Blossom algorithm

From Wikipedia, the free encyclopedia

The **blossom algorithm** is an algorithm in graph theory for constructing maximum matchings on graphs. The algorithm was developed by Jack Edmonds in 1961,[1] and published in 1965.[2] Given a general graph $G = (V, E)$, the algorithm finds a matching M such that each vertex in V is incident with at most one edge in M and $|M|$ is maximized. The matching is constructed by iteratively improving an initial empty matching along augmenting paths in the graph. Unlike bipartite matching, the key new idea is that an odd-length cycle in the graph (blossom) is contracted to a single vertex, with the search continuing iteratively in the contracted graph.

A major reason that the blossom algorithm is important is that it gave the first proof that a maximum-size matching could be found using a polynomial amount of computation time. Another reason is that it led to a linear programming polyhedral description of the matching polytope, yielding an algorithm for min-*weight* matching.[3] As elaborated by Alexander Schrijver, further significance of the result comes from the fact that this was the first polytope whose proof of integrality "does not simply follow just from total unimodularity, and its description was a breakthrough in polyhedral combinatorics."[4]

Contents

Augmenting paths

Given $G = (V, E)$ and a matching M of G, a vertex v is **exposed** if no edge of M is incident with v. A path in G is an **alternating path**, if its edges are alternately not in M and in M (or in M and not in M). An **augmenting path** P is an alternating path that starts and ends at two distinct exposed vertices. A **matching augmentation** along an augmenting path P is the operation of replacing M with a new matching $M_1 = M \oplus P = (M \setminus P) \cup (P \setminus M)$.
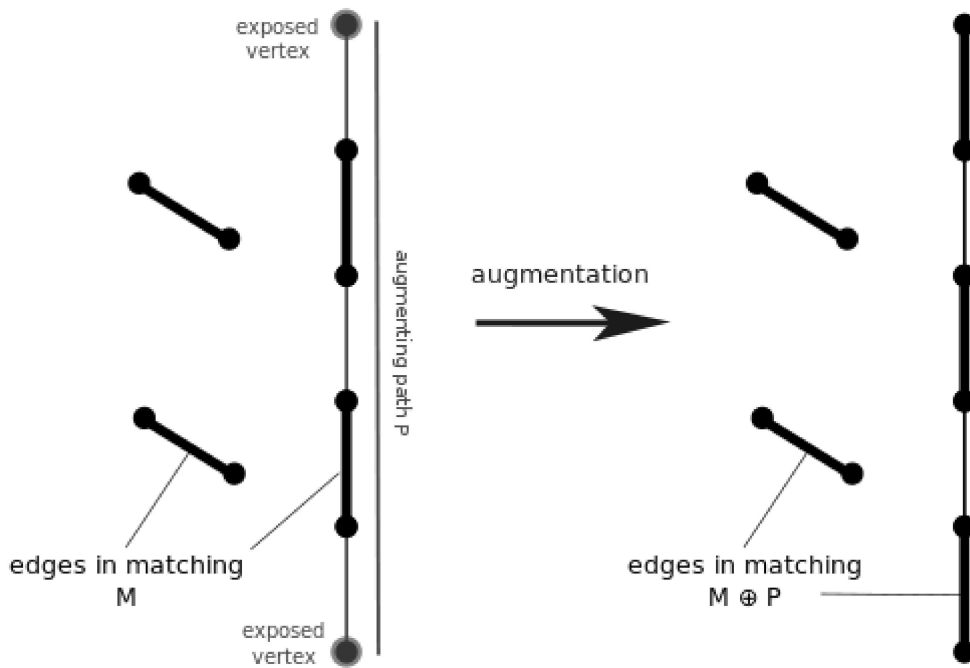
By Berge's lemma, matching M is maximum if and only if there is no M-augmenting path in G.[5][6] Hence, either a matching is maximum, or it can be augmented. Thus, starting from an initial matching, we can compute a maximum matching by augmenting the current matching with augmenting paths as long as we can find them, and return whenever no augmenting paths are left. We can formalize the algorithm as follows:

```
   INPUT:  Graph G, initial matching M on G
   OUTPUT: maximum matching M* on G
A1 function find_maximum_matching( G, M ) : M*
A2     P ← find_augmenting_path( G, M )
A3     if P is non-empty then
A4         return find_maximum_matching( G, augment M along P )
A5     else
A6         return M
A7     end if
A8 end function
```

