerards [Ger95, pp.179–182]). However, the enormous increase in problem size makes such an approach infeasible for practical purposes. Gabow [Gab83] proposes an efficient reduction technique which avoids increasing the problem size by orders of magnitude. Together with scaling this approach leads to an algorithm with a bound of $\mathcal{O}(|E|^2 \log |V| \log B_{max})$ where B_{max} is the largest capacity.

Edmonds showed that a b-matching problem can be solved by solving *one* polynomially sized general network flow problem and *one* polynomially sized perfect 1-matching problem (for the details, see again Gerards [Ger95, pp.186-187]). Hence, this observation leads to a strongly polynomial algorithm if both subproblems are solved by strongly polynomial algorithms.

Anstee [Ans87] suggested a staged algorithm. In a first stage, the fractional relaxation of the weighted perfect b-matching is solved via a transformation to a minimum cost flow problem on a bipartite graph, a so-called *Hitchcock transportation problem*. In stage two, the solution of the transportation problem is converted into an integral, but non-perfect b-matching by rounding techniques. In the final stage, Pulleyblank's algorithm is invoked with the intermediate solution from stage two. This staged approach also yields a strongly-polynomial algorithm for the weighted perfect b-matching problem if a strongly polynomial minimum cost flow algorithm is invoked to obtain the optimal fractional perfect matching in the first stage. The best strongly polynomial time bound for the (uncapacitated) Hitchcock transportation problem is $\mathcal{O}((|V| \log |V|)(|E| + |V| \log |V|)$ by Orlin's excess scaling algorithm [Orl88], and the second and third stage of Anstee's algorithm require at most $\mathcal{O}(|V|^2|E|)$.

Derigs & Metz [DM86] and Applegate & Cook [AC93] reported on the enormous savings using a fractional "jump start" of the blossom algorithm for weighted 1-matching. Miller & Pekny [MP95] modified Anstee's approach. Roughly speaking, instead of rounding on odd disjoint half integral cycles, their code iteratively looks for alternating paths connecting pairs of such cycles.

Padberg & Rao [PR82] developed a branch & cut approach for weighted b-matching. They showed that violated odd cut constraints can be detected in polynomial time by solving a minimum odd cut problem. Grötschel & Holland [GH85] reported a successful use of cutting planes for 1-matching problems. However, with present LP-solvers the solution time required to solve only the initial LP-relaxation, i. e. the fractional matching problem, is often observed to be in the range of the total run time required for the integral optimal solution by a pure combinatorial approach. Therefore, we did not follow this line of algorithms in our experiments.

With the exception of the paper by Miller & Pekny [MP95] we are not aware of a computational study on weighted b-matching. However, many ideas used for 1-matching can be reused and therefore strongly influenced our own approach. For example, Ball & Derigs [BD83] provide a framework for different implementation alternatives, but focus on how to achieve various asymptotical worst case guarantees.

For a recent survey on computer implementations for 1-matching codes, we refer to [CR97]. In particular, the recent "Blossom IV" code of Cook & Rohe [CR97] seems to be the fastest available code for weighted 1-matching on very large scale instances. We believe that almost all previous approaches for 1-matching are not extendible to b-matching. One reason is that one usually exploits for efficiency reasons the fact that each node can have at most one incident matched edge. Some implementations also assume that edge costs are all non-negative. The mesh refinement application, however, uses arbitrary cost values.

It seems that, in general, implementation studies focus on performance issues and do not address reuseability.

Overview. The rest of the paper is organized as follows. In Section 2 we give a brief review of the blossom algorithm as described by Pulleyblank. It will only be a simplified high-level presentation, but sufficient to discuss our design goals in Section 3 and to outline our solution in Section 4 afterwards. Finally, in Section 5 we summarize the advantages and disadvantages of our approach.

2 An Outline of the Blossom Algorithm

In this section, we give a rough outline of the primal-dual algorithm following the description by Pulleyblank [Pul73]. The purpose of this sketch is only to give a basis for the design issues to be discussed later, and to point out some differences to the 1-matching case. A self-contained treatment is given in Appendix A.

For an edge set $F \subseteq E$ and a vector $x \in \mathbb{N}_0^{|E|}$, we will often use the implicit summation abbreviation $x(F) := \sum_{e \in F} x_e$. Similarly, we will use $b(W) := \sum_{v \in W} b_v$ for a node set $W \subset V$.

Linear programming formulation. The blossom algorithm is based on a linear programming formulation of the maximum weighted perfect b-matching problem. To describe such a formulation, the *blossom description*, let $\Omega := \{ \; S \subset V \mid |S| \geq 3 \text{ and } |b(S)| \text{ is odd } \}$ and $q_S := \frac{1}{2}(b(S) - 1)$ for all $S \in \Omega$. Furthermore, for each $W \subset V$ let $\delta(W)$ denote the set of edges that meet exactly one node in W, and $\gamma(W)$ the set of edges with both endpoints in W. Then, a maximum weight b-matching solves the linear programming problem

$$
\begin{array}{llcll}
 & \multicolumn{4}{l}{\text{maximize } c^T x} \\
 & \multicolumn{4}{l}{\text{subject to}} \\
\text{(P1)} & x(\delta(v)) & = & b_v & \text{for } v \in V \\
\text{(P2)} & x_e & \geq & 0 & \text{for } e \in E \\
\text{(P3)} & x(\gamma(S)) & \leq & q_S & \text{for } S \in \Omega.
\end{array}
$$

The dual of this linear programming problem is

$$
\begin{array}{llcll}
 & \multicolumn{4}{l}{\text{minimize } y^T b + Y^T q} \\
 & \multicolumn{4}{l}{\text{subject to}} \\
\text{(D1)} & y_u + y_v + Y(\Omega_\gamma(e)) & \geq & c_e & \text{for } e = (u,v) \in E \\
\text{(D2)} & Y_S & \geq & 0 & \text{for } S \in \Omega
\end{array}
$$

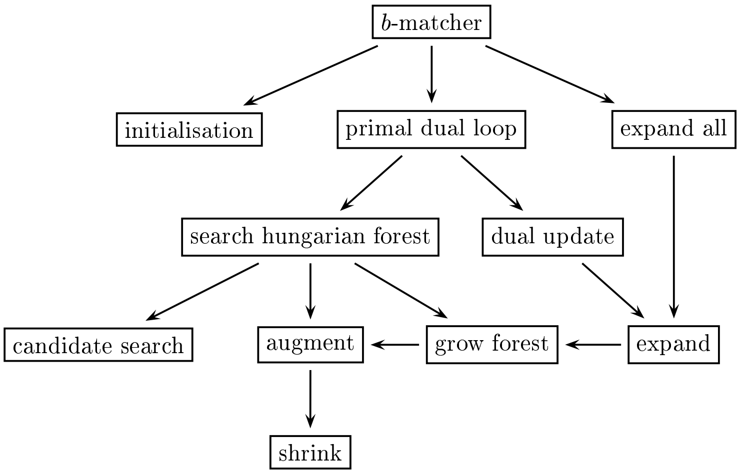

Figure 2: The structure of the blossom algorithm.

with $\Omega_\gamma(e) := \{ S \in \Omega \mid e \in \gamma(S) \}$.

We define the *reduced costs* as $\bar{c}_e := y_u + y_v + Y(\Omega_\gamma(e)) - c_e$ for all $e \in E$. A b-matching x and a feasible solution (y, Y) of the linear program above are optimal if and only if the following complementary slackness conditions are satisfied:

$$\begin{array}{llcrl} \text{(CS1)} & x_e > 0 & \Longrightarrow & \bar{c}_e = 0 & \text{for } e \in E \\ \text{(CS2)} & Y_S > 0 & \Longrightarrow & x(\gamma(S)) = q_S & \text{for } S \in \Omega. \end{array}$$

A primal-dual algorithm. The primal-dual approach starts with some not necessarily perfect b-matching x and a feasible dual solution (y, Y) which satisfy together the complementary slackness conditions (CS1) and (CS2). Even more, the b-matching x satisfies (P2) and (P3). Such a starting solution is easy to find, in fact, $x \equiv 0$, $y_v := \frac{1}{2}\max\{c_e | e = (v, w) \in E\}$ for all $v \in V$ and $Y \equiv 0$ is a feasible choice.

