# Contents



## 0.1 Maximum Cardinality Matchings in General Graphs

A *matching* $M$ in a graph $G$ is a subset of the edges no two of which share an endpoint, see Figure 0.1. The cardinality $|M|$ of a matching $M$ is the number of edges in $M$.

A node $v$ is called *matched* with respect to a matching $M$ if there is an edge in $M$ incident to $v$ and it is called *free* or *unmatched* otherwise. An edge $e$ is called matching if $e \in M$. A matching is called *perfect* if all nodes of $G$ are matched and is called *maximum* if it has maximum cardinality among all matchings.

The structure of this section is as follows. In Section 0.1.1 we discuss the functionality of our matching algorithms, in Section 0.1.2 we derive the so-called blossom shrinking algorithm for maximum matchings, and in Section 0.1.3 we give an implementation of it.

### 0.1.1 *Functionality*

The function

```
list<edge> MAX_CARD_MATCHING(const graph& G, int heur = 0)
```

returns a maximum matching in $G$. The underlying algorithm is the so-called blossom shrinking algorithm of Edmonds [Edm65b, Edm65a]. The worst case running time of the algorithm is $O(nm\alpha(m,n))$ ([Gab76]), the actual running time is usually much better. Table 0.1 contains some experimental data.

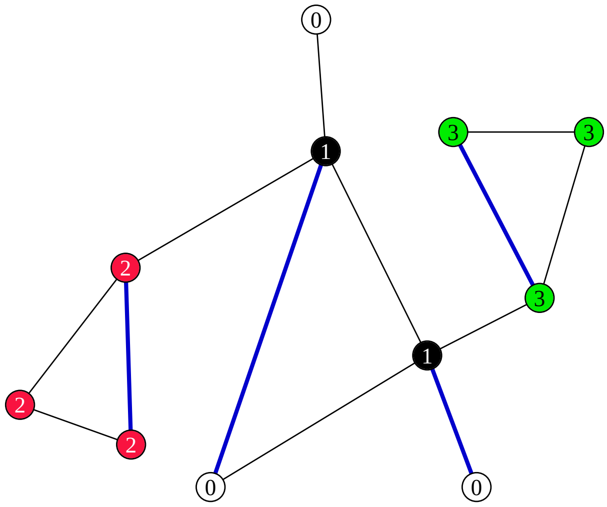


**Figure 0.1** A maximum matching and a proof of optimality: The edges of the matching are shown in bold. The node labels prove the optimality of the matching. Observe that every edge is either incident to a node labeled 1 or connects two nodes that are labeled 2 or connects two nodes that are labeled 3. There are two nodes labeled 1, three nodes labeled 2, and three nodes labeled 3. Thus no matching can have more than $2 + \lfloor 3/2 \rfloor + \lfloor 3/2 \rfloor = 4$ edges. The matching shown has four edges and is hence optimal. You may generate similar figures with the xlman-demo gw_mc_matching.

With $heur = 1$, the greedy heuristic is used to construct an initial matching which is then extended to a maximum matching by the blossom shrinking algorithm. As Table 0.1 shows, the influence of the greedy heuristic on the running time is small. It sometimes helps, it sometimes harms, and it never causes a dramatic change. The cost of checking optimality is negligible in all cases.

In the remainder of this section we discuss the check of optimality. A labeling $l$ of the nodes of $G$ with non-negative integers is said to *cover* $G$ (or to be a cover for $G$) if every edge of $G$ (which is not a self-loop) is either incident to a node labeled 1 or connects two nodes labeled with the same $i$, for some $i \geq 2$. The *capacity* of $l$ is defined as

$$cap(l) = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor,$$

where $n_i$ is the number of nodes labeled $i$. Observe that there may be nodes that are labeled zero. The capacity of a covering[1] is an upper bound on the cardinality of any matching.

**Lemma 1** *If $l$ covers $G$ and $M$ is any matching then $|M| \leq cap(l)$.*

[1] In bipartite graphs only the labels zero and one are needed. The nodes labeled one form a node cover in the sense of Section **??**.

| $n$ | $m$ | MCM | MCM+ | Check |
|---|---|---|---|---|
| 10000 | 10000 | 0.287 | 0.223 | 0.024 |
| 20000 | 20000 | 0.905 | 0.717 | 0.074 |
| 40000 | 40000 | 2.178 | 1.758 | 0.184 |
| 80000 | 80000 | 4.857 | 3.934 | 0.413 |
| 10000 | 15000 | 1.049 | 1.03 | 0.027 |
| 20000 | 30000 | 3.799 | 3.862 | 0.102 |
| 40000 | 60000 | 11.45 | 11.9 | 0.262 |
| 80000 | 120000 | 30.51 | 33.57 | 0.583 |
| 10000 | 20000 | 1.247 | 1.304 | 0.04199 |
| 20000 | 40000 | 4.876 | 5.357 | 0.136 |
| 40000 | 80000 | 14.2 | 15.3 | 0.343 |
| 80000 | 160000 | 38.42 | 43.81 | 0.789 |
| 10000 | 25000 | 1.322 | 1.347 | 0.05099 |
| 20000 | 50000 | 4.761 | 4.782 | 0.169 |
| 40000 | 100000 | 13.95 | 14.22 | 0.422 |
| 80000 | 200000 | 35.2 | 37.3 | 0.959 |

**Table 0.1** Running times of the general matching algorithm: The table shows the running time of the maximum cardinality matching algorithm without (MCM) and with the greedy heuristic (MCM+) and the time to check the result for random graphs with $n$ nodes and $m$ edges (generated by $random\_graph(G, n, m)$). In all cases the time for checking the result is negligible compared to the time for computing the maximum matching. In each of the four blocks we used $n = 2^i \cdot 10^4$ for $i = 0, 1, 2, 3$ and a fixed relationship between $n$ and $m$ ($m/n = 1, 3/2, 2, 5/2$). The time to compute the maximum matching seems approximately to triple if $n$ and $m$ are doubled. Each entry is the average of ten runs. Except on the very sparse instances ($m \approx n$) it does not pay to use the greedy heuristic.

*Proof* Since $l$ covers every edge of $G$ and hence every edge in $M$, each edge in $M$ is either incident to a node labeled one or connects two nodes labeled $i$ for some $i \geq 2$. There can be at most $n_1$ edges of the former kind and at most $\lfloor n_i/2 \rfloor$ edges of the second kind for any $i$, $i \geq 2$. Thus $|M| \leq cap(l)$. □

We will see in the next section that there is always a covering whose capacity is equal to the size of the maximum matching. The function

```
list<edge> MAX_CARD_MATCHING(const graph& G, node_array<int>& OSC,
                             int heur = 0)
```

returns a maximum matching $M$ and a labeling $OSC$ ($OSC$ stands for odd set cover, a name to be explained in the next section) with:

- $OSC$ covers $G$ and
- $|M| = cap(OSC)$.