We still have to describe how augmenting paths can be found efficiently. The subroutine to find them uses blossoms and contractions.
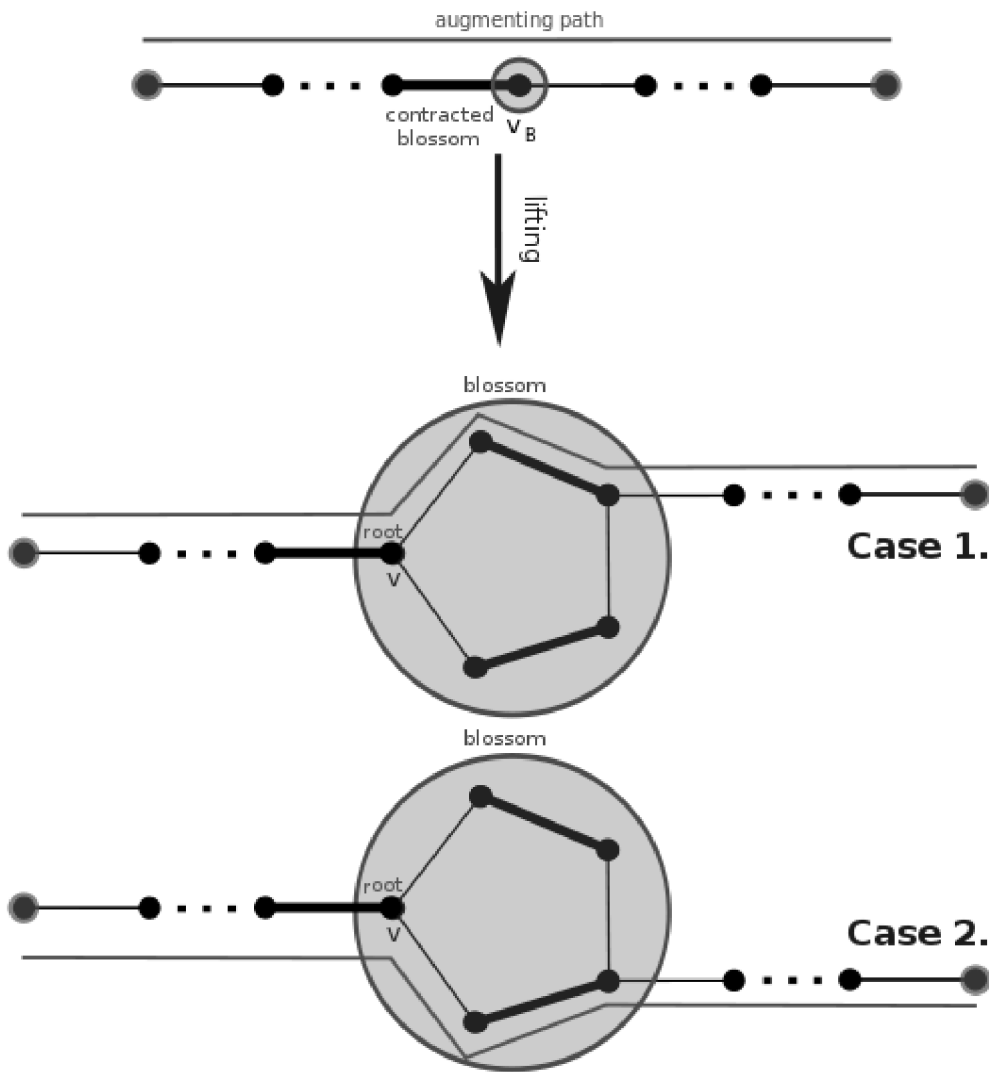
Blossoms and contractions

Given $G = (V, E)$ and a matching M of G, a *blossom B* is a cycle in G consisting of *2k + 1* edges of which exactly k belong to M, and where one of the vertices v of the cycle (the *base*) is such that there exists an alternating path of even length (the *stem*) from v to an exposed vertex w.