The basic idea is now to keep all satisfied conditions as invariants throughout the algorithm and to work iteratively towards primal feasibility. The latter means that one looks for possibilities to augment the current matching.

To maintain the complementary slackness condition (CS1) the search is restricted to the graph induced by edges of zero reduced costs with respect to the current dual solution, the so-called *equality subgraph* $G^=$. In a *primal step* of the algorithm, one looks for a maximum cardinality b-matching within $G^=$.

We grow a forest F which consists of trees rooted at nodes with a *deficit*, i. e. with $x(\delta(v)) < b_v$. Within each tree $T \in F$ the nodes are labeled *even* and *odd* according to the parity of the number of edges in the unique simple path to the root r (the root r itself is even). In addition, every even edge of a path from the root r to some node $v \in T$ must be matched, i. e. $x_e > 0$. *Candidate edges* to grow the forest are edges where one endpoint is labeled even and the other is either unlabeled or labeled even.

Augmentations are possible if there is a path of odd length between two deficit nodes on which we can alternatively add and subtract some δ from the current matching x without violating primal feasibility. Observe that we can augment if there is an edge between two even nodes of different trees of F. In some cases, an augmentation is also possible if we have such an edge between even nodes of the same tree, but not always. It is the latter case which is responsible for complications. If no augmentation is possible and there is no further edge available to grow

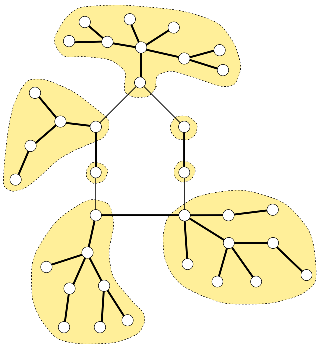

Figure 3: A sample blossom with an odd circuit of length seven. Each shaded region corresponds to a petal.

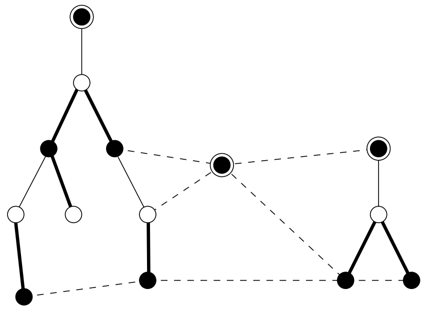

Figure 4: Example of an augmenting forest consisting of three trees. Even (odd) nodes are filled (non-filled), root nodes equipped with an extra circle, non-forest edges are dashed, matched (unmatched) edges are drawn with thick (thin) lines.

the forest, the forest is called *Hungarian forest*.

Edmonds' key insight was the observation that by *shrinking* of certain subgraphs (the *blossoms*) one can ensure that the tree growing procedure detects a way to augment the current b-matching, if the matching is not maximum. The reverse operation to shrinking is *expanding*. Hence, we are always working with a so-called *surface graph* which we obtain after a series of shrinking and expanding steps.

The main difference to 1-matching lies in the more complicated structure of the *blossoms* which we have to shrink into *pseudo-nodes*. Roughly speaking, when blossoms in the 1-matching case are merely formed by odd circuits C for which $x(\gamma(C)) = q_C$, a blossom B in the b-matching case contains such an odd circuit C but also the connected components of matched edges incident to nodes of C, the so-called *petals*. The additional complication is that C must be the only circuit of the blossom. (See Figure 3 for a sample blossom.)

Hence, in order to detect such blossoms efficiently it is suitable to maintain additional invariants on the structure of the current non-perfect b-matching which are trivially fulfilled in the 1-matching case. Namely, each connected component M of matched edges in the surface graph contains no even circuit, at most one odd circuit and at most one deficit node. Even more, if such a component M contains an odd circuit, then M contains no deficit node.

If the primal step finishes with a maximum cardinality matching which is perfect, we are done and the algorithm terminates the primal-dual loop. Otherwise, we start a dual update step. Roughly speaking, its purpose is to alter the current dual solution such that new candidate edges are created to enter the current forest F. Depending on the label of a node and whether it is an original node or a pseudo-node we add or subtract some ε (but leave unlabeled nodes unchanged to maintain (CS1)) which is chosen as the maximum value such that the reduced cost of all edges in the forest F remain unchanged (i. e. they remain in $G^=$), all other edges of the original G have non-negative reduced costs (D1), and the dual variables associated to pseudo-nodes remain non-negative (D2). If the dual variable associated to an odd pseudo-node becomes zero after a dual update, the pseudo-node will be expanded. This guarantees that no augmenting paths will be missed.

After finishing the primal-dual loop, all remaining pseudo-nodes are expanded, and the algorithm terminates. See Figure 2 for an overview on the structure of the primal-dual algorithm.

3 Design Goals

Specialized to our application, the most important general requirements on the flexibility of a design imply that the implementation of the blossom algorithm framework should have the following features:

- **Decoupled data structures and algorithms.** As software is often (in particular in our case) implemented first as a prototype, but later refined step-by-step to improve efficiency, the necessary modification should only affect small pieces of the code. The latter requires that both data structures and algorithms are exchangeable almost independently of each other.

  A basic design decision concerns the interplay between the blossom algorithm and the representation of graphs. The difficulty lies in the fact that the view on the graph objects changes throughout the algorithm: simultaneously, we have the original input graph, (moreover, in case of dense graphs it is useful to work on a sparse subgraph), then we have the *equality subgraph* (induced by edges of zero reduced costs), and finally the current *surface graph*, which is derived from the equality subgraph by blossom shrinking operations.

- **Exchangeable subalgorithms.** It should be easy to replace subalgorithms for at least two reasons:

  1. **Problem variants.** Suppose we apply our framework to a special case of b-matching, for example to ordinary 1-matching, to unweighted (i. e. cardinality) matching, or to matching with additional edge capacities, in particular to so-called *factor problems* where all edge capacities are set to one. For all such problem variants, the standard implementation of some subalgorithms (but only few!) should be exchangeable with an adapted version which is fine-tuned towards efficiency.
  2. **Algorithmic variants.** Within our framework we would like to test, for example, different dual update strategies, or exchange a forest with a single tree implementation, or apply heuristics to avoid the shrinking of blossoms.

- **Exchangeable data structures.** The candidate search for edges in the forest growing part of the blossom algorithm is an example for which we would like to explore different priority queue implementations.

- **Evaluation strategies.** Certain mathematical functions and terms have to be evaluated so often during the execution of an algorithm, that different evaluation strategies may reduce the overall computational costs significantly. Well-known techniques such as "lazy evaluation" (calculate a value only when it is needed), "over-eager evaluation" (calculate a value before it is needed), and "caching" (store each calculated value as long as possible) can, for example, be applied to the evaluation of reduced costs, maintaining dual potentials or node deficits.

- **Separate initialization and preprocessing.** A blossom algorithm either starts with an empty matching, some greedy matching, or it uses a jump-start solution. In many applications, the size of an instance can be reduced in a preprocessing step, for example by a special handling of isolated or degree-1 nodes, parallel edges or more complicated special structures.

- **Exchangeable graph classes.** The framework has to be adaptable to special graph classes. The standard implementation does not assume anything about special properties of the graph classes. However, if it is known in advance, that one wants to solve matching problems on special graph classes, such as planar graphs, Eucledian graphs or complete graphs, it should be possible to exploit the additional structure of such a class.

- **Algorithm analysis.** We want to be able to get statistical information from the execution of our code. Operation counting [AO96] is a useful concept for testing algorithms, as it can help to identify asymptotic bottleneck operations in an algorithm, to estimate the algorithm's running time for different problem sizes, and to compare algorithmic variants. Furthermore, such statistics gives additional insight into the relationship of a class of instances and its level of difficulty for a blossom algorithm, for example by counting the number of detected blossoms or the maximum nesting level of blossoms.

- **Robust, self-checking.** A robust algorithm should (be able to) check all invariants and pre- and postconditions. It has to terminate with a deterministic behavior in case of a violation. In particular, each violation of one of these conditions that indicates an implementation bug is found immediately. This reduces the total time spend with debugging dramatically.

4 An Object-Oriented Implementation

In this section we will outline our solution with respect to the desired goals. This discussion does not exhaust all of our design goals, but will highlight those aspects which might be most interesting. We formulate some features of our approach as general principles.