Thus $OSC$ proves the optimality of $M$. Figure 0.1 shows an example. The additional running time for computing the proof of optimality is negligible.

The function

```
void CHECK_MAX_CARD_MATCHING(const graph& G, const list<edge>& M,
                             const node_array<int>& OSC)
```

checks whether $OSC$ is a node labeling that covers $G$ and whose capacity is equal to the cardinality of $M$. The function aborts if this is not the case. It runs in linear time.

The implementation of the checker is trivial. We determine for each $i$ the number $n_i$ of nodes with label $i$ and then compute $S = n_1 + \sum_{i\geq 2}\lfloor n_i/2 \rfloor$. We assert that $S$ is equal to the size of the matching.

We also check whether all edges are covered by the node labeling. Every edge must either be incident to a node labeled one or connect two nodes labeled $i$ for some $i \geq 2$.

$\langle$*MCM: checker*$\rangle\equiv$

```
static bool False(string s)
{ cerr << "CHECK_MAX_CARD_MATCHING: " << s << "\n";
  return false;
}

bool CHECK_MAX_CARD_MATCHING(const graph& G, const list<edge>& M,
                                    const node_array<int>& OSC)
{ int n = Max(2,G.number_of_nodes());
  int K = 1;
  array<int> count(n);
  int i;
  for (i = 0; i < n; i++) count[i] = 0;
  node v; edge e;

  forall_nodes(v,G)
  { if ( OSC[v] < 0 || OSC[v] >= n )
      return False("negative label or label larger than n - 1");
    count[OSC[v]]++;
    if (OSC[v] > K) K = OSC[v];
  }

  int S = count[1];
  for (i = 2; i <= K; i++) S += count[i]/2;
  if ( S != M.length() )
    return False("OSC does not prove optimality");

  forall_edges(e,G)
  { node v = G.source(e); node w = G.target(e);
    if ( v == w || OSC[v] == 1 || OSC[w] == 1 ||
            ( OSC[v] == OSC[w] && OSC[v] >= 2) ) continue;
```

```
      return False("OSC is not a cover");
  }
  return true;
}
```

### 0.1.2 *The Blossom Shrinking Algorithm*

We derive the *blossom shrinking* algorithm of Edmonds [Edm65b, Edm65a] for maximum cardinality matching in non-bipartite graphs. In its original form the running time of the algorithm is $O(n^4)$. Gabow [Gab76] and Lawler [Law76] improved the running time to $O(n^3)$ and Gabow [Gab76] showed how to use the partition data structure of Section **??** to obtain a running time of $O(nm\alpha(m,n))$. Tarjan [Tar83] gave a very readable presentation of Edmond's algorithm and Gabow's improvement. Our presentation and our implementation is based on [Law76] and [Tar83].

The algorithm follows the general paradigm for matching algorithms: repeated augmentation by augmenting paths until a maximum matching is obtained. We assume familiarity with the paradigm, which can, for example, be obtained by reading Section **??**. The natural way to search for an augmenting path starting in a node $v$ is to grow a so-called *alternating tree* rooted at $v$.

The root of an alternating tree is a free node, the nodes on odd levels are reached by odd length alternating paths (and hence their incoming tree edge is a non-matching edge) and the nodes on even levels are reached by even length alternating paths (and hence their incoming tree edge is a matching edge). The root is even. All leaves in an alternating tree are even and odd nodes have exactly one child (namely their mate). Figure 0.2 shows an alternating tree. A node on an even level is called an *even* node and a node on an odd level is called an *odd* node. In the implementation an even node is labeled EVEN, an odd node is labeled ODD, and every node belonging to no alternating tree carries the label UNLABELED. This suggests calling a node *labeled* if it belongs to some alternating tree and calling it *unlabeled* otherwise.

We start the algorithm by making every free node the root of a trivial alternating tree (consisting only of the free node itself) and by labeling all free nodes even. We will maintain the following invariants:

- For each free node there is an alternating tree rooted at the free node.
- All nodes belonging to one of the alternating trees are labeled EVEN or ODD. Nodes on even levels are labeled EVEN and nodes on odd levels are labeled ODD.
- All nodes belonging to no alternating tree are unlabeled (= labeled UNLABELED).

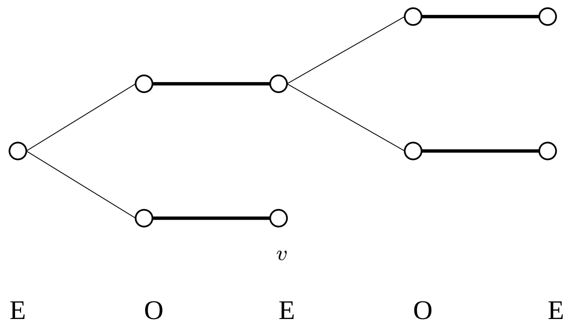


**Figure 0.2** An alternating tree: It is rooted at a free node, nodes on odd levels (= odd nodes) are reached by odd length alternating paths, and nodes on even levels (= even nodes) are reached by even length alternating paths.

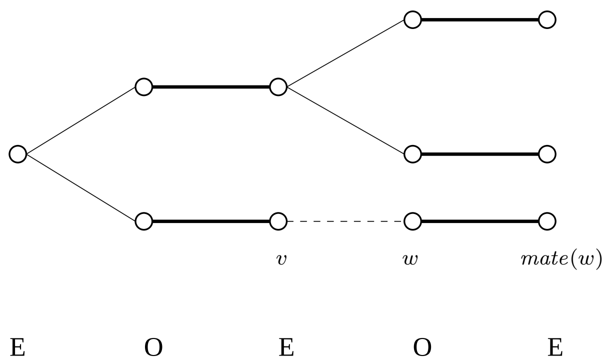


**Figure 0.3** Growing an alternating tree: Exploration of the edge $(v, w)$ turns $w$ and its mate into labeled nodes, $w$ becomes an odd node, and its mate becomes an even node.

- All unlabeled nodes are matched and if a node is unlabeled then its mate is also unlabeled.

An alternating tree is extended by exploring an edge $\{v, w\}$ incident to an even node $v$. It is a matter of implementation strategy which alternating tree is extended and which edge is chosen to extend it. There are four cases to be distinguished: $w$ may be unlabeled, $w$ may be odd, $w$ may be even and in a different tree, and $w$ may be even and in the same tree. The first three cases occur also in the bipartite case.

