To appear in Mathematical Programming Computation
The original publication with publisher's formatting is available at `http://www.springer.com`

Blossom V: A new implementation of a minimum cost perfect matching algorithm

Vladimir Kolmogorov
University College London
V.Kolmogorov@cs.ucl.ac.uk

Abstract

We describe a new implementation of the Edmonds's algorithm for computing a perfect matching of minimum cost, to which we refer as *Blossom V*. A key feature of our implementation is a combination of two ideas that were shown to be effective for this problem: the "variable dual updates" approach of Cook and Rohe [8] and the use of priority queues. We achieve this by maintaining an auxiliary graph whose nodes correspond to alternating trees in the Edmonds's algorithm. While our use of priority queues does not improve the worst-case complexity, it appears to lead to an efficient technique. In the majority of our tests Blossom V outperformed previous implementations of Cook and Rohe [8] and of Mehlhorn and Schäfer [24], sometimes by an order of magnitude. We also show that for large VLSI instances it is beneficial to update duals by solving a linear program, contrary to a conjecture by Cook and Rohe.

1 Introduction

We consider the problem of computing a perfect matching of minimum cost in an undirected weighted graph. (A *perfect matching* is a subset of edges such that each node in the graph is met by exactly one edge in the subset.)

In 1965, Edmonds [11, 12] invented the famous *blossom* algorithm that solves this problem in polynomial time. A straightforward implementation of Edmonds's algorithm requires $O(n^2m)$ time, where n is the number of nodes in the graph and m is the number of edges. Since then, the worst-case complexity of the blossom algorithm has been steadily improving. Both Lawler [22] and Gabow [16] achieved a running time of $O(n^3)$, Galil, Micali and Gabow [28] improved it to $O(nm \log n)$, which was further improved to $O(n(m \log\log\log_{\max\{m/n,2\}} n + n \log n))$ by Gabow, Galil and Spencer [18]. The current best known result in terms of n and m is $O(n(m + \log n))$ due to Gabow [17]. Somewhat better asymptotic running times are known for integral edge weights (see e.g. [8] for a survey).

There is also a long history of computer implementations of the blossom algorithm, starting with the *Blossom I* code of Edmonds, Johnson and Lockhart [13]. For several years, the *Blossom IV* code of Cook and Rohe [8] was considered as the fastest available implementation of the blossom algorithm (see [5, 8] for comparisons with earlier codes). The main feature of Blossom IV was a particular strategy for updating dual variables called the *variable dual updates* (or the *variable* δ) approach [8].

Blossom IV and earlier codes used rather simple data structures; in particular, they did not exploit priority queues for finding an edge with the smallest slack. A natural question is whether priority queues can improve a practical implementation, given that they are heavily used for improving the worst-case complexity. An affirmative answer was given by Mehlhorn and Schäfer [24] who developed an implementation that runs in $O(nm \log n)$ and showed an improvement over the Blossom IV code. The implementation of [24] follows the algorithm of Galil, Micali and Gabow [28], and uses *concatenable priority queues*. This algorithm uses a fixed δ approach, which is essential for achieving the $O(nm \log n)$ bound.

In this paper we describe a new implementation of Edmonds's algorithm to which we refer as *Blossom V*. Our motivation was to incorporate both the variable δ approach and the use of priority queues, which were showed to be effective in practice in [8] and in [24], respectively. A key feature of our implementation is the maintenance of an auxiliary graph whose nodes correspond to alternating trees in the Edmonds's algorithm. The edges of this graph store pointers to the corresponding priority queues. Our use of priority queues does not improve over the worst-case complexity of Blossom IV (which we believe to be $O(n^3m)$), but it appears to be quite effective in practice.

The rest of the paper is organized as follows. In section 2 we give an overview of the blossom algorithm and the variable δ approach. In section 3 we describe details of our implementation. Computational results are presented in section 4. Finally, section 5 presents conclusions and discusses future work.

2 Background: Edmonds's blossom algorithm

Let $G = (V, E, c)$ be an undirected weighted graph. A *matching* is a subset of edges $E' \subseteq E$ such that each node in V has at most one incident edge in E'. Matching E' is *perfect* if each node in V has exactly one incident edge in E'. The goal is it compute a perfect matching E' of minimum cost $c(E')$. (As usual, if A is a subset of B and z is a vector in $B^{\mathbb{R}}$, then $z(A)$ denotes $\sum_{i \in A} z_i$.) We assume that a perfect matching in G exists. Matchings $E' \subseteq E$ will be represented by incidence vectors $x \in \{0,1\}^E$.

LP formulation The blossom algorithm is based on the following linear programming formulation of the minimum cost perfect matching problem. For a subset $S \subseteq V$, let $\delta(S)$ be the set of boundary edges of S, i.e. $\delta(S) = \{(u,v) \in E \mid u \in S, v \in V - S\}$. For a single node $v \in V$ we denote $\delta(v) = \delta(\{v\})$. Let $\mathcal{O}$ be the set of all subsets of V of odd cardinality containing at least three nodes. Edmond's LP is then given by

PRIMAL

$$\begin{aligned} \min \quad & \textstyle\sum_{e \in E} c_e x_e & & \text{(1a)} \\ \text{subject to} \quad & x(\delta(v)) = 1 & \forall\, v \in V \quad & \text{(1b)} \\ & x(\delta(S)) \geq 1 & \forall\, S \in \mathcal{O} \quad & \text{(1c)} \\ & x_e \geq 0 & \forall\, e \in E \quad & \text{(1d)} \end{aligned}$$

DUAL

$$\begin{aligned} \max \quad & \textstyle\sum_{v \in V} y_v + \sum_{S \in \mathcal{O}} y_S & & \text{(2a)} \\ \text{subject to} \quad & \texttt{slack}(e) \geq 0 & \forall\, e \in E \quad & \text{(2b)} \\ & y_S \geq 0 & \forall\, S \in \mathcal{O} \quad & \text{(2c)} \end{aligned}$$

where $\texttt{slack}(e)$ in eq. (2b) denotes the reduced cost of edge $e = (u, v)$:

$$\texttt{slack}(e) = c_e - y_u - y_v - \sum_{S \in \mathcal{O}\,:\, e \in \delta(S)} y_S$$

Edges e with zero slack are called *tight*.

Edmond's algorithm maintains a feasible dual vector y and a (non-feasible) integer-valued primal vector x which corresponds to a matching. These vectors are updated so that the cardinality of matching x increases gradually until x becomes a perfect matching. At this point the complementary slackness conditions are satisfied:

$$\begin{aligned} \texttt{slack}(e) > 0 \quad &\Rightarrow \quad x_e = 0 & \text{(3a)} \\ y_S > 0 \quad &\Rightarrow \quad x(\delta(S)) = 1 & \text{(3b)} \end{aligned}$$

and thus x gives a perfect matching of minimum cost.

One potential concern is that the dual problem has an exponential number of variables y_S, $S \in \mathcal{O}$. However, this does not cause problems since at any moment there are at most $O(n)$ subsets $S \in \mathcal{O}$ with non-zero variable y_S. These subsets are called *blossoms*.

A blossom can be defined recursively as follows: it is a cycle containing an *odd* number of "pseudonodes", where a pseudonode is either a node in V or another blossom. Two consecutive pseudonodes in the cycle are connected via an edge called a *blossom-forming edge*. Examples of blossoms are shown in Fig. 1(a). If a pseudonode is contained in some blossom it is called *interior*, otherwise it is *exterior*.

The algorithm treats blossoms as ordinary nodes. This is possible due to the following key property: if we managed to match an edge coming out of one of the nodes inside the blossom, then we can recursively match the remaining even number of nodes to each other using only blossom-forming edges (see Fig. 1(b)). Blossom-forming edges are always tight, which ensures property (3a). Note, changing the dual variable y_v for blossom v (or changing other dual variables outside the blossom) does not affect the slacks of edges inside the blossom.