4.1 Decoupling of algorithms and data structures

A major obstacle on flexibility arises if algorithms and data structures are tightly coupled and large portions of the code depend directly on the concrete data structures.

Principle *Decouple algorithms from data structures.*

We first give an example why this decoupling is useful in the context of matching algorithms and describe its realization in the following two subsections on iterators and data accessors.

The primal-dual blossom algorithm uses different categories of graphs, namely the original graph, the equality subgraph and the surface graph. A closer look into the algorithm shows that we do not need to represent the equality graph explicitly. However, the internal representation of a graph where the node set remains static throughout the graph's lifetime is certainly different from a graph which must provide shrink and expand operations on its own subgraphs. Hence, we use two basic graph classes for the different cases (`unshrinkable_graph` and `surface_graph`). Below, we will give an example where the same algorithm is once used with with an instance of `unshrinkable_graph` and once with `surface_graph`. As shrinking of nodes can be nested, it is useful to have a map between an original node u and the corresponding pseudo-node or node in the surface graph, denoted by $outer(u)$.

4.2 Iterators

For graph algorithms, an appropriate way to decouple algorithms from data structures is given by the following principle.

Principle *A graph algorithm uses edge, node and adjacency iterators.*

An *iterator* provides a way to access the elements of an aggregate object sequentially. The underlying representation of the aggregate object remains hidden. Kühl & Weihe [KW96] applied this idea to graph algorithms. They introduced the following categories of iterators:

- **Node iterator.** A node iterator iterates over all nodes of a graph.
- **Edge iterator.** An edge iterator iterates over all edges of a graph.
- **Adjacency iterator.** An adjacency iterator iterates over all edges and nodes which are adjacent to a fixed node. It provides operations for requesting if there is a current adjacent node, for requesting the current adjacent edge and the current adjacent node as well for constructing a new adjacency iterator which iterates over the adjacency of the current adjacent node.

In our context, we want to hide the concrete representation of our surface graph. For example, the client of an instance of a `surface_graph` should not know whether the adjacency list of a pseudo-node is built explicitly as a list or if it is only implicitly available by an iteration through the contained nodes.

In general, our adjacency iterators are implemented as *skip iterators* which run through the whole adjacency of a node, decide for each edge whether it is "present" in the current graph or not, and show an edge only in the affirmative case but skip it otherwise. The decision whether an edge is present or not is based on an evaluation of the predicate ($\mathit{reduced\ costs}(e) == 0$) or ($\mathit{outer}(u) \neq \mathit{outer}(v)$) for an edge $e = (u, v)$. This means that we have different adjacency iterators for each specific view of a node onto its adjacency.

Recall that the surface graph contains two different types of nodes, namely original nodes and pseudo-nodes. This implies that one needs two different types of adjacency iterators. To be more precise, we use a pair of adjacency iterators, one for pseudo-nodes and one for original nodes. For each node of the surface graph, only one of them is valid. This pair of iterators is encapsulated in such a way that the client sees only a single iterator.

4.3 Data accessors

The *data accessor* pattern, introduced by Kühl & Weihe [KW97], provides a solution for the design problem how to encapsulate an attribute of an object or a mathematical expression such that the real representation or computation is hidden from the client.

Principle *Model access to attributes of objects like edges or nodes as a data accessor.*

There are several applications of this pattern in our context where it appears to be useful to hide the underlying representation of the data. A first example is the treatment of the deficit of a node. Possible evaluation strategies are to store and to update the node deficit explicitly, or to calculate it when it is needed from the current matching x and the node capacity b_v. A second example concerns the maintenance of the cost of an edge if the edge costs are induced by some metric distance function between coordinates of its endpoints. Here, it might be useful to calculate edge costs only on demand.

Moreover, it is a good idea to encapsulate the calculation of reduced costs. One reason is that there are several linear programming descriptions of the b-matching problem which can be used as alternatives in our algorithm. For simplicity, we only presented the blossom description, but the so-called *odd-cut description* [BD83, CR97] (see Appendix A) can be used with minor changes. One concrete difference lies in the calculation of the reduced costs. Hence, in order to evaluate which description is superior to the other, one would like to exchange silently the internal calculation of the reduced costs. (For a replacement of the linear description a second small change is necessary in the dual update.)

Finally, we mention that checking and debugging of alternative implementations can be facilitated by using data accessors. The idea is to use temporarily a data accessor which evaluates alternatives we want to check against each other and reports all cases where differences occur.

4.4 Exchange of subalgorithms

The *strategy pattern* [GHJV95] encapsulates each subalgorithm and defines a consistent interface for a set of subalgorithms such that an algorithm can vary its subalgorithms independently from the user of the algorithm. This leads to the following principle.

> **Principle** *Use the strategy pattern for subalgorithms of the framework.*

To facilitate the application of the strategy pattern, we use another principle.

> **Principle** *Each algorithm (and subalgorithm) should be implemented as a separate algorithmic class.*

Thus, Fig. 2 which gives an overview on the structure of the primal-dual algorithm also represents our algorithmic classes. (A number of lower level classes are omitted for brevity).

We elaborate on the use of different strategies taking the example of the candidate search for edges in the tree growing part of the algorithm. (It should be noted that such candidates can effectively be determined within the dual update.)

1. The direct approach to find candidate edges for the growing of the current forest F is to traverse all trees of F and to examine iteratively all edges adjacent to even nodes.

2. Ball & Derigs [BD83] proposed to keep a partition of the edge set into subsets according to the labels of endpoints. This gives some overhead to update these lists, but avoids to examine all those edges for which one endpoint is labeled `even` and the other is labeled `odd`.

3. A refinement of both previous strategies is to mark nodes as safe after the examination of their adjacency. As long as the label of a node does not change, the node remains safe and can be ignored for the further candidate search. An application of the *observer pattern* (which we describe below) ensures that any change in the state of a node label triggers an appropriate update, i. e. nodes become unsafe and will be considered in the candidate search.

4. Usually, each blossom which is detected will be shrunken immediately. As shrinking and expanding of blossoms is computationally very expensive it is useful to avoid shrinking operations heuristically. Each time a blossom forming edge has been detected, we do not shrink the blossom but store the edge instead. Only if no other candidate edges are left, we request the first blossom forming edge and shrink the corresponding blossom.

5. It is a fundamental strategic question whether one should perform the growing of trees simultaneously in a forest (as we did in our description) or to grow single trees one after another, i. e. to grow a tree of our forest until it becomes Hungarian and to switch then

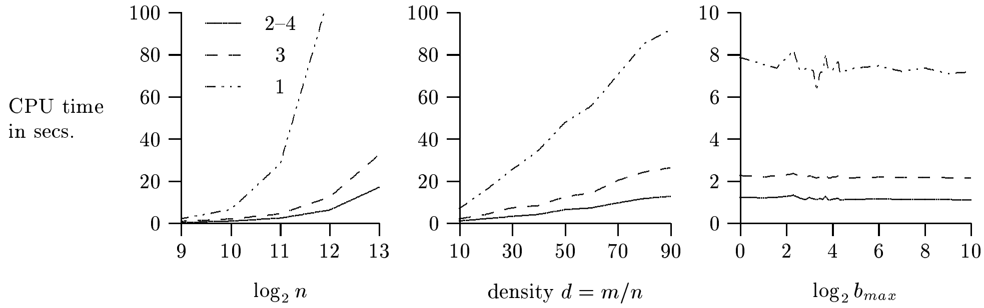

Figure 5: Experimental results for different strategies within the candidate search (strategy 1, strategy 3, and strategy 2 combined with strategy 3 and 4) for a test-suite on graphs with an Euclidean distance function.

to a different tree. Whereas a forest version leads to shorter augmenting paths, a single tree version has the advantage of a reduced overhead.

Note that some of these strategies can also be used in combination. Just to give a rough idea, Figure 5 shows the impact of the different strategies on the run-time for a test-suite of b-matching problems on Euclidean nearest-neighbor graphs (details of the experimental set-up are described in [MS99]).

4.5 Exchange of data structures

Apart from data accessors we also apply the "traditional" concept of *abstract data types* which provide a common interface for a certain functionality but can internally be realized in many different ways.