Define the **contracted graph** *G'* as the graph obtained from G by contracting every edge of B, and define the **contracted matching** *M'* as the matching of *G'* corresponding to M.
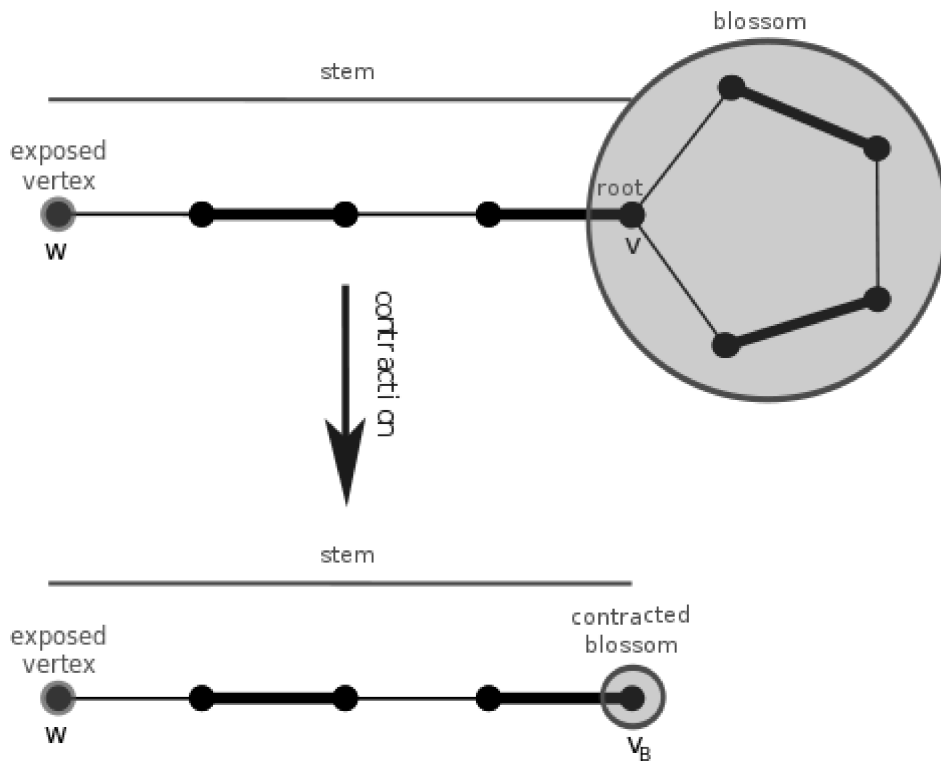
G' has an M'-augmenting path iff G has an M-augmenting path, and that any M'-augmenting path P' in G' can be **lifted** to an M-augmenting path in G by undoing the contraction by B so that the segment of P' (if any) traversing through v_B is replaced by an appropriate segment traversing through B.[7] In more detail:

- if P' traverses through a segment $u \to v_B \to w$ in G', then this segment is replaced with the segment $u \to (u' \to \dots \to w') \to w$ in G, where blossom vertices u' and w' and the side of B, $(u' \to \dots \to w')$, going from u' to w' are chosen to ensure that the new path is still alternating (u' is exposed with respect to $M \cap B$, $\{w', w\} \in E \setminus M$).

augmenting path

contracted blossom v_B

lifting

blossom

root

v

Case 1.

blossom

root

v

Case 2.