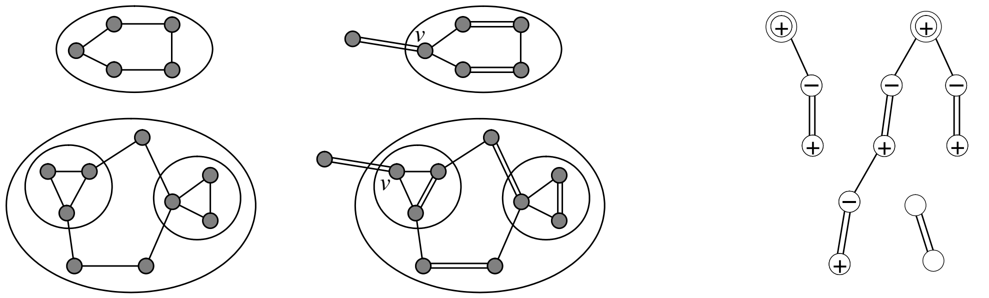
It should be said that there is one important difference between a blossom v and an ordinary node $u \in V$: variable y_v must be non-negative, while y_u can take any sign. Because of this blossoms sometimes have to be expanded to prevent y_v from becoming negative.

2.1 Overview of the algorithm

The algorithm works only with exterior pseudonodes; pseudonodes and edges inside blossoms are not considered (unless the blossom is expanded). From now on, we will refer to exterior pseudonodes as just "nodes", unless stated otherwise.

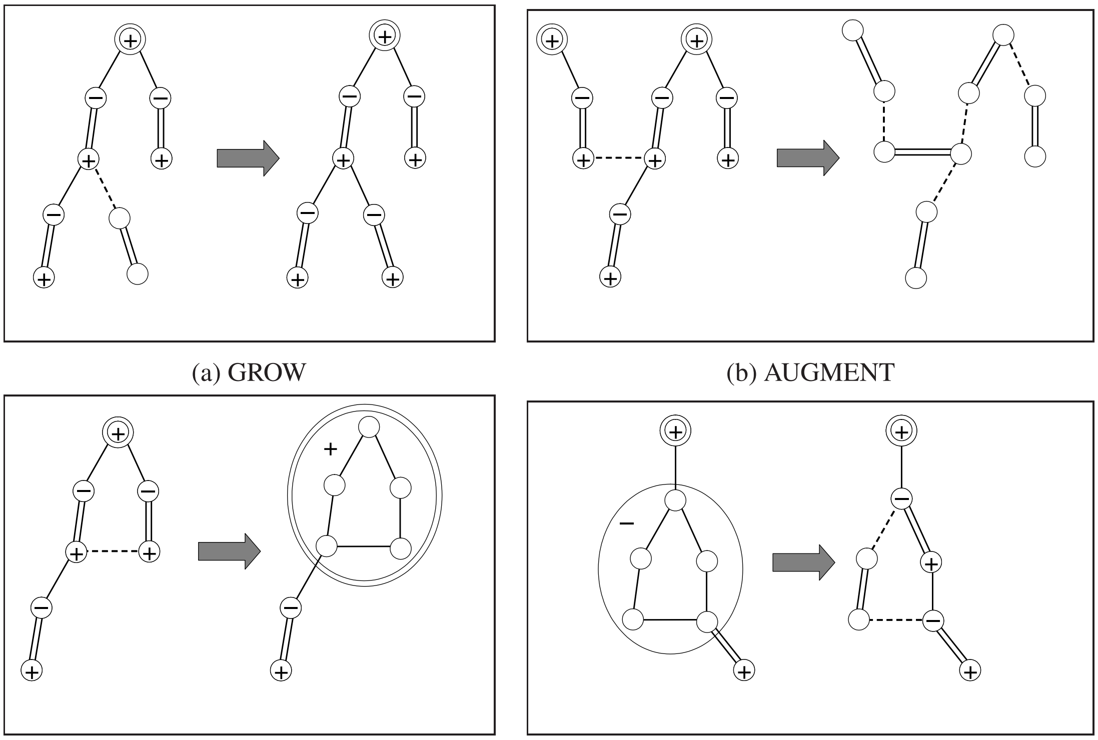
Each node v has a label $l(v) \in \{+, -, \varnothing\}$. Nodes v with label $l(v) = \varnothing$ are called *free* nodes; they are always matched to another free node via a tight edge. If $l(v) \neq \varnothing$, then v belongs to an *alternating tree* (Fig. 1(c)). If $l(v) = -$ then the parent of v is a "+" node, to which v is connected via a tight unmatched edge e (i.e. $x_e = 0$). If $l(v) = +$ then the parent of v is a "$-$" node, to which v is connected via a tight matched edge e ($x_e = 1$). The only exception is the root of the tree, which has the label "+" but is unmatched. Note, "+" nodes may have several children (or none), but a "$-$" node always has one child. Clearly, the number of trees equals the number of unmatched nodes in the graph.

The algorithm iterates between "primal updates" and "dual updates", which are described below.



(a) (b) (c)

Figure 1: **Basic notions. Solid lines show blossom forming edges or edges in a tree. Double solid lines show edges in the current matching. Other edges of the graph are not shown.** (a) Two valid blossoms. (b) Canonical matchings obtained after node v inside the blossom is matched to an exterior node. (c) A possible intermediate state of the algorithm. There are two alternating trees (their unmatched roots are at the top) and two free nodes.



(a) GROW (b) AUGMENT

(c) SHRINK (d) EXPAND

Figure 2: **Four possible operations performed during the primal updates.** Dashed lines denote tight edges which are not in any tree. In (c) the shrunk blossom becomes the root of the tree, since the original root is subsumed. The EXPAND operation for blossom v is allowed only if $l(v) = -$ and $y_v = 0$.

2.2 Primal updates

During primal updates the algorithm attempts to find a matching x of higher cardinality that uses only tight edges. (Dual variables y are kept constant.) Four operations are used:

- GROW: If edge (u, v) is tight, $l(u) = +$ and $l(v) = \varnothing$ then the tree to which u belongs can be "grown" by acquiring node u and the corresponding matched node (Fig. 2(a)).
- AUGMENT: If edge (u, v) is tight, $l(u) = l(v) = +$ and u, v belong to different trees then the cardinality of matching x can be increased by "flipping" variable x_e for edges e along the path connecting the roots of the two trees (Fig. 2(b)). All nodes in the trees become free.
- SHRINK: If edge (u, v) is tight, $l(u) = l(v) = +$ and u, v belong to the same tree then there is cycle of odd length that can be shrunk to a blossom (Fig. 2(c)). The dual variable for this new blossom is set to 0.
- EXPAND: If node v is a blossom with $y_v = 0$ and $l(v) = -$ then it can be expanded (Fig. 2(d)).

Note, AUGMENT is the only operation that changes the current matching x. The cardinality of the matching is increased by 1, so there are at most $n/2$ augmentations. Arguably, it is the most desirable operation and should have a priority over other operations.

2.3 Dual updates

These updates modify dual variables y for nodes in the trees while keeping trees and blossoms intact. For each tree T we choose the amount of dual change $\delta_T \geq 0$. For node $v \in T$ the dual variable y_v is then updated as follows: if $l(v) = -$ then set $y_v := y_v - \delta_T$, and if $l(v) = +$ then set $y_v := y_v + \delta_T$. The value of the dual objective function is increased by $\sum_T \delta_T$. Vector y must remain a feasible dual, which gives the following constraints on $\{\delta_T\}$:

$$
\begin{array}{rcll}
\delta_T & \leq & \texttt{slack}(u,v) & (u,v) \text{ is a } (+,\varnothing) \text{ edge, } u \in T \qquad \text{(4a)} \\
\delta_T + \delta_{T'} & \leq & \texttt{slack}(u,v) & (u,v) \text{ is a } (+,+) \text{ edge, } u \in T, v \in T', T \neq T' \qquad \text{(4b)} \\
\delta_T & \leq & \texttt{slack}(u,v)/2 & (u,v) \text{ is a } (+,+) \text{ edge, } u,v \in T \qquad \text{(4c)} \\
\delta_T & \leq & y_v & v \text{ is a ``}-\text{'' node which is a blossom, } v \in T \qquad \text{(4d)} \\
\delta_T - \delta_{T'} & \leq & \texttt{slack}(u,v) & (u,v) \text{ is a } (+,-) \text{ edge, } u \in T, v \in T', T \neq T' \qquad \text{(4e)}
\end{array}
$$

Suppose that the change δ_T for tree T is set to the maximal value so that one of the constraints in (4) becomes tight. If such constraint is (4a), then after the dual change the $(+, \varnothing)$ edge (u, v) becomes tight, so the GROW operation can be applied. Similarly, if constraints (4b),(4c),(4d) are tight then the operations AUGMENT, SHRINK, EXPAND can be performed, respectively. The only case when no immediate progress can be made on tree T is when δ_T is determined by the last constraint (4e).