We have already discussed the strategic question whether one should keep a partition of the edges according to the labels of their endpoints in the current forest or not. Computational results strongly indicated that explicitly maintaining such a edge partition is worth doing. But it is not at all clear which data structure to keep these edge sets is most efficient. Should we use simply a doubly-linked list structure which allows cheap insertion and deletion operations in $\mathcal{O}(1)$ per edge but requires linear time to find the edges with minimum reduced costs in the update step? Or is a priority queue like a d-heap or some Fibonacci heap the better choice because of the $\mathcal{O}(1)$ time to perform the minimum operation at the expense of more expensive insert and delete operations? Note that we usually have to perform much more insert/delete operations than minimum operations. Hence, an answer to these questions can only be given by computational testing.

4.6 Reusability of algorithms

We give one concrete example for a reuse of the blossom framework, namely fractional b-matching. Recall that fractional b-matching is the relaxation of b-matching which drops the integrality constraints on the matching x.

The adaption of our implementation to fractional b-matching becomes extremely easy. The only necessary modification is to exchange the data type of the matching x from `integer` to `double` or to `float`. (It is easy to see that one can keep the fractional matching half-integral throughout the algorithm.) This change suffices because the calculation of the maximal value by which we can augment after some forest growing step now returns a value of $\frac{1}{2}$ in those cases where the integral version would round down to an integer and therefore return a zero which invokes a shrinking step afterwards.

As shrinking is not necessary in a fractional algorithm, it is suitable for reasons of efficiency to start the algorithm with an instance of the graph class `unshrinkable_graph` instead of using the graph class `surface_graph` as the latter requires extra overhead for handling the `outer` information. Moreover, as the `shrinker` is never called in a normal execution of the algorithm it can be replaced by a `dummy_shrinker` which does nothing but returns an exception if it is called because this indicates an implementation bug.

Recall that fractional b-matching can be transformed into a Hitchcock transportation problem and therefore, in principle, be solved by any implementation for minimum cost flow problems, in particular by the network simplex. However, if we want to use the solution of the fractional matching problem as an improved basis for the integral matching algorithm, there is one pitfall. The problem is that if we use an algorithm for fractional matching as a black box, this algorithm certainly does not know that the input of the integral matching algorithm requires additional structural properties of the b-matching as preconditions. As a consequence, it is necessary to implement additional conversion algorithms which transform an integral matching obtained from rounding an optimal fractional b-matching into a starting matching fulfilling the requirements of Pulleyblank's algorithm. (This subtle detail is ignored in Anstee's paper [Ans87].) We put emphasis on this point as the fractional algorithm obtained as an adaption of the integral one gives us the desired structural properties of the b-matching almost for free.

4.7 Initialization

Pulleyblank's algorithm can be decomposed into an initialization phase, the primal-dual loop, and a final expanding phase. As there are many different possibilities for a concrete initialization it is useful to separate these parts strictly form each other.

Principle *Separate each algorithm from its initialization.*

The benefit from an exchange in the initialization phase can be dramatic. The "jump start" with a fractional matching solver is one example which we discussed earlier. Strengthening of the initial dual solution such that for each node at least one adjacent edge lies in the initial equality subgraph also proved to be useful. Similar experiences have been reported by Miller & Pekny [MP95].

For cardinality matching problems, the first author's experiments with several greedy starting heuristics showed that it is often possible to do almost all work in the initialization phase. In fact, our heuristics have been so powerful that the loop kernel of the algorithm often only works as a checker for optimality in non-perfect maximum cardinality problems [MM95].

4.8 Observer pattern

The *observer pattern* defines a dependency between an observer class and an observing class such that whenever the state of the observed object changes, the observing class is notified about the change. We have already discussed one nice application of the observer pattern as a prerequisite of an advanced strategy for the candidate search.

In addition, observers allow us to get additional insights into the course of the algorithm by collecting data on function calls. Profilers such as *gprof* or *quantify*[1], could be used to count the number of function calls as well as to measure the time spent inside the functions. However,

[1]Rational Software Corporation

this gives only the overall sum of calls to a certain function and requires that the data we are interested in can be expressed in the number of function calls.

Beyond mere operation counting observers can deliver much more detailed information. For example, we can determine the maximum nesting level of blossoms. This parameter is no operation and therefore not available to profilers, but is a valuable indicator for the hardness to solve some problem instance. For example, the nesting level is observed to be much lower in randomly generated instances than in structured instances.

Moreover, we may want to know how certain quantities change over time, in particular, we want to sample data from every iteration of the primal-dual loop (here we use also another pattern, the *loop kernel pattern* [KNW97]). For example, we collect a series of data from each dual update to find out which updates are most expensive.

This can even be used to control the selected strategy on-line. It has been observed [CR97] that the final ten augmentations usually require most of the overall computation time. Hence, if we can recognize with the help of an observer that the algorithm slows down it might be advisable to change the strategy. Cook & Rohe propose to switch from a single tree growing strategy to a forest growing strategy.

5 Summary and Discussion

We presented a case study oriented to weighted b-matching with emphasis on design problems. Our approach followed proposals of Weihe and co-workers to apply design patterns like graph iterators and data accessors in order to achieve a flexible design. The examples given in the previous section proved that we successfully realized flexibility with respect to several modifications.

Of course, flexibility has its price. Therefore, we briefly discuss two issues of potential drawbacks, namely ease of use and efficiency. We decided to take C++ as the programming language for our implementation. We heavily used templates (static polymorphism) and several new features of the recently finished ANSI/ISO standard for C++ [ANSI98], in particular the so-called *traits* mechanism (roughly speaking, in our context a traits class encapsulates a set of type definitions or constants which are template parameters of an algorithmic class.) Applying all the desired design patterns requires excellent expertise in advanced programming techniques, at least to a much higher degree than traditional concepts. Hence, ease of use may be a critical issue.

One cannot expect that a flexible framework as discussed in this paper is as efficient as a specialized hand-coded implementation. However, it is hard to estimate by which factor two such implementations may differ from each other. Today, the main disadvantage of templates is that this feature is not fully supported by most compilers. In principle, compilers should be able to handle templated code as well as code without templates, and to optimize away the additional overhead imposed by encapsulation. However, the current compiler technology of *gcc/g++*[2], version 2.8.1, as well as its offshoot *egcs*[3], version 1.1.1, does not seem to achieve this satisfactorily.

Therefore, it is quite remarkable, that the current version of our code is already significantly faster than the code of Miller & Pekny [MP95]. This was definitely not true for the first prototype of our framework. However, through experiments we have been able to identify the bottlenecks of our implementation, and by exchanging subalgorithms and data structures the speed-up was made possible by the flexibility of our framework. And we believe that there is still potential

[2]Free Software Foundation Inc.

[3]Cygnus Solutions

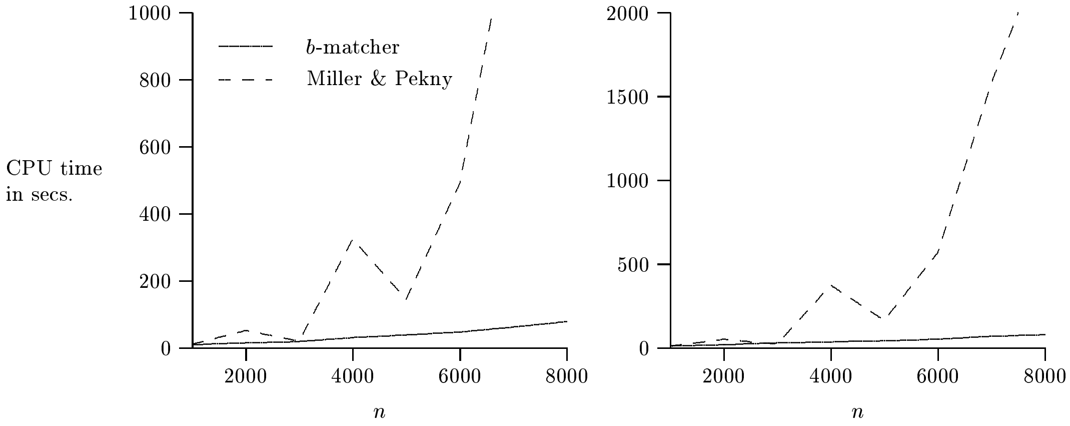

Figure 6: A comparison of the code from Miller & Pekny with our b-matcher on a test-suite of graphs with an Eucledian distance function, a density $d = m/n$ of 20 (left) and 30 (right), and b_v varying between 1 and 10.

for further improvements of efficiency. For an in-depth discussion of our computational results we refer to [MS99].