**Case 1, $w$ is unlabeled:** We make $w$ the child of $v$ and the mate of $w$ the child of $w$, see Figure 0.3. In this way, $w$ becomes an odd node, its mate becomes an even node, and both nodes become labeled. Observe that the growth action maintains the invariant that a matched node and its mate are either both labeled or both unlabeled.

**Case 2, $w$ is an odd node:** We have discovered another odd length alternating path to $w$ and do nothing.

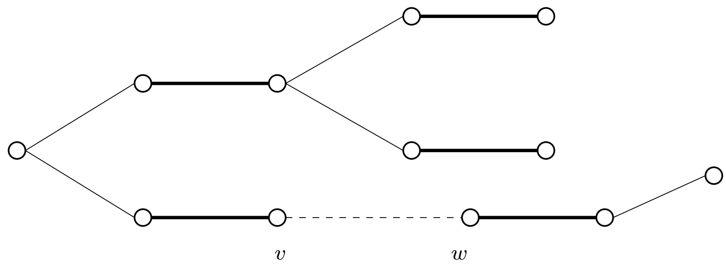


**Figure 0.4** Discovery of an augmenting path: $v$ and $w$ are even nodes in distinct trees. The edge $\{v, w\}$ and the tree paths from $v$ and $w$ to their respective roots form an augmenting path.

**Case 3, $w$ is an even node in a different tree:** We have discovered an augmenting path consisting of the edge $\{v, w\}$ and the tree paths from $v$ and $w$ to their respective roots, see Figure 0.4. We augment the matching by the augmenting path and unlabel all nodes in both trees. This makes all nodes in both trees matched (recall, that the root of an alternating tree is the only node in the tree that is unmatched) and destroys both trees. Observe that the remaining alternating trees, i.e., the ones whose roots are still free, are not affected by the augmentation. They are still augmenting trees with respect to the increased matching.

The three cases above also occur for bipartite graphs. The fourth and last case is new.

**Case 4, $w$ is an even node in the same tree as $v$:** We have discovered a so-called *blossom*, see Figure 0.5. Let $b$ be the lowest common ancestor of $v$ and $w$, i.e., $v$ and $w$ are both descendants of $b$ and there is no proper descendant of $b$ with the same property. Since only even nodes can have more than one child, $b$ is an even node. The blossom consists of the edge $\{v, w\}$ and the tree paths from $b$ to $v$ and $w$, respectively. The *stem* of the blossom consists of the tree path to $b$ and $b$ is called the *base* of the blossom. The stem is an even length alternating path ending in a matching edge; if the stem has length zero then $b$ is free. The blossom is an odd length cycle of length $2k + 1$ containing $k$ matching edges for some $k$, $k \geq 1$. All nodes in the blossom (except for the base) are reachable by an even and odd length alternating path from the root of the tree. For an even node $u$ the even length path is simply the tree path to $u$ and for an odd node $u$, say lying on the tree path from $b$ to $w$, the even length path is the tree path to $v$ followed by the edge $\{v, w\}$, followed by the path down the tree from $w$ to $u$. For the odd length paths, the situation is reversed.

The action to take is to *shrink the blossom*. To shrink a blossom means to collapse all nodes of the blossom into the base of the blossom. This removes all edges from the graph which connect two nodes in the blossom and replaces any edge $\{u, z\}$ where $u$ belongs to the blossom and $z$ does not belong to the blossom

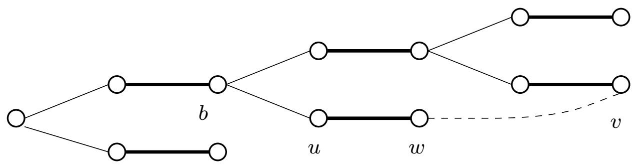


**Figure 0.5** Discovery of a blossom: $v$ and $w$ are even nodes in the same tree. The node $b$ is their lowest common ancestor. The blossom consists of the edge $\{v, w\}$ and the tree paths from $b$ to $v$ and $w$, respectively. The *stem* of the blossom consists of the tree path to $b$. The node $b$ is the base of the blossom. The blossom consists of seven edges, three of which are matching. The even length alternating path to $u$ follows the tree path to $v$, uses the edge $\{v, w\}$ and then proceeds down the tree to $u$.

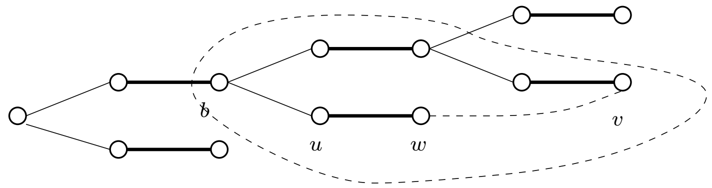


**Figure 0.6** Shrinking a blossom: All nodes of the blossom are collapsed into the base of the blossom. After the shrinking, $b$ stands for all the nodes enclosed by the dashed line.

by the edge $\{b, z\}$, see Figure 0.6. The node $b$ is free after the shrinking iff it was free before the shrinking.

**Lemma 2** *Let $G'$ be obtained from $G$ by shrinking a blossom with base $b$. If $G'$ contains an augmenting path then so does $G$.*

*Proof* Suppose $G'$ contains an augmenting path $p$. If $p$ avoids $b$ then $p$ is an augmenting path in $G$ and we are done. So let us assume that $b$ lies on $p$. We break $p$ at $b$ into two pieces $p_1$ and $p_2$ and assume w.l.o.g that $p_2$ uses a non-matching edge $e$ incident to $b$ (in $G'$). The path $p_1$ is either empty (if $b$ is free) or uses the matching edge incident to $b$. The edge $e = \{b, z\}$ in $G'$ is induced by an edge $\{u, z\}$ in $G$ where $u$ is some node of the blossom. An augmenting path in $G$ is obtained by first using $p_1$ then using the even length alternating path from $b$ to $u$ in the blossom, and then using $p_2$ (with its first edge replaced by $\{u, z\}$). □

We can now summarize the blossom shrinking algorithm. We grow alternating trees from the free nodes. Whenever a blossom is encountered it is shrunk. Whenever an augmenting path is discovered (this will in general happen after several shrinkings occurred), Lemma 2 is used to lift the augmenting path to the original graph. The matching is augmented by the augmenting path, the two trees involved are destroyed, all nodes in both trees are unlabeled, and the search for augmenting paths continues. The algorithm terminates when no alternating tree can be extended anymore. At this point the matching is maximum. Of course, this requires proof.

In order to show correctness we need the concept of an *odd-set cover*. It refines the notion of a covering introduced in Section 0.1.1.

For a subset $N$ of an odd number of vertices of $G$ we define the set of edges covered by $N$ and the capacity of $N$ as follows. If $|N| = 1$ then $N$ covers all edges incident to the node in $N$ and the capacity of $N$ is equal to one. If $|N| = 2k + 1$ for some $k \geq 1$ then $N$ covers all edges which have both endpoints in $N$ and the capacity of $N$ is $k$.