Let us now discuss specific strategies for choosing values $\{\delta_T\}$. Techniques proposed in the literature can be classified as follows:

1. A single tree approach. All primal and dual operations are applied to the current tree T until it is augmented. All other trees consist only of single root nodes (labeled as "$+$"). Note, case (4e) can never arise since there are no "$-$" nodes outside the current tree T.

2. A multiple tree approach with fixed δ (i.e. $\delta_T = \delta$ for all trees T). δ is set to the maximum value subject to constraints (4). Case (4e) is not a concern since $\delta_T - \delta_{T'} = 0$ for all pairs of trees T, T'. Thus, δ is determined by one of the constraints (4a),(4b),(4c),(4d), and so after the dual update further progress can be made in at least one of the trees.

3. A multiple tree approach with variable δ. This is the most flexible approach proposed by Cook and Rohe [8]. Updates $\{\delta_T\}$ could potentially be chosen as to maximize the increase in the dual objective $\sum_T \delta_T$, but this would be too costly. Instead, a certain greedy heuristic is used in [8] (see section 3.1).

Let us call the sequence of operations between two successive augmentations a "stage". Clearly, there are at most $n/2$ stages. It can be shown that each stage can perform at most $O(n)$ basic operations GROW, SHRINK, EXPAND. Assuming that (i) each dual update results in at least one basic operation and that (ii) each GROW, SHRINK, EXPAND, and a dual update take $O(m)$ time, we arrive at a straightforward bound $O(n^2 m)$ for the overall complexity.

Note that assumption (i) is valid for the first two approaches. However, in the third approach values of δ_T may be determined by the constraint (4e) rather than (4a),(4b),(4c),(4d). Thus, the $O(n^2 m)$ complexity is not guaranteed for the variable δ approach.

2.4 Comparison of different strategies

There are several factors to consider when choosing which strategy to use. The first factor may be the number of basic operations such as GROW, SHRINK, and EXPAND. The single tree strategy is usually considered to be the least efficient from this point of view [19, 8]. An intuition here is that exploring a single tree may result in a very long augmenting path and consequently in many basic operations, while a much shorter augmenting path may exist which starts from a different tree. The single tree strategy becomes particularly inefficient towards the end. After extensive experiments with the single tree approach and with the multiple tree approach with fixed δ Gerngross [19] suggested to use the former for matching the first 95% of the nodes and the latter for matching the remaining 5%. (Matching the last few remaining nodes often takes the most amount of time.)

Let us now discuss the two strategies that grow multiple trees. Cook and Rohe [8] motivated the variable δ approach over fixed δ as follows: computing the value of δ requires examining the edges out of the "$+$" nodes in all trees, but tight edges will be created only for a very small number of these trees. The experiments in [8] show an improvement of the multiple tree strategy with variable δ over the single tree implementation of Applegate and Cook [5].

It is not clear, however, whether the explanation in the previous paragraph still holds if priority queues are used for computing δ, as discussed below. In this case the examination of the edges out of the "+" nodes is not performed explicitly. Mehlhorn and Schäfer [24] implemented a multiple tree approach with fixed δ and with an extensive use of priority queues, and showed an improvement over the code of Cook and Rohe, sometimes asymptotic.

Still, the variable δ approach has a certain advantage over the fixed δ since it can increase the dual objective by a much larger amount. (With fixed δ the dual update is determined as a bottleneck among all trees, and thus can be quite small.) Intuitively, larger dual updates should result in a smaller number of basic operations.

Using priority queues The second factor to consider when selecting a strategy is the efficiency of computing dual updates satisfying constraints (4). A rather natural idea that has been exploited both for improving the worst-case bound and for faster real implementations is to store edges in priority queues, so that the operation of computing the minimum slack required in (4) can be performed efficiently.

To our knowledge, so far priority queues have been used only for the first two strategies (single tree and multiple trees with fixed δ). The implementation of Mehlhorn and Schäfer uses a special type of priority queues called *concatenable priority queue* within multiple trees with fixed δ framework. The reason for using this version of priority queues is that it can handle SHRINK and EXPAND operations. The implementation in [24] achieves the $O(nm \log n)$ worst-case bound proved in [28].

Mehlhorn and Schäfer pose the question whether the variable δ approach can be incorporated in a $O(nm \log n)$ algorithm. Unfortunately, this does not look straightforward. The difficulty lies in the following: with the fixed δ the slacks of edges of the same type, e.g. exterior $(+,+)$ edges, are decreased by the same amount 2δ, so all these edges can be stored in a single priority queue. This is no longer the case in the variable δ approach. Edges between different pairs of trees should thus be stored in different priority queues. As the worst case, the number of trees may be $\Omega(n)$ and the number of edges between trees may be $\Omega(m)$. (We say that there is an edge between trees T and T' if there is an edge of type $(+,+)$, $(+,-)$ or $(-,+)$ connecting a node $u \in T$ to a node $v \in T'$.) If computing the dual updates and finding new tight edges is done in a naïve way by traversing edges between trees, this would result in $O(m)$ time per dual update, yielding the $O(n^2 m)$ overall bound.

3 Description of our implementation

In this work we attempt to use priority queues in a variable δ approach. For reasons discussed above we abandoned the idea of achieving a better bound than $O(n^2 m)$. Instead, we assume that the worst-case described above does not occur in practice, i.e. the number of trees and the number of edges between trees are much smaller than the number of nodes and edges in the graph, respectively. This assumption is validated by previous studies: it has been observed that the algorithm often spends most of the time trying to match the last few remaining nodes, see e.g. [8] (although constructing a counter-example could potentially be possible[1]).

We maintain dynamically an auxiliary graph $(\mathcal{V}, \mathcal{E})$ whose nodes correspond to alternating trees T. This graph is stored using the adjacency list representation. Each node T of the graph has pointers to three priority queues $pq^{++}(T)$, $pq^{+\varnothing}(T)$ and $pq^{-}(T)$. These queues store respectively $(+,+)$ edges between "+" nodes of the tree T, $(+,\varnothing)$ edges from "+" nodes of T to free nodes, and "$-$" nodes in T which are blossoms. Similarly, an edge $(T,T') \in \mathcal{E}$ has pointers to priority queues $pq^{++}(T,T')$, $pq^{+-}(T,T')$ and $pq^{-+}(T,T')$ which store respectively $(+,+)$, $(+,-)$ and $(-,+)$ edges between a node in T and a node in T'.

We use Fibonacci heaps [15] as our priority queues. It requires 4 pointers per edge, an integer indicating the degree of the heap node representing the edge, and a binary flag. Each edge belongs to at most one queue, so no extra memory is allocated. It should be noted, though, that Fibonacci heaps may not be the optimal choice: it has been shown [25] that in practice other data structures such as pairing heaps [14] are more efficient. Pairing heaps also take less memory. We plan to investigate alternative priority queue implementations in the future.

When performing primal updates for tree T, we need to determine quickly for edge $(u,v) \in E$ with $u \in T$ the corresponding edge $(T,T') \in \mathcal{E}$ (if it already exists). One option would be to store an array of pointers at each tree. However, this would require $O(|\mathcal{V}|^2)$ memory, where $|\mathcal{V}|$ is the number of trees in the beginning of the algorithm (which can be quite large). We avoid the quadratic memory requirement by using the following scheme. We store a pointer `CURRENT_EDGE` at each tree. Initially, these pointers are `NULL`. Before processing T we go through edges $(T,T') \in \mathcal{E}$ and set `CURRENT_EDGE`(T') to point to the edge (T,T') (and simultaneously check whether an augmentation is possible for trees (T,T')). After the update we go through the edges (T,T') again and set the pointers back to `NULL`.

[1] Consider, for example, a graph consisting of $O(\sqrt{n})$ components, where each component contains an $O(\sqrt{n})$ odd number of nodes. Assume that each component is connected via low-cost edges and different components are connected via high-cost edges, so that augmentations between two components can be performed only when one of the components has been completely shrunk. Consider the first stage when no component is completely shrunk. Intuitively, it seems possible to construct a graph such that there would be $\Omega(\sqrt{n})$ trees in this stage and $\Omega(\sqrt{n})$ dual updates (determined by low-cost edges), assuming that each update results in at most one SHRINK in each component.