The fact that our code is already superior to the only b-matching executable available for a comparison (of Miller & Pekny) encourages hopes that the design concepts are suitable for high performance computations. At least, we got an implementation which is mature enough to solve even the hardest instances of the mesh refinement application in less than 13 seconds for a sparse graph with more than 17000 nodes on a SUN UltraSPARC2 with 200 MHz running under Solaris 2.6. The solution for associated b-matching problem to the example shown in Figure 1 took only 3 seconds.

Future work will show whether the flexibility also pays off for further specializations or extensions of the b-matching problem.

Acknowledgment. The authors wish to thank Donald Miller, Joseph Pekny, and his student Paul Bunch for providing us with an executable of their b-matching code.

References

[AC93] D. Applegate and W. Cook, *Solving large-scale matching problems*, Network Flows and Matching, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (D. S. Johnson and C. C. McGeoch, eds.), vol. 12, 1993, pp. 557–576.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows*, Prentice Hall, 1993.

[ANSI98] American National Standards Institute, *Programming Languages — C++*, International Standard ISO/IEC 14882, New York, 1998.

[Ans87] R. P. Anstee, *A polynomial algorithm for b-matching: An alternative approach*, Information Processing Letters **24** (1987), 153–157.

[AO96] R. K. Ahuja and J. B. Orlin, *Use of representative operation counts in computational testing of algorithms*, INFORMS Journal on Computing (1996), 318–330.

[BD83] M. O. Ball and U. Derigs, *An analysis of alternative strategies for implementing matching algorithms*, Networks **13** (1983), 517–549.

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

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

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

[Edm65b] J. Edmonds, *Paths, trees, and flowers*, Can. J. Math. **17** (1965), 449–467.

[Edm67] J. Edmonds, *An introduction to matching*, Lecture notes, University of Michigan, Ann Arbor, 1967.

[EJ70] J. Edmonds and E. L. Johnson, *Matching: A well-solved class of integer linear programs*, Combinatorial Structures and their Applications, Calgary International Conference, Gordon and Breach (1970), 89–92.

[EJL69] J. Edmonds, E. L. Johnson, and S. C. Lockhart, *Blossom I: a computer code for the matching problem*, unpublished report, IBM T. J. Watson Research Center, Yorktown Heights, New York, 1969.

[Gab83] H. N. Gabow, *An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems*, Proceedings of the 15th Annual ACM Symposium on the Theory of Computing (1983), 448–456.

[Ger95] A. M. H. Gerards, *Matching*, (M. O. Ball et al., ed.), Handbooks in Operations Research and Management Science, vol. 7, North-Holland, 1995, pp. 135–224.

[GH85] M. Grötschel and O. Holland, *Solving matching problems with linear programming*, Math. Prog. **33** (1985), 243–259.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995.

[KNW97] D. Kühl, M. Nissen, and K. Weihe, *Efficient, adaptable implementations of graph algorithms*, Workshop on Algorithm Engineering (WAE'97), 1997, http://www.dsi.unive.it/˜wae97/proceedings.

[KW96] D. Kühl and K. Weihe, *Iterators and handles for nodes and edges in graphs*, Konstanzer Schriften in Mathematik und Informatik Nr. 15, Universität Konstanz, 1996, http://www.informatik.uni-konstanz.de/˜weihe/manuscripts.html#paper24.

[KW97] D. Kühl and K. Weihe, *Data access templates*, C++–Report **9** (1997), no. 7, pp. 15 and 18–21.

[LP86] L. Lovász and M. D. Plummer, *Matching theory*, Annals of Discrete Mathematics, vol. 29, North-Holland, 1986.

[Mar79] A. B. Marsh III, *Matching algorithms*, Ph.D. thesis, The John Hopkins University, Baltimore, 1979.

[MH97] M. Müller-Hannemann, *High quality quadrilateral surface meshing without template restrictions: A new approach based on network flow techniques*, Proceedings of the 6th International Meshing Roundtable, Park City, Utah, Sandia National Laboratories, Albuquerque, USA, 1997, pp. 293–307, also to appear in the International Journal of Computational Geometry and Applications (IJCGA).

[Mil95] D. L. Miller, *A matching based exact algorithm for capacitated vehicle routing problems*, ORSA J. of Computing (1995), 1–9.

[MM95] R. H. Möhring and M. Müller-Hannemann, *Cardinality matching: Heuristic search for augmenting paths*, Technical report No. 439/1995, Fachbereich Mathematik, Technische Universität Berlin, 1995, ftp://ftp.math.tu-berlin.de/pub/Preprints/combi/Report-439-1995.ps.Z.

[MM97] R. H. Möhring and M. Müller-Hannemann, *Complexity and modeling aspects of mesh refinement into quadrilaterals*, Proceedings of the 8th Annual International Symposium on Algorithms and Computation, ISAAC'97, Singapore, Lecture Notes in Computer Science **1350**, Springer-Verlag, 1997, pp. 263–273.

[MMW97] R. H. Möhring, M. Müller-Hannemann, and K. Weihe, *Mesh refinement via bidirected flows: Modeling, complexity, and computational results*, Journal of the ACM **44** (1997), 395–426.

[MP95] D. L. Miller and J. F. Pekny, *A staged primal-dual algorithm for perfect b-matching with edge capacities*, ORSA J. of Computing **7** (1995), 298–320.

[MS99] M. Müller-Hannemann and A. Schwartz, *Implementing weighted b-matching algorithms: Insights from a computational study*, Proceedings of the Workshop on Algorithm Engineering and Experimentation (ALENEX'99), Lecture Notes in Computer Science, Springer, to appear, 1999.

[Orl88] J. B. Orlin, *A faster strongly polynomial minimum cost flow algorithm*, Proceedings of the 20th Annual ACM Symposium on Theory of Computing (1988), 377–387.

[PR82] M. Padberg and M. R. Rao, *Odd minimum cut-sets and b-matchings*, Math. Oper. Res. **7** (1982), 67–80.

[Pul73] W. R. Pulleyblank, *Faces of matching polyhedra*, Ph.D. thesis, Faculty of Mathematics, University of Waterloo, 1973.

[Pul95] W. R. Pulleyblank, *Matchings and extensions*, (R. L. Graham, M. Grötschel, and L. Lovász, eds.), Handbook of Combinatorics, vol. 1, North-Holland, 1995, pp. 179–232.

[Wei98] K. Weihe, *A software engineering perspective on algorithms*, Konstanzer Schriften in Mathematik und Informatik Nr. 50, Universität Konstanz, ftp://ftp.informatik.uni-konstanz.de/pub/preprints/1998/preprint-050.ps.Z, 1998.

Appendix

A The Blossom Algorithm for b–Matching

For simplicity, we present the blossom algorithm only for the case of maximum weight perfect b-matching. Recall the definitions from Section 2.

A.1 Basic concepts and terminology

We use the convention that the minimum over an empty set is $+\infty$. Let $G = (V, E)$ be a simple undirected graph with node set V, edge set E, and $b \in \mathbb{N}^{|V|}$ a vector of *node capacities*. A vector $x \in \mathbb{N}_0^{|E|}$ is called a *b-matching* if $x(\delta(v)) \leq b_v$ for $v \in V$. We call a b-matching x a *perfect b-matching* if $x(\delta(v)) = b_v$ for $v \in V$. A b-matching x is called *near-perfect b-matching* if there is a unique node $t \in V$ with

$$\begin{aligned} x(\delta(v)) &= b_v \quad \text{for } v \in V \setminus \{t\} \quad \text{and} \\ x(\delta(t)) &= b_t - 1. \end{aligned}$$

An edge is *matched* if $x_e \geq 1$. Each node $v \in V$ with $x(\delta(v)) < b_v$ is called a *deficit node.* We define the *node deficit* $\Delta_{G,x}(v) := b_v - x(\delta_G(v))$ or simply $\Delta(v) := b_v - x(\delta(v))$.