An *odd-set cover*[2] $OSC$ of $G$ is a family $\{N_1, \ldots, N_r\}$ of odd cardinality subsets of $V$ such that each edge of $G$ is covered by at least one of the sets in $OSC$. The capacity $c(OSC)$ of $OSC$ is the sum of the capacities of the sets in $OSC$.

**Lemma 3** *Let $OSC$ be an odd-set cover in a graph $G$. Then the cardinality of any matching in $G$ is at most $c(OSC)$.*

*Proof* Let $M$ be any matching and let $e$ be any edge in $M$. Then $e$ must be covered by some set in $OSC$. Moreover, the number of edges in $M$ covered by any particular set in $OSC$ is at most the capacity of the set. □

We are now ready for the correctness proof of the blossom shrinking algorithm. We will show that if the blossom shrinking algorithm does not find an augmenting path with respect to a matching $M$ then there is an odd-set cover whose capacity is equal to the size of $M$, thus proving the optimality of $M$.

Let $G^{(0)} = G$ be our graph and let $M$ be a matching in $G$. Suppose that the blossom shrinking algorithm does not discover an augmenting path. The blossom shrinking algorithm constructs a sequence $G^{(0)}$, $G^{(1)}$, $G^{(2)}$, $\ldots$, $G^{(h)}$ of graphs where for all $i$, $0 < i \leq h$, $G^{(i)}$ is obtained from $G^{(i-1)}$ by shrinking a blossom. Each node $v$ of every $G^{(i)}$ stands for a set of nodes of $G$. In $G^{(0)}$ every node represents itself, and a node $v$ in $G^{(i)}$ either stands for the same set as in $G^{(i-1)}$ or, if $v$ is equal to the base node of the shrunken blossom, stands for all nodes represented by the nodes of $G^{(i-1)}$ collapsed into it.

**Lemma 4** *For every $i$ and every node $v$ of $G^{(i)}$:*

- *$v$ stands for an odd set of nodes in $G$,*
- *if $v$ is odd or unlabeled then $v$ stands for the singleton set consisting of $v$ itself,*
- *if $v$ stands for a set $B$ of $2k + 1$ nodes in $G$ for some $k \geq 1$ then the number of edges in $M$ connecting nodes in $B$ is equal to $k$.*

[2] An odd-set cover gives rise to an integer labeling of the nodes as follows: nodes that are contained in no set of the cover are labeled zero, nodes that are contained in a singleton set are labeled one, and nodes that are contained in an odd set of cardinality larger than one are labeled $i$ for some $i > 1$. Distinct $i$'s are used for distinct sets.

*Proof* The claim is certainly true for $i$ equal to zero. When a blossom is shrunk an odd number of nodes is collapsed into a single node. By induction hypothesis each collapsed node represents an odd number of nodes of $G$. The sum of an odd number of odd numbers is odd.

The result of a shrinking operation is an even node. Thus odd and unlabeled nodes represent only themselves.

Consider a shrinking operation that collapses $2r+1$ nodes into one. Out of these nodes, $r+1$ were even before the shrinking (namely the base $v$ and every even node on the two tree paths belonging to the blossom) and $r$ were odd. Every odd node represents a single node of $G$ and every even node stands for an odd set of nodes of $G$. Suppose that the $i$-th odd node represents a set $B_i$ of $2k_i + 1$ nodes in $G$.

After the shrinking operation $v$ stands for the $r$ odd nodes and the union of the $B_i$'s. Thus $B$ consists of

$$r + \sum_{1 \le i \le r+1} (2k_i + 1) = 2(r + \sum_{1 \le i \le r+1} k_i) + 1$$

nodes and hence $k = r + \sum_{1 \le i \le r+1} k_i$. The number of edges in $M$ running between nodes of $B_i$ is $k_i$, and the number of edges of $M$ belonging to the blossom is $r$. We conclude that $k$ edges of $M$ connect nodes in $B$. □

Consider now the graph $G^{(h)}$. In $G^{(h)}$ we have an alternating tree rooted at each free node and the tree growing process has come to a halt. Thus there cannot be an edge connecting two even nodes (because this would imply the existence of either an augmenting path or a blossom) and there cannot be an edge connecting an even node to an unlabeled node (as this would allow us to grow one of the alternating trees). Thus every edge either connects two nodes contained in the same blossom, or is incident to an odd node, or connects two unlabeled nodes. Every unlabeled node is matched to an unlabeled node (since a matched node and its mate are either both unlabeled or both matched) and hence the number of unlabeled nodes is even. We construct an odd-set cover $OSC$ whose capacity is equal to $M$. $OSC$ consists of:

- all odd nodes (interpreted as singleton sets),
- for each even node that stands for a set of cardinality at least three: the set represented by the node,
- no further set if there is no unlabeled node, a singleton set consisting of an arbitrary unlabeled node if there are exactly two unlabeled nodes, and a singleton set consisting of an arbitrary unlabeled node and a set consisting of the remaining unlabeled nodes if there are more than two unlabeled nodes.

**Lemma 5** *The capacity of the odd-set cover $OSC$ is equal to the cardinality of $M$.*

*Proof* The number of edges in $M$ that still exist in $G^{(h)}$, i.e., have not been shrunken into a blossom in the course of the algorithm, is equal to the number of odd nodes plus half of the number of unlabeled nodes. For each even node $v$ of $G^{(h)}$, representing a set $B$ of $2r+1$ nodes of $G$, the number of edges in $M$ connecting nodes in $B$ is equal to $r$ by Lemma 4. This concludes the proof. □

**Theorem 1** *The blossom shrinking algorithm is correct.*

*Proof* The algorithm terminates when it does not find an augmenting path. When this happens, there is, by Lemma 5, an odd-set cover whose capacity is equal to the size of $M$. Thus $M$ is optimal. □

### 0.1.3 *The Implementation*

The goal of this section is to implement the blossom shrinking algorithm. Our implementation refines the implementation described in [Tar83] and is similar to the implementation given in [KP98]. The refinement does not change the worst case running time, but improves the best case running time from $\Omega(n^2)$ to $O(m)$. The observed behavior on random graphs with $m = O(n)$ seems to be much better than $O(n^2)$, see Table 0.1.