It is not difficult to see how to maintain the auxiliary graph during primal updates, as well the correct membership of edges in priority queues. Consider, for example, the GROW operation. Suppose that a free matched edge (u,v) has been added to a tree T, so that u becomes a "$-$" node and v becomes a "$+$" node. First, we process node u. We add u to the queue $pq^{-}(T)$, if u is a blossom. We then go through all incident edges (u,w). If w is a "$+$" node in some tree T' then we do the following:

- Remove (u,w) from the queue $pq^{+\varnothing}(T')$. (Note, each node has a pointer to the tree that it belongs to, so node T' in the auxiliary graph can be easily located.)
- If $T' \neq T$ then we locate the edge (T,T') using pointer CURRENT_EDGE(T') and add (u,w) to the queue $pq^{-+}(T,T')$. (It may happen that CURRENT_EDGE(T') is NULL if there is no edge (T,T'); in that case we first allocate and add this edge to the auxiliary graph and set CURRENT_EDGE(T') accordingly.)

After processing u we go through all edges (v,w) and do the following:

- If w is free, add (v,w) to $pq^{+\varnothing}(T)$.
- If $w \in T$ and $l(w) = +$, remove (v,w) from $pq^{+\varnothing}(T)$ and add it to $pq^{++}(T)$.
- If $w \in T'$, $T' \neq T$ and $l(w) = -$, add (v,w) to $pq^{+-}(T,T')$.
- If $w \in T'$, $T' \neq T$ and $l(w) = +$, remove (v,w) from $pq^{+\varnothing}(T)$ and add it to $pq^{++}(T,T')$.

In the last two cases we first need to make sure that an edge (T,T') exists by allocating it, if necessary.

SHRINK, EXPAND and AUGMENT operations can be considered similarly. Note, when augmenting trees (T,T') we traverse all edges stored in all queues associated with T,T' and incident edges of the auxiliary graph, and deallocate these trees and edges. This concludes the description of operations performed with the auxiliary graph.[2]

Maintaining dual variables An important issue is how to store edge slacks and variables y_v for nodes v. Maintaining them explicitly would be too costly since after each dual update we would need to go through all nodes and edges in the tree and update these variables. Following a long tradition, we use an implicit method which avoids going through all nodes and edges. Namely, each node v has variable $\bar{y}_v$. The true variable y_v can be determined as follows. For interior nodes inside blossoms and for free exterior nodes we have $y_v = \bar{y}_v$. For an exterior node in tree T we have $y_v = \bar{y}_v + \epsilon_T$ if $l(v) = +$, and $y_v = \bar{y}_v - \epsilon_T$ if $l(v) = -$. Here ϵ_T is a variable stored at tree T which accumulates all dual updates for T. We use an analogous technique for edges. Each edge $a = (u,v)$ stores a variable $\overline{\texttt{slack}}(a)$. A true slack for an exterior edge $a = (u,v)$ is determined as follows: (1) take value $\overline{\texttt{slack}}(a)$; (2) if $u \in T$ then subtract or add ϵ_T, depending on whether $l(u) = +$ or $l(u) = -$; (3) do the same for node v. Whenever a node changes its label, we go through the incident edges a and update $\overline{\texttt{slack}}(a)$ accordingly. This is done at the same time as we update the membership of a in priority queues.

3.1 Updating duals

In this section we discuss how to compute updates $\{\delta_T\}$. A possible heuristic would be to go through trees T in a certain order and greedily increase δ_T as much as possible. Unfortunately, this procedure would get stuck if there are circular constraints $\delta_{T_1} - \delta_{T_2} \le 0, \ldots, \delta_{T_{k-1}} - \delta_{T_k} \le 0, \delta_{T_k} - \delta_{T_1} \le 0$. To overcome this issue, Cook and Rohe propose first to compute strongly connected components (SCC) over trees in order to detect such circular constraints, and then use a greedy technique in which a single δ is used for each component. Their Blossom IV implementation actually uses connected components (CC) instead of strongly connected components; this still guarantees that some progress will be made.

It is worth noting that such greedy dual updates do not guarantee to make one of the inequalities (4a)-(4d) tight. If the CC procedure is used, then it may take up to $O(n)$ dual updates before a basic primal operation can be applied. (Note, each CC update either results in a basic primal operation or merges some components together.) The bound $O(n)$ can be tight if, for example, there are circular constraints $\delta_{T_1} - \delta_{T_2} \le 1, \ldots, \delta_{T_{k-1}} - \delta_{T_k} \le 1, \delta_{T_k} - \delta_{T_1} \le 1$. For this reason we believe that the Blossom IV implementation is actually $O(n^3 m)$, rather than $O(n^2 m)$.[3] The same applies to our code[4]. Interestingly, with the SCC procedure it may take a non-polynomial number of dual updates before a basic primal operation can be applied, as can be demonstrated on the same example.

[2] Our implementation is actually slightly more complicated than what is described above. Namely, if we encounter a $(+,\varnothing)$ edge (v,w) with slack 0 while growing node v, then instead of adding (v,w) to $pq^{+\varnothing}(T)$ we add w and the corresponding matching node to the tree, and mark them as "unprocessed". Trees are then grown in a depth-first search fashion. Other operations during GROW and AUGMENT should thus take into account the "processed" status of nodes. SHRINK and EXPAND are only called if there are no unprocessed nodes.

[3] I thank Bill Cook for confirming this conclusion.

[4] It is not difficult to improve the complexity to $O(n^2 m)$. For example, one could check after each pass over trees whether any progress has been made, and if not make an extra call to the dual update procedure with fixed δ. We decided not to do this, however, in order to guarantee that the dual variables stay half-integral for integral input weights.

We chose the following scheme as our default option. We go through trees in a cyclic order performing primal updates. If no augmentations were performed during a cycle then we run the heuristic dual update procedure of Cook and Rohe with connected components.

Cook and Rohe ask the question whether maximizing the dual objective $\sum_T \delta_T$ may give any benefit, although they remark that solving this linear program would be prohibitively costly. We decided to investigate this question. We solved the linear program by reducing it to (the dual of) the minimum cost network flow in a graph with $2|\mathcal{V}|$ nodes and $2|\mathcal{E}|$ edges using the transformation described by Hochbaum [20], and applying the successive shortest path algorithm of Ford and Fulkerson [4] to the latter problem. In many cases we did not see any significant difference. However, for one class of problems (structured geometric instances) solving the LP appears to perform better than greedy dual updates.

The structure of our implementation suggests another possibility for updating duals. Recall that we go through edges (T, T') before and after processing tree T for updating `CURRENT_EDGE` pointers. We then may as well compute the maximal dual change δ_T for this tree. Unfortunately, we believe that the number of steps can then be non-polynomial, as with the SCC procedure. Nevertheless, in our informal tests we did not see major differences between different versions of greedy dual updates. In our initial experiments (not reported here) we updated duals more frequently (after processing each tree T), and we also used SCC updates at the end of each cycle with no augmentations. For some classes of problems this version actually seemed to perform slightly better than our default scheme with CC updates only.

3.2 Data structures

For completeness, we sketch the data structures that we used. The graph is stored using the adjacency list representation. Each edge a has pointers `TAIL_ORIG`(a), `HEAD_ORIG`(a), `TAIL`(a), `HEAD`(a); the first two point to the original nodes in V, and the last two point to current pseudonodes. For each interior node we also store an `ANCESTOR` pointer in addition to the `PARENT` pointer. `ANCESTOR` pointers help to determine the exterior grandparent node for an interior node more quickly; they are updated using the path compression technique. In our implementation we try to make sure that `ANCESTOR`'s point to penultimate nodes (i.e. interior nodes whose parents are exterior). Note, expanded nodes cannot be deallocated immediately since some `ANCESTOR`'s may point to them. Instead, if the number of expanded nodes exceeds a certain threshold we go through all nodes, correct `ANCESTOR`'s and only then deallocate expanded nodes.

We implemented two versions of the algorithm; we refer to them as A and B. Version A is somewhat simpler. We perform all shrinks and expands explicitly, by moving edges from one pseudonode to another and updating `TAIL` or `HEAD` pointers accordingly. Edges a inside blossoms are not in any priority queue, and satisfy $\mathtt{slack}(a) = \overline{\mathtt{slack}}(a)$.

With this structure we observed that sometimes SHRINK operations take a very long time and become the bottleneck of the algorithm. A closer inspection revealed that there is a "cascading" sequence of shrinks during the same stage in which the degree of the blossom becomes extremely large. For each shrink we go through all incident edges and update `TAIL` or `HEAD` pointers again and again, and this becomes the bottleneck.