A second linear programming description. An alternative to the blossom description which we used for the outline in Section 2 is the following *odd cut description*:

$$\begin{array}{llcll} & \text{maximize } c^T x & & & \\ & \text{subject to} & & & \\ \text{(P1)} & x(\delta(v)) & = & b_v & \text{for } v \in V \\ \text{(P2)} & x_e & \geq & 0 & \text{for } e \in E \\ \text{(P3-O)} & x(\delta(S)) & \geq & 1 & \text{for } S \in \Omega. \end{array}$$

Its dual linear program can be written as

$$\begin{array}{llcll} & \text{minimize } \sum_{v \in V} b_v y_v - \sum_{S \in \Omega} Y_S & & & \\ & \text{subject to} & & & \\ \text{(D1-O)} & y_u + y_v - Y(\Omega_\delta(e)) & \geq & c_e & \text{for } e = (u, v) \in E \\ \text{(D2)} & Y_S & \geq & 0 & \text{for } S \in \Omega, \end{array}$$

where $\Omega_\delta(e) := \{ S \in \Omega \mid e \in \delta(S) \}$. The *reduced costs* are defined as

$$\bar{c}_e := y_u + y_v - Y(\Omega_\delta(e)) - c_e \text{ for all } e \in E.$$

The corresponding complementary slackness conditions have the following form:

$$\begin{array}{llcll} \text{(CS1)} & x_e > 0 & \Longrightarrow & \bar{c}_e = 0 & \text{for } e \in E \\ \text{(CS2-O)} & Y_S > 0 & \Longrightarrow & x(\delta(S)) = 1 & \text{for } S \in \Omega. \end{array}$$

Augmenting paths, trees, and forests. A *path* $\pi = (v_0, e_0, v_1, \ldots, v_k)$ is an alternating, ordered sequence of nodes and edges with $e_i = \{v_i, v_{i+1}\}$, $i = 0, \ldots, k-1$. We do not require that a path is simple. The length of a path π is the number of edges in π, denoted by $length(\pi) := |E(\pi)|$. Let $head(\pi)$ and $tail(\pi)$ denote the first and last node of π. Nodes and edges are *even* or *odd* according to the parity of the index in the given sequence. A path π in a graph G is called *alternating* with respect to a b-matching x if each even edge is matched (the other edges may be matched or un-matched). Denote by k_e the number of times that e appears as an even minus the number that e appears as an odd edge on π. (For a simple path, $k_e = 1$ if e is even, and $k_e = -1$, otherwise.)

An alternating path π of odd length is called *σ-augmenting path* or just *augmenting path* with respect to a b-matching x if $head(\pi)$ and $tail(\pi)$ are deficit nodes and $x_e \geq k_e \cdot \sigma$ for each edge $e \in \pi$. Replacing the b-matching x by a new matching x' defined as

$$x'_e := \begin{cases} x_e - k_e \cdot \sigma & \text{if } e \in E(\pi) \\ x_e & \text{otherwise} \end{cases}$$

is called a *σ-augmentation*.

Let T be a tree contained in G rooted at node r. Denote by $\pi(v)$ the unique simple path in T from the the root r to $v \in V(T)$. We call v an *even* node of T if $\pi(v)$ has even length, and *odd* otherwise. In particular, the root node is an even node of T. An edge $e = (v, u) \in E(T)$ is *even* if it is the last edge of an even length simple path $\pi(u)$ or $\pi(v)$. A tree T with root node r is called *augmenting tree* with respect to x if

(T1) $\Delta(r) > 0$,
(T2) $\Delta(v) = 0$ for all nodes $v \in V(T)$ with $v \neq r$,
(T3) each *even* edge in $E(T)$ is matched, and
(T4) each matched edge adjacent to a node in T belongs to $E(T)$.

A node-disjoint union of augmenting trees is called *augmenting forest*. A node v in a forest F is *even (odd)* if v is even (odd) with respect to the tree in which v is contained. We denote the set of even (odd) nodes in F by $even(F)$ ($odd(F)$). An augmenting forest F with respect to a b-matching x is called *Hungarian* if no node $u \in even(F)$ is adjacent to a node $v \in odd(F)$.

Blossoms and shrinking. Let B be an induced subgraph of G and $t \in V(B)$. B is called a *blossom* with respect to a b-matching x in G if

(B1) B is connected,
(B2) B contains no even circuit,
(B3) B contains exactly one odd circuit C,
(B4) $x_{|_B}$ is a near-perfect b-matching in B with deficit node t,
(B5) $x_e \geq 1$ for all $e \in E(B) \setminus E(C)$, and
(B6) for each node $v \in V(B) \setminus \{t\}$ there is an even length path π_v from t to v with $x_e \geq 1$ for all even edges in π_v.

We call the node $t \in V(B)$ the *tip node* of B. A node $v \in V(B)$ is called *terminal* if $|\delta_B(v)| = 1$. Removing the edges of the circuit C disconnects B into $|C|$ trees, the so-called *petals*. Note that a petal can be a single node.

A node set S is called *shrinkable* if the subgraph $(S, \gamma(S))$ of G forms a blossom with respect to a b-matching x. Observe that $b(V(B))$ is odd, if B is a blossom.

By *shrinking* a node set $S \subset V(G)$ in the graph G into a *pseudo-node* s we obtain a new graph $G' := G \times S$ in the following manner:

$$\begin{aligned} G \times S &:= (V', E') \\ V' &:= V \setminus S \cup \{s\} \\ E' &:= \gamma(V \setminus S) \cup \{ (u,s) \mid (u,v) \in \delta(S), v \in S \} \\ c'_e &:= c_e \qquad \text{for } e \in \bar{E} \\ b'_v &:= \begin{cases} 1 & v = s \\ b_v & v \in V. \end{cases} \end{aligned}$$

The reverse operation is called *expanding a pseudo-node.* The *matching obtained from shrinking* S in G into a pseudo-node s is defined as a vector $x \times S \in \mathbb{N}_0^{|E(G \times S)|}$ with

$$(x \times S)_e := \begin{cases} x_e & \text{if } e \in E(G) \cap E(G \times S) \\ x_{(u,v)} & \text{if } e = (u,s) \text{ and there is } v \in V(G) \text{ with } (u,v) \in E(G). \end{cases}$$

The blossom algorithm performs iteratively several shrinking and expanding steps which all replace the current graph G_{k-1} with a new graph G_k. The obtained graph G_k can be represented as

$$G_k = G \times \mathcal{R} := G \times \mathcal{S} := (G \times S_1) \times S_2) \ldots \times S_\ell$$

with the family of shrinkable sets $\mathcal{S} = \{S_1, \ldots, S_\ell\}$ and the set of pseudo-nodes $\mathcal{R} = \{s_1, \ldots, s_\ell\}$. Nodes contained in the node set V of the original graph are called *real nodes.* We distinguish between shrunken nodes, called *interior nodes*, and non-shrunken nodes, called *exterior nodes.* For a node $v \in V(G)$, we use the notation $outer_k(v)$ (or simply $outer(v)$ for the current graph) to denote the exterior node in G_k which contains v as an interior node or is identical to v.

A.2 Maximum cardinality b–matching

Let us first consider the case of unweighted, not necessarily perfect b-matchings in a graph G.

Surface graph. The framework for the b-matching algorithm has to shrink and to expand blossoms. Therefore it works on a current graph, the so-called *surface graph* $\widetilde{G}$ which can be represented as

$$\widetilde{G} = G \times \mathcal{R} = ((G \times S_1) \times S_2) \ldots \times S_\ell$$

where $\mathcal{S} = \{S_1, \ldots, S_\ell\}$ is a family of shrinkable sets and $\mathcal{R} = \{s_1, \ldots, s_\ell\}$ the corresponding set of pseudo-nodes. Initially we set $\widetilde{G} := G$.

Matching subgraph. The *matching subgraph* is defined as

$$\begin{aligned} G^+(x) &:= (V(\widetilde{G}), E^+(x)) \\ E^+(x) &:= \{ e \in E(\widetilde{G}) \mid x_e > 0 \}. \end{aligned}$$

Throughout the algorithm, each component $M \subseteq G^+(x)$ satisfies the following invariants:

- (M1) M contains no even polygon,
- (M2) M contains at most one odd polygon,
- (M3) there is at most one deficit node in M, and
- (M4) if M contains an odd polygon then M contains no deficit nodes.

We maintain an augmenting forest F which holds the invariant

(F1) For each $v \in V(\widetilde{G})$ with $\Delta_{\widetilde{G},x}(v) > 0$ there is a tree $T \in F$ with $v = root(T)$.