The overall structure of our implementation is given below. In the main loop we iterate over all nodes of $G$. Let $v_1, \ldots, v_n$ be an arbitrary ordering of the nodes of $G$. When $v = v_i$ is considered, every free node $v_j$ with $j \geq i$ is the root of a trivial alternating tree, and the collection of alternating trees rooted at free nodes $v_j$ with $j < i$ is *stable*. A collection $\mathcal{T}$ of alternating trees is stable if every edge $\{u, w\}$ incident to an even node $u$ in $\mathcal{T}$ connects $u$ to an odd node $w$ in $\mathcal{T}$. In other words, every edge $\{u, w\}$ connecting a node $u$ in $\mathcal{T}$ to a node outside $\mathcal{T}$ has $u$ odd, and every edge connecting two nodes contained in $\mathcal{T}$ has at least one odd endpoint. It follows from our tree growing rules that the trees in $\mathcal{T}$ will not change in the future.

When $v = v_i$ is considered and $v$ is already matched we do nothing. If $v$ is still unmatched we grow the alternating tree $T$ with root $v$ until either an augmenting path is found or the growth comes to an end. We use a *node_list* $Q$ to store all even nodes in $T$ which have unexplored incident edges. We organize $Q$ as a queue and hence grow the tree in breadth-first manner.

The growth process comes to an end when $Q$ becomes empty. We claim that $\mathcal{T} \cup \{\mathcal{T}\}$ is stable when $Q$ becomes empty. Consider any edge $\{u, w\}$ with $u$ an even node in $T$. Then $w$ is odd, since otherwise the growth of $T$ would not have come to an end. Moreover, $w$ belongs to a tree in $\mathcal{T} \cup \{\mathcal{T}\}$, since trees outside $\mathcal{T} \cup \{\mathcal{T}\}$ are rooted at free nodes $v_j$, $j > i$, and consist only of a root and roots are even. Thus $T$ can be added to our stable collection of alternating trees (this requires no action in the implementation) and the next free node can be considered.

When an augmenting path is found by exploring an edge $\{u, w\}$ with $u$ an even node in $T$ and $w$ an even node in a tree different from $T$, $w$ must be a free node $v_j$ with $j > i$. Observe, that $w$ cannot belong to $T$ (since $u$ and $w$ are in distinct trees) and that $w$ cannot belong to a tree in $\mathcal{T}$ (since $\mathcal{T}$ is stable). Thus $w$ must belong to a tree rooted at some $v_j$, $j > i$, and hence must be equal to some $v_j$, $j > i$ (since the trees rooted at these nodes are trivial). When the matching is augmented by the augmenting path from $v$ to $w$, all nodes in $T \cup w$ become matched and unlabeled. In order to be able to unlabel all nodes in $T \cup w$ in time proportional to the size of $T$ we collect all nodes in $T$ in a list of nodes (which we call $T$). We also set the variable *breakthrough* to *true* whenever an augmenting path is found in order to guarantee that we proceed to the next node in the main loop.

⟨*_mc_matching*⟩≡

```
enum LABEL {ODD, EVEN, UNLABELED};
⟨MCM: helpers⟩
list<edge> MAX_CARD_MATCHING(const graph& G,
                             node_array<int>& OSC, int heur)
{
  ⟨MCM: data structures⟩
  ⟨MCM: heuristics⟩
  node v; edge e;
  forall_nodes(v,G)
  { if ( mate[v] != nil ) continue;
    node_list Q; Q.append(v);
    list<node> T; T.append(v);
    bool breakthrough = false;
    while (!breakthrough && !Q.empty()) // grow tree rooted at v
    {
      node v = Q.pop();
      ⟨explore edges out of the even node v⟩
    }
  }
  list<edge> M;
  ⟨MCM: compute M⟩
  ⟨general checking: compute OSC⟩
  return M;
}
```

**The Main Data Structures:** We next discuss the main data structures used in the program. We use a *node_array*<*node*> *mate* to keep track of the current matching and we use a *node_partition base* to keep track of the blossoms.

⟨*MCM: data structures*⟩≡

```
  node_array<node> mate(G,nil);
  node_partition base(G);    // now base(v) = v for all nodes v
```

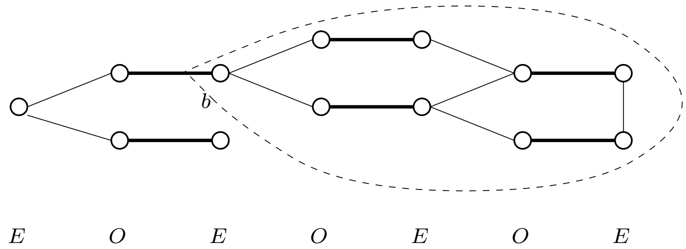


**Figure 0.7** Snapshot of the data structure: The node labels are indicated by the labels "E" and "O". All nodes enclosed by the dashed line form a blossom and hence a block of the partition *base*. The canonical element of this block is $b$.

If two nodes $v$ and $w$ are matched then $mate[v] = w$ and $mate[w] = v$ and if a node $v$ is free then $mate[v] = nil$. At the beginning, all nodes are free.

The node partition (see Section **??**) *base* establishes the relationship between the current graph $G'$ and the original graph $G$; recall that the current graph is obtained from the original graph by a sequence of shrinkings of blossoms, that a node partition partitions the nodes of a graph into disjoint sets called blocks, and that for a node $v$, $base(v)$ is the canonical representative of the block containing $v$. The relationship between $G$ and $G'$ is as follows:

- For any node $v$ of $G$: if $base(v) = v$ then $v$ is a node of $G'$ and if $base(v) \neq v$ then $v$ was collapsed into $base(v)$. Thus $\{\,base(v)\ ;\ v \in V\,\}$ is the set of nodes of $G'$.
- An edge $\{v, w\}$ represents the edge $\{\,base(v), base(w)\,\}$ of $G'$.

Every node is labeled as either EVEN, ODD, or UNLABELED. A node is labeled UNLABELED if it does not belong to any alternating tree and it is labeled EVEN or ODD otherwise. A node is labeled when it is added to an alternating tree. It retains its label when it is collapsed into another node. At the beginning all nodes are free and hence the root of an alternating tree. Thus all nodes are EVEN at the beginning. For an odd node $v$ we use $pred[v]$ to store its parent node in the alternating tree. The pred value is set when a node is added to an alternating tree; it is not changed when the node is collapsed into another node.

$\langle$*MCM: data structures*$\rangle{+}{\equiv}$

```
node_array<int>  label(G,EVEN);
node_array<node> pred(G,nil);
```

Figure 0.7 shows an example.

**Exploring an Edge:** Having defined most of the data structures we can give the details of exploring edges. Assume that $v$ is an even node and let $e = \{v, w\}$ be an edge incident to $v$. Recall that $e$ stands for the edge $\{base(v), base(w)\}$ in the current graph.