To fix this issue, we implemented a modified version, which we call version B. (We used this version in all experiments.) Shrinks are now performed lazily: when contracting an odd cycle into a blossom, we go through the incident edges of "$-$" nodes, but not "$+$" nodes.[5] As a result, the `TAIL`/`HEAD` pointers for exterior edges may not be correct, i.e. they do not point to an exterior node. Thus, when processing an edge we must now always check for this, and update the `TAIL`/`HEAD` pointers if necessary. During this update we also move the edge from the adjacency list of the interior node to the list of the exterior node. (This implies that we now need to use doubly linked lists).

In version B priority queues pq^{++} for trees T may now contain interior edges. Thus, whenever we call the `FindMin` operation for this priority queue, we need to check the returned edge. If this edge is interior, we remove it from the queue and move it to the list of "selfloops" for the penultimate node. The only exception is when the ends of the edge are already penultimate nodes, then we insert the edge to the correct place immediately. When the tree is augmented we go through edges stored in all priority queues associated with the tree (and also through all edges incident to "$-$" nodes), and correct inconsistencies. Processing each edge during these operations is quite efficient because of the use of `ANCESTOR` pointers.

When we expand an exterior node, we process selfloops at penultimate nodes. These edges are moved one level further.

It is worth noting that similar "cascading" behavior can potentially occur during EXPAND operations; the same edge out of a "$-$" node could be processed many times during the same stage. We did not put much effort into

[5] After completing the first draft of this paper, we realized that the "cascading shrinks" problem has been fixed in version B only partially. Although we no longer go through all incident edges of "+" nodes, we traverse their tree children. During a "cascading" sequence of shrinks the blossom may acquire an extremely large number of children, which are traversed again and again. We observed that for the last instance in table 9 the B5 method spends most of the time traversing children in SHRINK operations. We hope to fix this issue in the future.

optimizing EXPAND operations since in our experience the number of expands is usually significantly smaller than the number of shrinks (so most blossoms are never expanded). This observation also motivated some of the choices described earlier; in particular, the use of “selfloops” defers an expensive part of the operation (determining precise ends of an edge) to a later EXPAND operation, which may never occur.

In most of our tests the time spent in the EXPAND operations was negligible. However, we did encounter several large instances in which these operations were a bottleneck.

3.3 Initialization

Similar to the implementations of [8, 24], our code supports the two standard strategies for finding the initial solutions:

- *Greedy initialization*. First, for each node v we go through the incident edges, find the smallest weight and set y_v to the half of this weight. (This guarantees that all edges now have non-negative slacks.) We then go over nodes again, and for each node v greedily increase y_v and choose a matching for v, if possible.

- *Fractional matching initialization*. Such initialization was proposed by Derigs and Metz [9]. It first solves the *fractional matching* problem (consisting of constraints (1b) and (1d) only) and then rounds the obtained half-integral matching. We implemented the approach described by Applegate and Cook [5] with Fibonacci heaps as priority queues for solving the fractional matching problem.

3.4 Price-and-repair

A standard approach for solving very dense instances (e.g. complete graphs) is to use a *price-and-repair* technique [10, 5, 8]. It starts by selecting an initial (sparse) subset of edges and then iterates the following procedure. First, the optimal perfect matching is computed over the current subset of edges. The next step is to compute slacks of the remaining edges. (This operation is called *pricing*). If all slacks are non-negative then the method stops - we have an optimal solution. Otherwise we add edges with negative slacks and repeat the procedure.

If a few new edges have been added, then it may be more efficient to initialize the perfect matching algorithm by *repairing* the existing solution rather than starting from scratch. Several repair techniques have been proposed in the literature. We implemented the *careless repairs* method of Cook and Rohe [8]. (They showed that careless repairs outperform the method of Ball and Derigs [6].)

Our code supports the price-and-repair approach by providing interface functions `StartUpdate`, `AddNewEdge`, `FinishUpdate`. The `AddNewEdge` function for nodes u and v computes the slack of the edge (u, v) and adds the edge only if the slack is negative. Note, if u and v belong to the same blossom then computing the slack involves finding the least common ancestor (LCA) of these nodes in the tree formed by the blossom structure. (More precisely, we need to determine nodes that are penultimate to LCA(u,v).) Previous implementations of Applegate and Cook [5] and Cook and Rohe [8] used efficient data structures for determining LCA. In our code we implemented the data structure of Berkman and Vishkin [7] that allows to compute LCA in $O(1)$ time. This data structure is constructed during `StartUpdate` and deallocated during `FinishUpdate`.

Solving complete geometric instances In [5, 8] the price-and-repair approach was used for solving complete geometric instances (that is, given a set of 2D points, compute a minimum cost perfect matching in the complete graph induced by these points; the edge weight is taken as the Euclidean distance rounded to the nearest integer.) We added this functionality to our code as well. A key computational task here is identifying edges in the complete graph with negative slacks. Going through all edges explicitly would be computationally infeasible. Instead, we use the underestimate of the edge slack proposed by Applegate and Cook [5]:

$$\mathtt{slack}(u, v) \geq \gamma(u, v) = c_{uv} - \mathtt{sum}(u) - \mathtt{sum}(v)$$

where

$$\mathtt{sum}(v) = y_v + \sum_{S \in \mathcal{O}: v \in S} y_S$$

As suggested in [5], we find edges (u, v) with negative $\gamma(u, v)$ and call the `AddNewEdge` function only for those edges. Following Cook and Rohe [8], we implemented a kd-tree structure for identifying edges with $\gamma(u, v) < 0$. ($\mathtt{sum}(\cdot)$ is treated as an extra geometric coordinate and $\gamma(\cdot)$ is treated as the distance function.)

4 Computational results

In this section we compare the performances of the Blossom IV code [8], the code of Mehlhorn and Schäfer [24], and our code [1], version 1.0. We will refer to them as B4, MS and B5, respectively. We used the MS code accompanying the paper [24]. It requires the LEDA library [23], version 4.2. Unfortunately, this version was no longer available. Instead, we used the current free version of LEDA (`LEDA-6.0-free-FC8_i386-g++-4.1.2-std`), but we had to make changes to the MS perfect matching code to make it compile with the newer library. Mainly, it involved changing the locations of .h files.[6] Note, the MS algorithm that we tested is not identical to the algorithm in LEDA 6.1. A comparison with the latter code is left as a future work.

The tests were performed on Intel Pentium III processor, 1133MHz with 512KB cache and 2GB memory running Linux 2.6.9. Codes were compiled with the GNU c++ compiler, version 3.4.6, using the -O5 optimization flag. Running times were measured via the `clock_gettime(CLOCK_PROCESS_CPUTIME_ID, ...)` function. We counted only the time spent in the main function that is called for computing the perfect matching; time spent for reading the problem from a file and allocating the graph was not included.

Below, we report the times in seconds of 6 techniques: B4$^-$, B4, MS, MS$^-$, B5$^-$, B5. (The "$-$" flag indicates that a greedy initialization was used instead of the fractional matching procedure of Derigs and Metz [9] - see section 3.3). For the B5 code we also measured the time for initialization (either greedy or fractional) and the time spent in the EXPAND operations; they are given in square brackets.

We observed that sometimes the MS code had large memory requirements. For some instances it caused disk swapping; the Unix `top` command reported that almost all available memory of 2Gb was allocated to this process. In the tables below we indicate such cases by "mem". The memory usage of B4 and B5 was manageable; for example, for the largest instance in table 5 B4 required 323Mb (245Mb for B5), and for the largest instance in table 9 B4 required 181Mb (183 for B5).

In two cases the MS code also gave incorrect results (table 2) - the returned solution was either not a perfect matching or had a larger cost than the solutions returned by B4 and B5. We believe that this was caused by an overflow: for these instances the cost of the matching exceeded the capacity of 32-integer numbers[7]. These cases are marked by "of".

We used 9 problem types. Note, generators for some of them take a random seed as an input. In such cases we report the average statistics over t instances with different random seeds; the number of trials t is then specified in the table caption. Graph sizes are indicated in the tables (columns n and m). For all experiments in this paper (except for those in section 4.2) we randomly permuted the order of nodes and edges of the input graph.

Triangulation instances (tables 1,2) We generated n random points from a $2^{20} \times 2^{20}$ square and then computed triangulations of this point set. As in [24], we tested two kinds of triangulations: Delaunay triangulations (computed with the code of Shewchuk [26]) and triangulations by a sweep-line algorithm (we used the generator accompanying the paper [24]). Edge weights were set to the Euclidean distance between the endpoints (rounded to the nearest integer for Delaunay triangulations, and rounded down for sweep-line triangulations).