Trivial Initialization. One can start with an empty matching and $\mathcal{R} = \emptyset$. Each node $v \in V$ with $b_v \neq 0$ becomes the root of a trivial augmenting tree. Thus (F1), (M1) – (M4), and (T1) – (T4) are satisfied.

Algorithm outline. The basic idea of the blossom algorithm is to grow an augmenting forest F until we determine that the forest is Hungarian or find an augmenting path. In the latter case we augment along this path, and continue afterwards with forest growing steps. There is one pitfall. If the forest is Hungarian, but contains odd pseudo-nodes, we may miss augmenting paths. Therefore, when F becomes Hungarian, we expand all odd pseudo-nodes, adjust the forest, and try again to grow F, until F is Hungarian and does not contain odd pseudo-nodes.

Search for an Hungarian forest. Let us define the set of *candidate edges* as

$$\textsc{Candidates}(F, \widetilde{G}) := \{\ (u, v) \in E(\widetilde{G} \mid u \in even(F)\ \wedge\ v \notin odd(F)\ \}.$$

If this set is non-empty, the forest F is not Hungarian. Hence we examine if there is a *candidate edge* $e \in \textsc{Candidates}(F, \widetilde{G})$. If we have a candidate $e = (u, v)$, we modify the augmenting forest in the following way: If $v \notin V(F)$ then v belongs to a matched component $M \in G^+(x)$ which can be attached to the tree containing u, by calling subalgorithm $\textsc{Grow\_Forest}$. If $v \in even(F)$ we have either u and v in different trees or in the same one. In the former case we are able to augment with $\textsc{Augment\_Two\_Trees}$, whereas in the latter case, we try to augment with $\textsc{Augment\_One\_Tree}$, but may be forced to shrink a blossom to maintain (M1) - (M4).

Grow forest. $\textsc{Grow\_Forest}$ is invoked with a candidate edge $e = (u, v)$ where $u \in V(F)$ and $v \notin V(F)$. Let $M \in G^+(x)$ be the matched component which contains v (and exists as v has no deficit). If M itself is a tree, then it can be attached to the tree $T_u \in F$ containing u in the obvious way, and the nodes of M become labeled even or odd according to the parity of the path to the root of T_u.

If M contains a circuit, attaching a component $M \in G^+(x)$ to the forest F involves a complication. Assume M contains a circuit C. Then C has odd length, because of (M1) and (M2). We can attach to the tree T_u one after another all edges of M with the exception of one edge $f \in E(C)$ which would close a circuit in T_u. The edge f is matched, and so violates (T1). Calling $\textsc{Augment\_One\_Tree}$ with edge f reensures invariant (T1).

Augmenting (two trees). Suppose that the selected candidate edge $e = (v_1, v_2)$ has its endpoints in two different trees $T_1, T_2 \in F$. $\textsc{Augment\_Two\_Trees}$ determines the the augmenting path π built by the unique path $\pi_1 \in T_1$ from the root of T_1 to v_1, the edge $v_1 v_2$ and unique path from v_2 to the root of T_2. It also calculates the maximal change value σ on π. By (T1) and (T3), $\sigma \geq 1$ which implies that augmenting on π is possible. It is clear from the choice of σ that there is at least one violation of (T1), (T3) or (T4) after the augmentation. Therefore, we remove trees without deficit from the forest to keep (T1). If an even edge becomes unmatched, we cut off the corresponding subtree above such an edge. Finally, if either v_1 or v_2 is still in the forest (note that only one can be in F), we call $\textsc{Grow\_Forest}$ with edge e. After this modification (T1), (T3) and (T4) will be satisfied again. As we do not create a matched polygon, (M1) – (M4) remain fulfilled.

Augmenting (one tree). Suppose that the selected candidate edge $e = (v_1, v_2)$ has both endpoints in the same tree $T \in F$. Let π_{v_1}, π_{v_2} the path from the root r of T to v_1 or v_2, respectively and π_S the common portion of π_{v_1} and π_{v_2}. We determine first a value σ as follows:

$$\begin{aligned}
\sigma_S &:= \min\{\, x_e \mid e \text{ is an even edge of } \pi_S \,\},\\
\sigma_i &:= \min\{\, x_e \mid e \text{ is an even edge of } \pi_i \,\} \quad \text{for } i = 1, 2,\\
\sigma &:= \min\{\lfloor \tfrac{1}{2}\sigma_S \rfloor, \sigma_1, \sigma_2, \lfloor \tfrac{1}{2}\Delta(r) \rfloor\}.
\end{aligned}$$

If $\sigma = 0$ there is no augmenting path and the circuit in $T \cup e$ belongs to a blossom which will be shrunken. Otherwise AUGMENT_ONE_TREE augments on the σ-augmenting path π_{v_1}, the edge $v_1 v_2$ and the reverse path π_{v_1}.

It is clear from the choice of σ that there is at least one violation of (T1), (T3) or (T4) after the augmentation. Therefore, we repair the forest in a similar way as in AUGMENT_TWO_TREES such that (T1), (T3) and (T4) will be satisfied again.

Expansion of a blossom. Suppose that s is the pseudo-node which corresponds to the blossom B which we want to expand. There are several cases.

Case 1: $s \notin V(F)$.
By (F1), this implies that s has no deficit. As $b_s = 1$, there must exist an edge $e = (u, v) \in E$ with $u \in V(B)$ and $v \notin V(B)$ for which the corresponding edge $(outer(u), v) = (s, v) \in E(\widetilde{G})$ is matched. After expanding the blossom, its tip node t has deficit one if $t \neq u$. By (B6), however, we can correct the matching within the blossom on an even length alternating path from the tip node of B to u, such that each node of B is perfectly matched. The forest F remains unchanged.

Case 2: $s \in V(F)$ and s is root of some tree $T \in F$.
As s is root of some tree, it has a deficit of 1. Therefore, s is not incident to any matched edge by (T4). We remove T from F, and expand the blossom. We choose the tip node t of the blossom as the root of a new tree T' and call GROW_FOREST for all adjacent edges to t which are matched.

Case 3: $s \in V(F)$, belongs to a tree $T \in F$, but is not its root.
Because of (F1), s has no deficit. Hence, there is a matched edge e adjacent to s, and we can apply the same matching correction after expansion of B as in Case 1. Let $f \in E(T)$ be the edge incident to s on the unique simple path to the root of T. We cut off from T the whole subtree above f. If f is matched, we call GROW_FOREST with f to reensure (T4).

We note that one should try for reasons of efficiency to reuse subtrees which are cut off first, but have to be reattached afterwards to keep invariant (T4). However, we skip the technical details.

A.3 The primal-dual blossom algorithm

We maintain

- a dual feasible solution y with satisfies (D1), (D2), and
- a primal solution x with satisfies (P2), (P3) and

$$\text{(P1')} \qquad x(\delta(v)) \;\leq\; b_v \qquad \text{for } v \in V.$$

The *equality subgraph* $G^=$ with respect to the original graph G and reduced costs $\bar{c}$ is defined as follows.

$$\begin{aligned}
G^= &:= (V, E^=)\\
E^= &:= \{\, e \in E \mid \bar{c}_e = 0 \,\}.
\end{aligned}$$

Initialization (trivial version). A feasible dual solution y is given with $y_v := \frac{1}{2}\max\{ c_e \mid e = (v,w) \in E \}$ for all $v \in V$, and $Y \equiv 0$. All other values are initialized as in the cardinality case. Thus (F1), (T1) – (T4), (M1) – (M4), (D1), (D2), (P1'), (P2), (P3), (CS1) and (CS2) are satisfied.

Primal-dual loop. The algorithm consists of a loop which alternates between primal and dual steps. In the primal part, we reuse the maximum cardinality b-matching algorithm SEARCH_HUNGARIAN_FOREST from the previous subsection with the equality subgraph $G^=$. This algorithm terminates with an Hungarian forest F.

If F is empty, the primal-dual algorithm also terminates after expanding all remaining pseudo-nodes with a perfect b-matching x. Otherwise Dual_Update tries to change the dual variables and if it succeeds, we first expand all odd pseudo-nodes for which the associated dual variables became zero, and continue with SEARCH_HUNGARIAN_FOREST. If Dual_Update fails to change the dual variables, the algorithm terminates. In this case, the problem is infeasible, that means, there is no perfect b-matching.