We do nothing if $e$ is a self-loop or if $base(w)$ is ODD. If $base(w)$ is UNLABELED (this is equivalent to $w$ being unlabeled) we grow the alternating tree containing $v$ and if $base(w)$ is EVEN we have either discovered an augmenting path or a blossom.

⟨*explore edges out of the even node v*⟩≡

```
forall_inout_edges(e,v)
{ node w = G.opposite(v,e);
  if ( base(v) == base(w) || label[base(w)] == ODD )
     continue;   // do nothing
  if ( label[w] == UNLABELED )
    { ⟨grow tree⟩ }
  else  // base(w) is EVEN
    { ⟨augment or shrink blossom⟩ }
}
```

**Growing the Tree:** Let us first give the details of growing a tree. We label $w$ as odd, make $v$ the parent of $w$, label the mate of $w$ as even, add the mate of $w$ to $Q$, and add $w$ and the mate of $w$ to $T$.

⟨*grow tree*⟩≡

```
label[w] = ODD;             T.append(w);
pred[w] = v;
label[mate[w]] = EVEN;      T.append(mate[w]);
Q.append(mate[w]);
```

**Discovery of a Blossom or an Augmenting Path:** The node $base(w)$ is even. We have either found an augmenting path or a blossom. We have found an augmenting path if $base(v)$ and $base(w)$ belong to distinct trees and we have discovered a blossom if they belong to the same tree. We distinguish the two cases by tracing both tree paths in lock-step fashion until we either encounter a node that lies on both paths or reach both roots[3].

We discover a node lying on both paths as follows. We keep a counter *strue* which we increment in every execution of ⟨*augment or shrink blossom*⟩. Since there are at most $n$ augmentations and at most $n$ shrinkings between two augmentations the maximal value of the counter is bounded by $n^2$. It would therefore be unsafe to use type *int* for the counter, but type *double* is safe.

We use the counter as follows. As we trace the two tree paths we set *path1* $[hv]$

[3] An alternative strategy is as follows: we have found an augmenting path if $w$ is the root of a tree outside $\mathcal{T} \cup \{\mathcal{T}\}$. We could, for each node, keep a bit to record this fact. The alternative simplifies the distinction between blossom shrinking and augmentations. However, it does not simplify the code overall, as all the information gathered in the program chunk ⟨*augment or shrink blossom*⟩ is needed in later steps of the algorithm.

to *strue* for all even nodes *hv* on the first path and *path2*[*hw*] to *strue* for all even nodes *hw* on the second path. The two paths meet iff *path1*[*hw*] or *path2*[*hv*] is equal to *strue* for some even *hw* on the second path or some even *hv* on the first path. The first node for which this is true is the base of the blossom. Recall that the base of a blossom is always even.

The cost of tracing the paths is proportional to the size of the blossom found, if a blossom is discovered, and is proportional to the length of the augmenting path found otherwise. Also observe that we define the arrays *path1* and *path2* outside the loop that searches for augmenting paths. Thus the cost for their initialization arises only once.

⟨*MCM: data structures*⟩+≡

```
double strue = 0;
node_array<double>  path1(G,0);
node_array<double>  path2(G,0);
```

⟨*augment or shrink blossom*⟩≡

```
node hv = base(v);
node hw = base(w);
strue++;
path1[hv] = path2[hw] = strue;
while ((path1[hw] != strue && path2[hv] != strue) &&
       (mate[hv] != nil || mate[hw] != nil) )
{ if (mate[hv] != nil)
  { hv = base(pred[mate[hv]]);
    path1[hv] = strue;
  }
  if (mate[hw] != nil)
  { hw = base(pred[mate[hw]]);
    path2[hw] = strue;
  }
}
if (path1[hw] == strue || path2[hv] == strue)
  { ⟨shrink blossom⟩ }
else
  { ⟨augment path⟩ }
```

**Shrinking a Blossom:** Let us see how to shrink a blossom. The base $b$ of the blossom[4] is either *hv* or *hw*. It is *hw* if *hw* also lies on the first path and it is *hv* otherwise. We shrink the blossom by shrinking the two paths that form the blossom.

The call *shrink_path*$(b, v, w, \ldots)$ collapses the path from $v$ to $b$ into $b$ and the call

[4] With the alternative case distinction between blossom shrinking and augmentation we would have to compute *hv* and *hw* at this point.

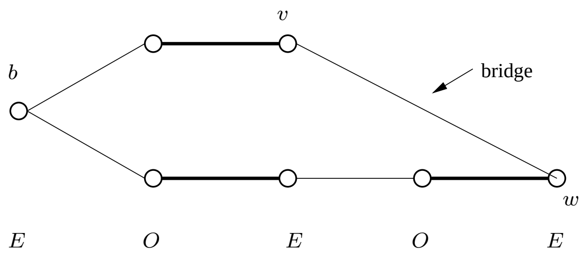


**Figure 0.8** The bridge of a blossom: The edge $\{v, w\}$ closes a blossom with base $b$. For the odd nodes on the tree path from $b$ to $v$ we set *source_bridge* to $v$ and *target_bridge* to $w$ and for the odd nodes on the tree path from $b$ to $w$ we set *source_bridge* to $w$ and *target_bridge* to $v$.

*shrink_path*$(b, w, v, \ldots)$ collapses the path from $w$ to $b$ into $b$. Both calls also have the other end of the edge that closes the blossom as an argument.

⟨*shrink blossom*⟩≡

```
node b = (path1[hw] == strue) ? hw : hv;    // Base
shrink_path(b,v,w,base,mate,pred,source_bridge,target_bridge,Q);
shrink_path(b,w,v,base,mate,pred,source_bridge,target_bridge,Q);
```

Before we can give the details of the procedure *shrink_path* we need to introduce two more node labels. When an edge $\{v, w\}$ closes a blossom, all odd nodes in the blossom also get an even length alternating path to the root of their alternating tree. This path goes through the edge that closes the blossom. We call this edge the *bridge* of the blossom. The odd nodes on the tree path from $v$ to $b$ use the bridge in the direction from $v$ to $w$ and the odd nodes on the tree path from $w$ to $b$ use the bridge in the direction from $w$ to $v$. We use the node arrays *source_bridge* and *target_bridge* to record for each odd node shrunken into a blossom the source node and the target node of its bridge (now viewed as a directed edge).

⟨*MCM: data structures*⟩+≡

```
node_array<node> source_bridge(G,nil);
node_array<node> target_bridge(G,nil);
```

The details of collapsing the tree path from $v$ to $b$ into $b$ are now simple. For each node $x$ on the path we perform *union_blocks*$(x, b)$ to union the blocks containing $x$ and $b$, for each odd node we set *source_bridge* to $v$ and *target_bridge* to $w$, and we add all odd nodes to $Q$ (because the edges out of the odd nodes now emanate from the even node $b$), see Figure 0.8.