Random instances (tables 3,4) We generated random graphs with n nodes and m edges (while ensuring that a perfect matching exists). Edge weights were assigned a random integer value in $[1, 2^{16}]$. We used the same sizes as in [24].

Planar Ising models (table 5) We considered the problem of computing a minimum cut in a planar graph, where edge weights can have an arbitrary sign. (Clearly, it is equivalent to a maximum cut problem.) In physics it is often called the ground state computation of an Ising model without magnetic field. We reduced the problem to a minimum cost perfect matching problem as described in [27]. (This was the application that motivated our interest in the perfect matching problem.)

We used 4-connected square grids. Edge costs for the Ising problem were chosen as random integers from $[-2^{15}, 2^{15}]$.

DIMACS instances (tables 6,7,8) We used three families from the first DIMACS implementational challenge [21]. They were generated by the following programs:

- hardcard.f written by B. Mattingly. This family was shown by Gabow to be hard for Edmonds's cardinality matching algorithm.
- t.f and tt.f written by N. Ritchey and B. Mattingly. They generate a sequence of K one- and tri-connected triangles, respectively. According to comments, the first family tends to generate a lot of blossoms.

All three programs take an input number K; it is specified in the tables 6,7,8.

[6] We contacted the authors of [24] about the compilation issue. Guido Schäfer recommended using the latest version of LEDA (which now incorporates a min cost perfect matching algorithm), as compiling the original code of [24] with a newer version of LEDA comes with no warranty. Unfortunately, we were not able to do this at the time of writing: we discovered that the perfect matching algorithm in a newer version of LEDA had a bug which caused the program to crash. The support team at Algorithmic Solutions Software GmbH informed us that the bug had recently been fixed, but the fix will appear only in the next official release of LEDA after LEDA 6.1 scheduled for the second half of October 2008.

[7] Guido Schäfer informed us that some checkers had been added to the subsequent version of the code that verify whether an overflow might occur.

n	m	B4^{-}	B4	MS^{-}	MS	B5^{-}	B5
10000	29973	0.41	0.39	1.27	1.03	0.13 [0.02,0.00]	**0.11** [0.04,0.00]
20000	59971	2.05	2.01	2.70	2.20	0.30 [0.05,0.01]	**0.27** [0.08,0.01]
40000	119968	7.70	7.59	5.74	4.74	0.68 [0.11,0.01]	**0.60** [0.17,0.02]
80000	239970	22.1	21.8	12.2	10.0	1.49 [0.23,0.04]	**1.33** [0.37,0.04]
160000	479960	53.6	52.7	30.5	21.3	3.34 [0.48,0.12]	**3.02** [0.77,0.12]

Table 1: **Delaunay triangulations.** $t = 20$.

n	m	B4^{-}	B4	MS^{-}	MS	B5^{-}	B5
10000	29973	1.74	1.66	1.56	1.10	0.15 [0.02,0.00]	**0.13** [0.04,0.00]
20000	59971	8.06	7.89	3.38	2.39	0.38 [0.05,0.01]	**0.32** [0.10,0.01]
40000	119968	38.4	38.1	7.53	5.28	0.92 [0.11,0.03]	**0.77** [0.22,0.03]
80000	239970	188	188	19.2	of	3.17 [0.22,0.26]	**2.83** [0.47,0.24]
160000	479960	550	551	43.4	of	8.75 [0.47,0.96]	**8.02** [0.98,0.94]

Table 2: **Sweep-line triangulations.** $t = 20$.

n	m	B4^{-}	B4	MS^{-}	MS	B5^{-}	B5
10000	60000	6.90	1.29	3.01	1.24	**0.41** [0.04,0.00]	0.55 [0.53,0.00]
10000	80000	9.55	1.67	3.78	1.47	**0.56** [0.05,0.00]	0.78 [0.76,0.00]
10000	100000	2.94	2.25	4.55	1.56	**0.73** [0.06,0.00]	0.99 [0.97,0.00]
20000	120000	7.93	4.92	6.90	3.23	**0.99** [0.09,0.00]	1.76 [1.72,0.00]
20000	160000	21.5	7.79	8.74	3.72	**1.40** [0.12,0.00]	2.31 [2.29,0.00]
20000	200000	15.2	10.7	10.5	4.65	**1.81** [0.14,0.00]	3.00 [2.95,0.00]
40000	240000	116	22.6	21.6	9.12	**2.40** [0.19,0.01]	5.46 [5.42,0.00]
40000	320000	25.6	31.9	20.1	10.5	**3.20** [0.25,0.04]	7.23 [7.14,0.00]
40000	400000	95.7	43.8	24.5	12.0	**4.16** [0.31,0.18]	8.96 [8.90,0.00]

Table 3: **Sparse random instances.** $t = 20$.

n	m	B4^{-}	B4	MS^{-}	MS	B5^{-}	B5
1000	100000	1.05	0.95	1.61	0.43	0.42 [0.05,0.00]	**0.39** [0.35,0.00]
1000	200000	1.97	2.57	3.21	**0.78**	0.87 [0.10,0.00]	0.80 [0.74,0.00]
1000	400000	3.45	4.56	4.85	**1.25**	1.34 [0.16,0.00]	1.30 [1.18,0.00]
2000	200000	4.15	3.38	4.57	1.18	**1.02** [0.10,0.00]	1.04 [0.99,0.00]
2000	400000	5.86	7.92	9.27	**2.06**	2.16 [0.22,0.03]	2.15 [2.03,0.00]
2000	800000	8.46	12.3	13.1	**3.14**	3.24 [0.33,0.00]	3.30 [3.12,0.00]
4000	400000	8.92	12.0	12.0	2.90	**2.64** [0.23,0.01]	2.81 [2.71,0.00]
4000	800000	25.7	26.0	23.0	6.45	**5.59** [0.49,0.02]	6.60 [6.36,0.00]
4000	1600000	20.5	36.7	34.9	9.47	**8.39** [0.76,0.01]	9.61 [9.20,0.00]

Table 4: **Dense random instances.** $t = 20$.

size	n	m	B4$^-$	B4	MS$^-$	MS	B5$^-$	B5
25×25	6230	10591	0.05	0.05	0.52	0.53	**0.03** [0.01,0.00]	**0.03** [0.01,0.00]
35×35	12230	20791	0.19	0.18	1.11	1.12	**0.07** [0.01,0.00]	**0.07** [0.02,0.00]
50×50	24980	42466	0.58	0.59	2.36	2.39	**0.17** [0.04,0.00]	**0.17** [0.06,0.00]
71×71	50390	85663	2.27	2.33	5.02	5.07	**0.38** [0.09,0.01]	0.39 [0.12,0.01]
100×100	99980	169966	7.14	7.09	10.7	10.7	**0.82** [0.18,0.02]	0.83 [0.25,0.02]
141×141	198790	337943	19.7	19.0	22.9	22.6	**1.75** [0.38,0.04]	1.81 [0.52,0.04]
200×200	399980	679966	75.2	75.2	mem	mem	**3.81** [0.79,0.10]	4.02 [1.09,0.10]
283×283	800870	1361479	222	211	mem	mem	**8.50** [1.69,0.23]	8.70 [2.32,0.22]
400×400	1599980	2719966	634	645	mem	mem	**18.2** [3.59,0.40]	18.8 [4.93,0.39]

Table 5: **Planar Ising models.** $t = 20$. Note, "size" corresponds to the input planar grid graph, while n and m correspond to the graph in which a perfect matching is computed.