Dual update. DUAL_UPDATE tries to change the dual solution such that we can keep all matched edges in the current surface graph $\widetilde{G}$, but get new candidate edges or reduce the potential of some odd pseudo-node to zero such that it is possible to expand it. Using the abbreviations

$$\begin{aligned} EN(F,\widetilde{G}) &:= \{ uv \in E(\widetilde{G}) \mid u \in even(F) \wedge v \notin V(F) \}, \\ EE(F,\widetilde{G}) &:= \{ uv \in E(\widetilde{G}) \mid u,v \in even(F) \}, \\ OP(F,\mathcal{R}) &:= \{ s \in \mathcal{R} \mid s \in odd(F) \} \end{aligned}$$

we calculate first $\varepsilon_1, \varepsilon_2, \varepsilon_3$ according to

$$\begin{aligned} \varepsilon_1 &:= \min\{ \bar{c}_e \mid e \in EN(F,\widetilde{G}) \}, \\ \varepsilon_2 &:= \min\{ \bar{c}_e \mid e \in EE(F,\widetilde{G}) \}, \\ \varepsilon_3 &:= \min\{ Y_s \mid s \in OP(F,\mathcal{R}) \}. \end{aligned}$$

If we use the blossom description, we determine ε as

$$\varepsilon := \min\{\varepsilon_1, \frac{\varepsilon_2}{2}, \frac{\varepsilon_3}{2}\},$$

whereas for the odd cut description we define

$$\varepsilon := \min\{\varepsilon_1, \frac{\varepsilon_2}{2}, \varepsilon_3\}.$$

If $\varepsilon = \infty$ the forest F is Hungarian and the primal-dual loops stops. Otherwise we perform the following dual update. Let us start with the blossom description case. For all real nodes $v \in V$ we set

$$y_v := \begin{cases} y_v - \varepsilon & \text{if } outer(v) \in even(F) \\ y_v + \varepsilon & \text{if } outer(v) \in odd(F). \end{cases}$$

For all pseudo-nodes $s \in \mathcal{R}$ we set

$$Y_s := \begin{cases} Y_s + 2\varepsilon & \text{if } s \in even(F) \\ Y_s - 2\varepsilon & \text{if } s \in odd(F). \end{cases}$$

In the odd cut description case, the dual update looks as follows: For all exterior real nodes of the surface graph $v \in V(\widetilde{G})$ we set

$$y_v := \begin{cases} y_v - \varepsilon & \text{if } v \in \mathit{even}(F) \\ y_v + \varepsilon & \text{if } v \in \mathit{odd}(F). \end{cases}$$

For all exterior pseudo-nodes $s \in \mathcal{R}$ we set

$$Y_s := \begin{cases} Y_s + \varepsilon & \text{if } s \in \mathit{even}(F) \\ Y_s - \varepsilon & \text{if } s \in \mathit{odd}(F). \end{cases}$$

Reports from the group

"Algorithmic Discrete Mathematics"

of the Department of Mathematics, TU Berlin

605/1998 *Friedrich Eisenbrand:* A Note on the Membership Problem for the Elementary Closure of a Polyhedron

596/1998 *Rolf H. Möhring and Frederik Stork and Marc Uetz:* Resource Constrained Project Scheduling with Time Windows: A Branching Scheme Based on Dynamic Release Dates

595/1998 *Rolf H. Möhring and Andreas S. Schulz and Marc Uetz:* Approximation in Stochastic Scheduling: The Power of LP-based Priority Rules

591/1998 *Matthias Müller–Hannemann and Alexander Schwartz:* Implementing Weighted b-Matching Algorithms: Towards a Flexible Software Design

590/1998 *Stefan Felsner and Jens Gustedt and Michel Morvan:* Interval Reductions and Extensions of Orders: Bijections to Chains in Lattices

577/1998 *Martin Skutella:* Semidefinite Relaxations for Parallel Machine Scheduling

566/1997 *Jens Gustedt:* Minimum Spanning Trees for Minor-Closed Graph Classes in Parallel

565/1997 *Andreas S. Schulz, David B. Shmoys, and David P. Williamson:* Approximation Algorithms

564/1997 *Uta Wille:* On Extending Closure Systems to Matroids

559/1997 *Matthias Müller–Hannemann and Karsten Weihe:* Improved Approximations for Minimum Cardinality Quadrangulations of Finite Element Meshes

554/1997 *Rolf H. Möhring and Matthias Müller–Hannemann:* Complexity and Modeling Aspects of Mesh Refinement into Quadrilaterals

551/1997 *Hans Bodlaender, Jens Gustedt and Jan Arne Telle:* Linear-Time Register Allocation for a Fixed Number of Registers and no Stack Variables

550/1997 *Karell Bertet, Jens Gustedt and Michel Morvan:* Weak-Order Extensions of an Order

549/1997 *Andreas S. Schulz and Martin Skutella:* Random-Based Scheduling: New Approximations and LP Lower Bounds

542/1996 *Stephan Hartmann:* On the NP–Completeness of Channel and Switchbox Routing Problems

536/1996 *Cynthia A. Phillips, Andreas S. Schulz, David B. Shmoys, Cliff Stein, and Joel Wein:* Improved Bounds on Relaxations of a Parallel Machine Scheduling Problem

535/1996 *Rainer Schrader, Andreas S. Schulz, and Georg Wambach:* Base Polytopes of Series-Parallel Posets: Linear Description and Optimization

533/1996 *Andreas S. Schulz and Martin Skutella:* Scheduling–LPs Bear Probabilities: Randomized Approximations for Min–Sum Criteria

530/1996 *Ulrich H. Kortenkamp, Jürgen Richter-Gebert, Aravamuthan Sarangarajan, and Günter M. Ziegler:* Extremal Properties of 0/1-Polytopes

524/1996 *Elias Dahlhaus, Jens Gustedt and Ross McConnell:* Efficient and Practical Modular Decomposition

523/1996 *Jens Gustedt and Christophe Fiorio:* Memory Management for Union-Find Algorithms

520/1996 *Rolf H. Möhring, Matthias Müller-Hannemann, and Karsten Weihe:* Mesh Refinement via Bidirected Flows: Modeling, Complexity, and Computational Results

519/1996 *Matthias Müller-Hannemann and Karsten Weihe:* Minimum Strictly Convex Quadrangulations of Convex Polygons

517/1996 *Rolf H. Möhring, Markus W. Schäffter, and Andreas S. Schulz:* Scheduling Jobs with Communication Delays: Using Infeasible Solutions for Approximation

516/1996 *Leslie A. Hall, Andreas S. Schulz, David B. Shmoys, and Joel Wein:* Scheduling to Minimize Average Completion Time: Off-line and On-line Approximation Algorithms

515/1996 *Christophe Fiorio and Jens Gustedt:* Volume Segmentation of 3-dimensional Images

514/1996 *Martin Skutella:* Approximation Algorithms for the Discrete Time-Cost Tradeoff Problem

509/1996 *Soumen Chakrabarti, Cynthia A. Phillips, Andreas S. Schulz, David B. Shmoys, Cliff Stein, and Joel Wein:* Improved Scheduling Algorithms for Minsum Criteria

508/1996 *Rudolf Müller and Andreas S. Schulz:* Transitive Packing

506/1996 *Rolf H. Möhring and Markus W. Schäffter:* A Simple Approximation Algorithm for Scheduling Forests with Unit Processing Times and Zero-One Communication Delays

505/1996 *Rolf H. Möhring and Dorothea Wagner:* Combinatorial Topics in VLSI Design: An Annotated Bibliography

504/1996 *Uta Wille:* The Role of Synthetic Geometry in Representational Measurement Theory

502/1996 *Nina Amenta and Günter M.Ziegler:* Deformed Products and Maximal Shadows of Polytopes

500/1996 *Stephan Hartmann and Markus W. Schäffter and Andreas S. Schulz:* Switchbox Routing in VLSI Design: Closing the Complexity Gap

498/1996 *Ewa Malesinska, Alessandro Panconesi:* On the Hardness of Allocating Frequencies for Hybrid Networks

496/1996 *Jörg Rambau:* Triangulations of Cyclic Polytopes and higher Bruhat Orders