There is one subtle point. After a union operation the canonical element of the newly formed block is unspecified (it may be any element of the resulting block). It is important, however, that $b$ stays the canonical element of the block containing it. We therefore explicitly make $b$ the canonical element by *base.make_rep*$(b)$.

⟨*MCM: helpers*⟩≡

```
static void shrink_path(node b, node v, node w,
          node_partition& base, node_array<node>& mate,
          node_array<node>& pred, node_array<node>& source_bridge,
          node_array<node>& target_bridge, node_list& Q)
{ node x = base(v);
  while (x != b)
  {
    base.union_blocks(x,b);
    x = mate[x];
    base.union_blocks(x,b);
    base.make_rep(b);
    Q.append(x);
    source_bridge[x] = v;  target_bridge[x] = w;
    x = base(pred[x]);
  }
}
```

**Augmentation:** We treat the discovery of an augmenting path. The nodes $v$ and $w$ belong to distinct alternating trees with roots $hv$ and $hw$, respectively. In fact, $w$ is a root itself. The augmenting path consists of the edge $\{w, v\}$ plus the even length alternating path from $v$ to its root $hv$.

For a node $v$ let $p(v)$ be the even length alternating path from $v$ to its root (if it exists). The path $p(v)$ can be defined inductively as follows:

If $v$ is a root then $p(v)$ is the trivial path consisting solely of $v$.

If $v$ is EVEN, $p(v)$ goes through the mate of $v$ to the predecessor of the mate and then follows $p(pred[mate[v]])$.

If $v$ is ODD, $p(v)$ consists of the alternating path from $v$ to $source\_bridge[v]$ concatenated with $p(target\_bridge[v])$.

**Lemma 6** *The above characterization of* $p(v)$ *is correct.*

*Proof* The claim is certainly true when $v$ is a root. So assume otherwise and consider the time when $p(v)$ is discovered in the course of the algorithm. For an even node this is the time when $v$ is labeled EVEN and for an odd node this is the case when it becomes part of a blossom. In either case the characterization is correct. □

How can we find the alternating path from $v$ to $source\_bridge[v]$ when $v$ is odd? The problem is that the *pred*-pointers are directed towards the roots of alternating trees and hence there is no direct way to walk from $v$ to $source\_bridge[v]$. We walk from $source\_bridge[v]$ to $v$ instead and then take the reversal of the resulting path. The path from $source\_bridge[v]$ to $v$ is the prefix of $p(source\_bridge[v])$ ending in $v$, see Figure 0.9.

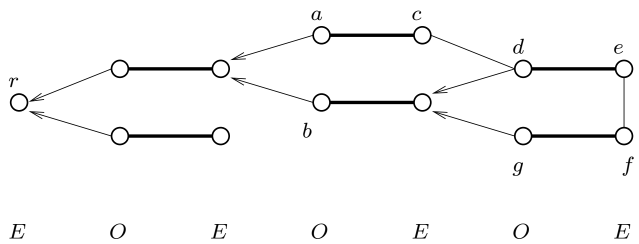


**Figure 0.9** Tracing augmenting paths: The node labels are indicated by the labels “E” and “O”. The predecessor pointer of the odd nodes are shown. When the bridge $\{e, f\}$ was explored we set *source_bridge*$[d]$ to $e$, *target_bridge*$[d]$ to $f$, *source_bridge*$[g]$ to $f$, and *target_bridge*$[g]$ to $e$, and when the bridge $\{c, d\}$ was explored we set *source_bridge*$[a]$ to $c$, *target_bridge*$[a]$ to $d$, *source_bridge*$[b]$ to $d$, and *target_bridge*$[b]$ to $c$. The even length alternating path from $b$ to its root $r$ consists of the reversal of the path from $d =$ *source_bridge*$[b]$ to $b$ followed by the even length alternating path from $c =$ *target_bridge*$[b]$ to $r$. The former path consists of the reversal of the alternating path from $e =$ *source_bridge*$[d]$ to $d$ followed by the alternating path from $f =$ *target_bridge*$[d]$ to $b$.

We cast this reasoning into a program by defining a procedure *find_path*$(P, x, y, \ldots)$ that takes two nodes $x$ and $y$, such that $y$ lies on $p(x)$ and such that the prefix of $p(x)$ ending in $y$ has even length (the program would be slightly less elegant without the second assumption), and appends the prefix of $p(x)$ ending in $y$ to the list $P$. *Find_path* distinguishes three cases:

If $x$ is equal to $y$ then the path consists of the single node $x$.

If $x \neq y$ and $x$ is EVEN the path consists of $x$, *mate*$[x]$, followed by the path from *pred*$[$*mate*$[x]]$ to $y$.

If $x \neq y$ and $x$ is ODD, let *P1* and *P2* be the paths from *target_bridge*$[x]$ to $y$ and from *source_bridge*$[x]$ to *mate*$[x]$, respectively. Then path consists of $x$ followed by the reversal of *P2* followed by *P1*.

⟨*MCM: helpers*⟩+≡

```
static void find_path(list<node>& P, node x, node y,
                node_array<int>&  label, node_array<node>& pred,
                node_array<node>& mate,
                node_array<node>& source_bridge,
                node_array<node>& target_bridge)
{ if ( x == y )
  {
    P.append(x);
    return;
  }
  if ( label[x] == EVEN )
  {
    P.append(x);
    P.append(mate[x]);
    find_path(P,pred[mate[x]],y,label,pred,mate,
                   source_bridge,target_bridge);
```

```
      return;
    }
    else // x is ODD
    {
      P.append(x);
      list<node> P2;
      find_path(P2,source_bridge[x],mate[x],label,pred,mate,
                              source_bridge,target_bridge);
      P2.reverse_items();
      P.conc(P2);
      find_path(P,target_bridge[x],y,label,pred,mate,
                       source_bridge,target_bridge);
      return;
    }
  }
```

Given *find_path*, it is trivial to construct the augmenting path. We construct the path from $v$ to $hv$ in $P$ and append $w$ to the front of the path. We augment the current matching by the path by walking along the path and changing *mate* accordingly.

It remains to prepare for the next search for an augmenting path. All nodes in $T \cup \{w\}$ are now matched. We unlabel all nodes in $T \cup \{w\}$ and split the blocks of *base* containing nodes of $T$. No action is required for the other alternating trees.

Finally, we set *breakthrough* to *true* and break from the forall-inout-edges loop. Setting *breakthrough* to *true* makes sure that we also leave the grow tree loop. The next action will therefore be to grow an alternating tree from the next free node.