K	n	m	B4$^-$	B4	MS$^-$	MS	B5$^-$	B5
100	600	80000	3.1	**0.02**	0.24	0.08	0.59 [0.03,0.00]	0.05 [0.05,0.00]
200	1200	320000	32.2	**0.21**	1.28	0.32	4.86 [0.13,0.00]	0.23 [0.23,0.00]
400	2400	1280000	341	10.3	9.68	1.37	29.6 [0.59,0.00]	**1.10** [1.10,0.00]
800	4800	5120000	3607	**2.61**	41.1	5.98	241 [3.18,0.00]	6.26 [6.24,0.00]
1600	9600	20480000	-	787	177	31.6	1637 [16.6,0.00]	**30.5** [30.5,0.00]

Table 6: **DIMACS instances: hardcard.f generator.**

K	n	m	B4$^-$	B4	MS$^-$	MS	B5$^-$	B5
10000	30000	39999	5.57	3.13	2.89	3.11	**0.16** [0.04,0.00]	0.17 [0.06,0.00]
20000	60000	79999	17.3	36.2	6.12	6.54	**0.41** [0.09,0.00]	0.44 [0.14,0.00]
40000	120000	159999	70.6	88.2	13.3	13.5	**0.68** [0.19,0.00]	0.90 [0.30,0.00]
80000	240000	319999	305	1521	30.0	29.3	1.80 [0.40,0.00]	**1.72** [0.58,0.00]
160000	480000	639999	4803	809	mem	mem	**3.67** [0.83,0.00]	3.88 [1.22,0.00]
320000	960000	1279999	1839	1096	mem	mem	**7.93** [1.77,0.00]	9.83 [2.60,0.00]

Table 7: **DIMACS instances: t.f generator.**

K	n	m	B4$^-$	B4	MS$^-$	MS	B5$^-$	B5
10000	30000	59997	0.23	0.26	2.70	2.65	0.25 [0.06,0.00]	**0.21** [0.07,0.00]
20000	60000	119997	0.63	**0.44**	6.32	5.30	0.47 [0.12,0.00]	0.48 [0.15,0.00]
40000	120000	239997	1.02	1.57	14.6	11.3	**0.99** [0.24,0.00]	1.62 [0.33,0.00]
80000	240000	479997	4.83	2.85	28.4	28.0	2.77 [0.50,0.00]	**1.82** [0.67,0.00]
160000	480000	959997	**5.49**	9.35	mem	mem	6.14 [1.06,0.00]	5.59 [1.41,0.00]
320000	960000	1919997	14.9	15.8	mem	mem	**12.4** [2.23,0.00]	16.4 [2.99,0.00]

Table 8: **DIMACS instances: tt.f generator.**

name	n	m	B4$^-$	B4	MS$^-$	MS	B5$^-$	B5
fnl4461	4460	13359	0.04	0.04	0.48	0.37	**0.03** [0.01,0.00]	**0.03** [0.01,0.00]
brd14051	14050	42134	0.35	0.31	2.13	1.75	0.21 [0.03,0.00]	**0.18** [0.06,0.01]
d15112	15112	45800	0.50	0.46	2.18	1.65	0.23 [0.04,0.01]	**0.20** [0.06,0.01]
d18512	18512	55525	0.44	0.38	2.42	1.94	**0.22** [0.05,0.00]	**0.22** [0.08,0.00]
dan59296	59296	177336	1.32	1.15	7.60	6.42	0.93 [0.16,0.03]	**0.80** [0.26,0.03]
sra104815	104814	314416	4.42	4.60	15.3	12.8	2.04 [0.30,0.33]	**1.91** [0.43,0.29]
ara238025	238024	714003	186	261	61.2	**43.9**	147 [0.72,134]	172 [1.06,156]
lra498378	498378	1495586	339	344	mem	mem	125 [1.59,109]	**112** [2.68,96.3]
lrb744710	744710	2234362	6585	4978	mem	mem	4020 [2.47,3146]	**882** [3.79,111]

Table 9: **TSPLIB and VLSI geometric problems.**

Structured geometric instances (table 9) For our last set of problems we used TSPLIB data from [2] maintained by Gerd Reinelt and VLSI data from [3] provided by Andre Rohe. (Some these instances have an odd number of points; in such cases we removed the last point.) The graph was obtained as a Delaunay triangulation of a given set of 2D points (using the code of Shewchuk [26]). Edge weights were set to the Euclidean distance between endpoints rounded to the nearest integer. The Delaunay triangulation is known to contain a perfect matching, however this may not be the case in practice due to numerical errors. To guarantee that the initial subset contains a perfect matching, we compute greedily a matching over the existing edges and then add $n'/2$ edges between remaining n' unmatched nodes (see [1] for more details).

Such problems were used for experiments by Cook and Rohe [8]. They showed that Delaunay triangulation leads to a very good approximation to the complete geometric problem. Delaunay triangulation can also be used as the initial subset of edges for the price-and-repair technique discussed in section 3.4.

From tables 1-9 we can conclude the following:

- In the majority of our examples the B5 code outperforms B4 and MS implementations. For some classes of problems the improvement is roughly by an order of magnitude (tables 1,2,5,7).
- B5 spends most of the time in the fractional matching initialization procedure on problems in tables 3,4,6. Thus, these problems are "easy" in the sense that they can be solved by fractional matching almost to optimality.
- The time in EXPAND operations for the B5 code is negligible in all instances except for large VLSI problems in table 9, where it can become a bottleneck.

Note, the MS code is competitive only on "easy" problems in tables 3,4,6, but with one notable exception - the `ara238025` instance in tables 9. We believe that this instance indicates the importance of data structures that guarantee $O(m \log n)$ work per augmentation for difficult problems.

4.1 Testing dual update strategies

In this section we test the effect of different dual update strategies in our code on a subset of problems used earlier:

[P1] Delaunay triangulation with $n = 80000$ (penultimate line in table 1)

[P2] Sweep-line triangulation with $n = 80000$ (penultimate line in table 2)

[P3] Sparse random problem triangulation with $n = 40000$, $m = 400000$ (last line in table 3)

[P4] Dense random problem triangulation with $n = 4000$, $m = 1600000$ (last line in table 4)

[P5] Planar Ising model 283×283 (penultimate line in table 5)

We ran $t = 5$ trials for each problem above. We also used 5 VLSI instances listed in the second part of table 9; we will refer to them as G1, G2, G3, G4, G5.

For P3 and P4 the fractional matching initialization would already solve the problems almost to optimality, therefore we decided not to use this initialization for them. Thus, we used B5$^-$ for P3, P4 and B5 for other problems.

In table 10 we compare the fixed δ approach and three variable δ approaches: with connected components (CC), strongly connected components (SCC) and solving the LP by transforming it to a minimum cost network flow problem

	fixed δ	variable δ		
		CC	SCC	LP
P1	14.7 [59916,20706,4474]	1.37 [48323,16933,1988]	1.37 [48275,16939,1985]	1.39 [48140,16924,1977]
P2	12.2 [86394,17251,4615]	3.02 [58471,13379,1291]	3.02 [58360,13364,1283]	3.04 [58035,13363,1290]
P3	314 [98712,1736,490]	3.73 [47512,13,3]	3.27 [40050,5,0]	3.20 [38808,5,0]
P4	33.0 [6261,54,17]	8.65 [4020,4,1]	8.06 [3582,10,0]	8.06 [3637,5,0]
P5	1057 [987480,191765,73823]	9.09 [711697,136581,24800]	9.31 [730444,137278,25485]	11.7 [721123,137182,25430]
G1	0.96 [38977,13050,1839]	0.87 [36043,12065,1326]	0.88 [36213,12069,1334]	0.88 [35834,12041,1312]
G2	3.05 [69594,31617,5645]	2.07 [66028,30151,4295]	2.14 [66137,30269,4367]	1.80 [62395,29120,3403]
G3	703 [223537,118898,45075]	173 [203919,102791,27280]	232 [204534,105001,29448]	12.1 [163172,81185,11155]
G4	-	115 [416300,217396,54840]	113 [416308,217184,54719]	28.0 [384720,196242,32288]
G5	-	894 [694748,321946,47283]	908 [699927,323816,49235]	28.0 [484937,200195,23045]

Table 10: **The effect of different dual strategies.** The running time excludes the time spent in dual updates. Double precision numbers are used instead of integers. Format: *primal time* [*GROWs, SHRINKs, EXPANDs*].

	$\mu = 1.0$	$\mu = 0.016$	$\mu = 0.008$	$\mu = 0.004$	$\mu = 0.002$	$\mu = 0.001$
G2	$1.80 + 44.1$ [3403]	$1.93 + 1.30$ [3840]	$1.94 + 0.14$ [3989]	$1.96 + 0.08$ [4023]	$1.84 + 0.05$ [3661]	$1.90 + 0.04$ [3771]
G3	$12.1 + 273$ [11155]	$29.5 + 4.47$ [14919]	$30.4 + 1.14$ [14724]	$31.3 + 0.25$ [15367]	$24.7 + 0.18$ [14819]	$31.3 + 0.12$ [16685]
G4	$28.0 + 753$ [32288]	$27.6 + 91.7$ [33413]	$35.0 + 15.4$ [36298]	$37.3 + 3.16$ [38264]	$77.2 + 0.83$ [45517]	$86.1 + 0.50$ [46489]
G5	$28.0 + 2586$ [23045]	$27.3 + 117$ [26391]	$30.5 + 29.8$ [27976]	$65.2 + 1.49$ [32259]	$95.9 + 0.66$ [32707]	$141 + 0.47$ [33584]

Table 11: **Performance as a function of μ.** (If the number of trees is smaller than μn then duals are updated by solving the LP, otherwise via CC dual updates.) Double precision numbers are used. Format: *primal time+dual time* [*EXPANDs*]. Note, the CC column in table 10 corresponds to $\mu = 0$.