$\langle$*augment path*$\rangle \equiv$

```
list<node> P;
find_path(P,v,hv,label,pred,mate,source_bridge,target_bridge);
P.push(w);
while(! P.empty())
{ node a = P.pop();
  node b = P.pop();
  mate[a] = b;
  mate[b] = a;
}
T.append(w);
forall(v,T) label[v] = UNLABELED;
base.split(T);
breakthrough = true;
break;
```

**Computing the Node Labeling** $OSC$**:** We compute the node labeling $OSC$ as described in the paragraph preceding Lemma 5. We initialize $OSC[v]$ to $-1$ for all nodes $v$. This will allow us to recognize nodes without a proper $OSC$-label later.

We then determine the number of unlabeled nodes (= nodes labeled *UNLABELED* and select an arbitrary unlabeled node. If there are unlabeled nodes, the selected unlabeled node is labeled one and all other unlabeled nodes are either labeled zero (if there are exactly two unlabeled nodes) or two (if there are more than two unlabeled nodes). We then set $K$ to the smallest unused label larger than one.

Next we determine the number of sets of cardinality at least three and assign distinct labels to their representatives. We do so by iterating over all nodes. Every node $v$ with $base(v) \neq v$ indicates a set of cardinality at least three. If its base is still unlabeled, we label it.

Finally, we label all other nodes. Nodes belonging to a set of cardinality at least two inherit the label of the base, and nodes that belong to sets of cardinality one (they satisfy `base(v) == v && OSC[base(v)] == -1`) are labeled one iff they are ODD and are labeled zero if they are EVEN.

⟨*general checking: compute OSC*⟩≡

```
forall_nodes(v,G) OSC[v] = -1;
int number_of_unlabeled = 0;
node arb_u_node;
forall_nodes(v,G)
 if ( label[v] == UNLABELED )
 { number_of_unlabeled++;
   arb_u_node = v;
 }
if ( number_of_unlabeled > 0 )
{ OSC[arb_u_node] = 1;
  int L = ( number_of_unlabeled == 2 ? 0 : 2 );
  forall_nodes(v,G)
    if ( label[v] == UNLABELED && v != arb_u_node ) OSC[v] = L;
}
int K = ( number_of_unlabeled <= 2 ? 2 : 3);
forall_nodes(v,G)
 if ( base(v) != v && OSC[base(v)] == -1 ) OSC[base(v)] = K++;
forall_nodes(v,G)
{ if ( base(v) == v && OSC[v] == -1 )
  { if ( label[v] == EVEN ) OSC[v] = 0;
    if ( label[v] == ODD  ) OSC[v] = 1;
  }
  if ( base(v) != v ) OSC[v] = OSC[base(v)];
}
```

**Computing the List of Matching Edges:** The list $M$ of matching edges is readily constructed. We iterate over all edges. Whenever an edge is encountered whose endpoints are matched with each other, the edge is added to the matching. We also "unmate" the endpoints in order to avoid adding parallel edges to $M$.

⟨*MCM: compute M*⟩≡

```
forall_edges(e,G)
{ node v = source(e);
  node w = target(e);
  if ( v != w  &&  mate[v] == w )
  { M.append(e);
    mate[v] = v;
    mate[w] = w;
   }
 }
```

**Heuristics:** If $heur = 1$, the greedy heuristic is used to compute an initial matching. We iterate over all edges. If both endpoints of an edge are unmatched, we match the endpoints and declare both endpoints unlabeled. Recall that matched nodes that do not belong to an alternating tree are UNLABELED.

⟨*MCM: heuristics*⟩≡

```
switch (heur) {
case 0: break;
case 1: { edge e;
          forall_edges(e,G)
          { node v = G.source(e); node w = G.target(e);
            if ( v != w && mate[v] == nil && mate[w] == nil)
            { mate[v] = w; label[v] = UNLABELED;
              mate[w] = v; label[w] = UNLABELED;
            }
          }
          break;
        }
}
```

**Summary:** We summarize and complete the running time analysis. The algorithm computes a maximum matching in phases. In each phase an alternating tree $T$ from a free node is grown to find an augmenting path. If the search for an augmenting path is successful, the matching is increased and all nodes in the alternating tree are unlabeled, and if the search is unsuccessful, the tree will stay around and will never be looked at again.

The running time of a phase is $O((n_T + m_T)\alpha(n_T, m_T))$, where $n_T$ is the number of nodes included into $T$, $m_T$ is the number of edges having at least one endpoint in $T$, and $\alpha(n, m_T)$ is the cost of $m_T$ operations on a node partition of $n$ nodes. This can be seen as follows. In a phase zero or more blossoms are shrunken. The search for a blossom (if successful) has cost proportional to the size of the blossom, and shrinking a blossom of size $2k+1$ removes $2k$ nodes from the graph. Therefore the total size of all blossoms shrunk in a phase is $O(n_T)$. In each phase each edge is explored at most twice (once from each endpoint). Each exploration of an edge and each removal of a node involves a constant number of operations on the node partition *base*. We conclude that the total cost of a phase is $O((n_T + m_T)\alpha(n, m_T)) = O((n+m)\alpha(n,m)) = O(m\alpha(n,m))$, since $n_T \le n \le m$ and $m_T \le m$.

There are at most $n$ phases and hence the total running time is $O(nm\alpha(n,m))$ in the worst case. One may hope that $n_T$ is significantly smaller than $n$ and $m_T$ is significantly smaller than $m$ for many phases. The running times reported in Section 0.1.1 show that the hope is justified in the case of random graphs. There are no analytical results concerning the average case behavior of general matching algorithms.

In an earlier implementation of the blossom shrinking algorithm we did not collect the nodes of the alternating tree grown into a set $T$. Rather, we iterated over all nodes at the beginning of a phase and labeled all free nodes EVEN and all matched nodes UNLABELED. With this implementation the running time is $\Omega(n^2)$. The implementation discussed in this section is significantly faster. It is superior for two reasons. Firstly, the cost of a phase is proportional to the size of the alternating tree grown in the phase and hence may be sublinear, and secondly, an alternating tree that does not lead to a breakthrough is not destroyed, but kept till the end of the execution.

***Exercises for 0.1***

1 Compare the running time of the general matching algorithm and the bipartite matching algorithm on bipartite graphs.

2 Exhibit a family of graphs where the running time of our matching algorithm is $\Omega(nm)$. Write a program to generate such graphs and provide it as an LEP.

# Bibliography

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

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

[Gab76] H. N. Gabow. An efficient implementation of Edmond's algorithm for maximum matching on graphs. *Journal of the ACM*, 23:221–234, 1976.

[KP98] J.D. Kececioglu and J. Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Proceedings of the 2nd Workshop on Algorithm Engineering (WAE'98)*, pages 121–132. Max-PlanckInstitut für Informatik, 1998.

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

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, 1983.