(see section 3.1). Note, in these experiments we used double precision floating point numbers to ensure the correctness of the LP approach[8]. Table 10 reports the running time in seconds **excluding the time for dual updates**. (For fixed δ, CC and SCC the time for dual updates was a small fraction of the total time - 15% at most, but for LP dual updates were the dominating factor.) Three numbers in square brackets represent mean numbers of GROW, SHRINK and EXPAND operations, respectively.

The first conclusion that we can draw from table 10 is that the number of operations in the fixed δ approach is consistently larger than in the variable δ approaches. This confirms the finding of Cook and Rohe [8] about the importance of the variable δ approach. (Note, the running times of the fixed δ approach is also considerably larger compared to CC and SCC, but this is not a fair comparison since the fixed δ approach does not require the auxiliary graph, and thus can potentially be implemented faster.)

Let us discuss the relative performance of CC, SCC and LP. For problems P1-P5 they look comparable, so a global LP approach is unlikely to give much gain even if we could solve the LP very efficiently. However, for VLSI instances G2-G5 solving the LP results in a significantly smaller number of basic operations (especially EXPAND operations) and consequently in a much smaller time for primal updates. Unfortunately, the time for solving the LP negates this gain.

To remedy the situation, we could solve the LP only if the auxiliary graph is sufficiently small, e.g. the number of trees is smaller than μn for some constant $\mu \in [0, 1]$. Table 11 shows how μ influences the performance. As μ decreases, the time for primal updates and the number of EXPAND operations increase, while the time for dual updates decreases. For large VLSI instances G3, G4, G5 we get a significant speed-up with values $\mu \in \{0.002, 0.004, 0.008\}$ compared to the time reported in table 9. This refutes the conjecture of Cook and Rohe [8] that computing an optimal solution of the linear program is unlikely to be of practical value due to the time required to solve the LP.

4.2 Complete geometric instances via price-and-repair

In this section we compare the running times for solving complete geometric instances using the iterative price-and-repair approach discussed in section 3.4. We compare only codes B4 and B5, since the price-and-repair technique is

[8]Using an argumentation similar to the one in [5], one can show that in CC and SCC approaches dual variables are guaranteed to be multiples of 1/2 throughout the algorithm. However, the LP approach does not have such guarantees, as we observed experimentally.

not implemented in the MS code. We used the same initial subset of edges for both codes (computed by the default setting of the Blossom IV code - *quad nearest neighbors*). The running times below exclude the time for generating the initial subset. Note, this experiment was the only one in which nodes and edges were not randomly permuted.

name	B4		B5		B5 (LP, $\mu = 0.005$)	
dan59296	**8.19**	(5)	10.2 $[2.03, 0.28]$	(7)	10.0 $[1.94, 0.27]$	(7)
sra104815	117	(6)	31.0 $[8.65, 1.05]$	(9)	**25.3** $[5.86, 0.87]$	(8)
ara238025	3496	(5)	452 $[351, 31.7]$	(7)	**114** $[56.1, 5.36]$	(8)
lra498378	10758	(6)	**1952** $[332, 837]$	(10)	2118 $[230, 1001]$	(11)
lrb744710	32364	(5)	3647 $[3439, 23.9]$	(8)	**329** $[191, 5.95]$	(7)

Table 12: **Complete geometric instances via price-and-repair.**

Motivated by experiments in the previous section, we tested the B5 code with the default settings and with the option $\mu = 0.005$ (see section 4.1 for the description of this parameter; this option required double precision floating point numbers). Running times are given in table 12. The number in round parentheses gives the number of iterations of the price-and-repair method. For the B5 code we also report the time spent in the perfect matching subroutine (the first number in square brackets) and the time in the `AddNewEdge` function (the second number in square brackets).

5 Conclusions and future work

We described a new implementation of Edmonds's blossom algorithm for computing a perfect matching of minimum cost. In the majority of our experiments our Blossom V code outperformed previous implementations of Cook and Rohe [8] and of Mehlhorn and Schäfer [24] (although we were able to compare only with the original code of [24] adapted to compile with a newer LEDA library, not with the algorithm in the current LEDA version). For some classes of problems the improvement is roughly by an order of magnitude (tables 1,2,5,7).

On large VLSI instances we observed two interesting effects not exhibited on other problems: (i) the time for EXPAND operations often becomes the bottleneck, and (ii) updating the duals by solving the linear program gives a substantial speed-up compared to greedy dual updates. Future work may include changing the data structures to make EXPANDs more efficient (although this may degrade the performance on other classes of problems where EXPANDs are not a bottleneck), and speeding up the algorithm for solving the linear program for the dual updates. (At the moment we transform the problem to a minimum cost flow as described by Hochbaum [20] and then use our own implementation of a successive shortest path algorithm.)

In the future we may also try to implement a maximum cost (non-perfect) matching algorithm, using the framework of the Blossom V code.

Acnlowledgements

I thank Guido Schäfer and the support team at Algorithmic Solutions Software GmbH for answering questions about their implementation, and anonymous reviewers for comments that helped to improve the presentation of the paper.

References

[1] http://www.cs.ucl.ac.uk/staff/V.Kolmogorov/software.html (Blossom V code, version 1.0).

[2] http://www.iwr.uni-heidelberg.de/groups/comopt/software/tsplib95/tsp/.

[3] http://www.tsp.gatech.edu/vlsi/index.html.

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[5] D. Applegate and W. Cook. *Solving large-scale matching problems*, pages 557–576. Network Flows and Matchings, volume 12 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1993.

[6] M. O. Ball and U. Derigs. An analysis of alternate strategies for implementing matching algorithms. *Networks*, 13:517–549, 1983.

[7] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Computing*, 22(2):221–242, 1993.

[8] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11(2):138–148, February 1999. Computer code available at http://www2.isye.gatech.edu/˜ wcook/blossom4/.

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

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

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

[12] J. Edmonds. Path, trees, and flowers. *Canadian J. Math.*, 17:449–467, 1965.

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

[14] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[15] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[16] H. Gabow. *Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs*. PhD thesis, Stanford University, 1973.

[17] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 434–443, 1990.

[18] H. N. Gabow, Z. Galil, and T. H. Spencer. Efficient implementation of graph algorithms using contraction. *J. ACM*, 36(3):540–572, 1989.

[19] P. Gerngross. Zur implementation von edmonds' matching algorithmus: Datenstrukturen und verschiedene varianten. Diplomarbeit, Institut fur Mathematik, Universitat Augsburg, 1991.

[20] D. Hochbaum. Instant recognition of half integrality and 2-approximations. In *3rd International Workshop on Approximation Algorithms for Combinatorial Optimization*, 1998.

[21] D. S. Johnson and C. C. McGeoch. *Network Flows and Matching: First DIMACS Implementation Challenge*. American Mathematical Society, Providence, RI, 1993. Generators available at ftp://dimacs.rutgers.edu/pub/netflow/generators/matching.

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

[23] K. Mehlhorn and S. Näher. *LEDA: a platform for combinatorial and geometric computing*. Cambridge University Press, New York, NY, 1999.

[24] K. Mehlhorn and G. Schäfer. Implementation of O($nm\log n$) weighted matchings in general graphs: the power of data structures. *Journal of Experimental Algorithmics (JEA)*, 7:4, 2002.

[25] B. Moret and H. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *2nd Workshop on Algorithms and Data Structures*, pages 400–411, 1991.

[26] J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *LNCS*, pages 203–222. Springer-Verlag, 1996. Computer code available at http://www.cs.cmu.edu/quake/triangle.html.

[27] W.-K. Shih, S. Wu, and Y. S. Kuo. Unifying maximum cut and minimum cut of a planar graph. *Trans. on Computers*, 39(5):694–697, 1990.

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