- if P' has an endpoint v_B, then the path segment $u \rightarrow v_B$ in G' is replaced with the segment $u \rightarrow (u' \rightarrow \ldots \rightarrow v')$ in G, where blossom vertices u' and v' and the side of B, $(u' \rightarrow \ldots \rightarrow v')$, going from u' to v' are chosen to ensure that the path is alternating (v' is exposed, $\{u', u\} \in E \setminus M$).
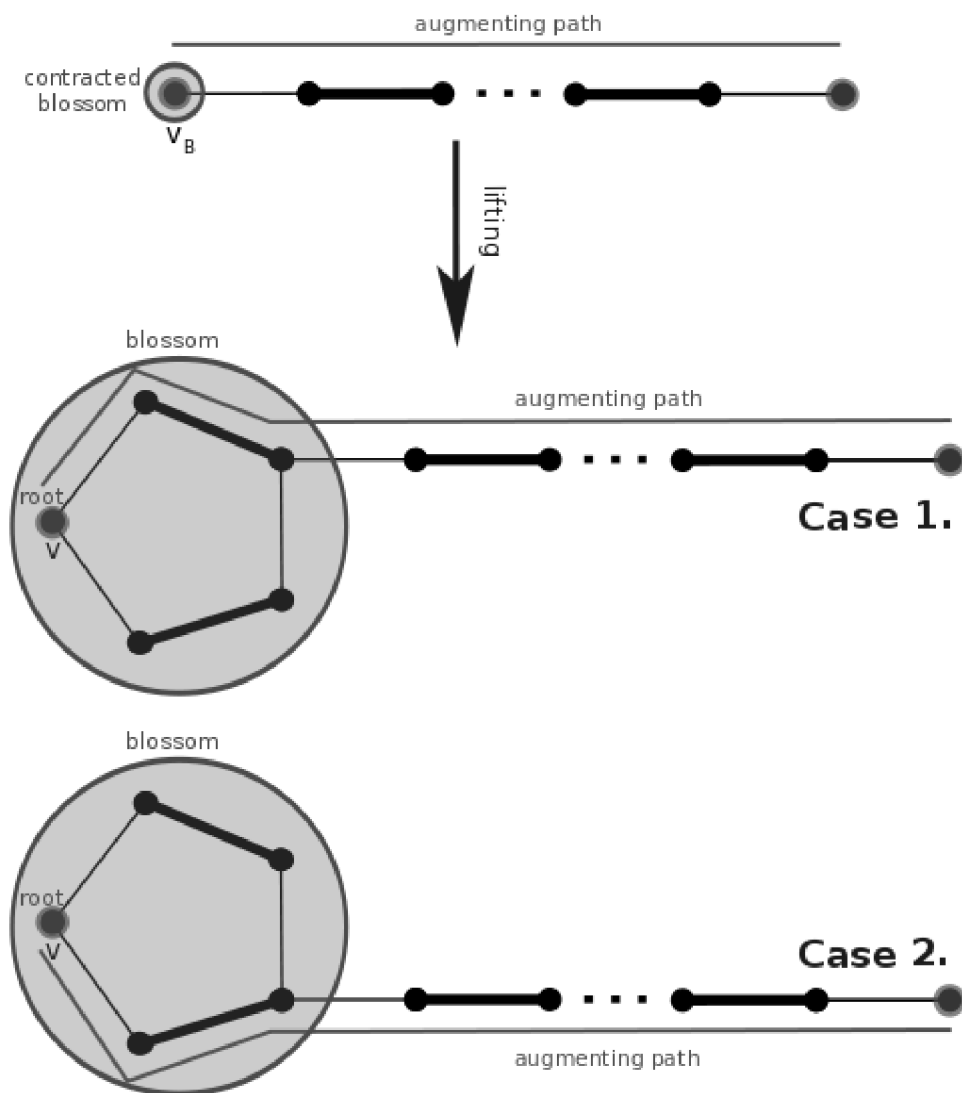
Thus blossoms can be contracted and search performed in the contracted graphs. This reduction is at the heart of Edmonds' algorithm.

Finding an augmenting path

The search for an augmenting path uses an auxiliary data structure consisting of a forest F whose individual trees correspond to specific portions of the graph G. In fact, the forest F is the same that would be used to find maximum matchings in bipartite graphs (without need for shrinking blossoms). In each iteration the algorithm either (1) finds an augmenting path, (2) finds a blossom and recurses onto the corresponding contracted graph, or (3) concludes there are no augmenting paths. The auxiliary structure is built by an incremental procedure discussed next.[7]

The construction procedure considers vertices v and edges e in G and incrementally updates F as appropriate. If v is in a tree T of the forest, we let *root(v)* denote the root of T. If both u and v are in the same tree T in F, we let *distance(u,v)* denote the length of the unique path from u to v in T.

```
    INPUT:  Graph G, matching M on G
    OUTPUT: augmenting path P in G or empty path if none found
B01 function find_augmenting_path( G, M ) : P
B02     F ← empty forest
B03     unmark all vertices and edges in G, mark all edges of M
B05     for each exposed vertex v do
B06         create a singleton tree { v } and add the tree to F
B07     end for
B08     while there is an unmarked vertex v in F with distance( v, root( v ) ) even do
B09         while there exists an unmarked edge e = { v, w } do
```

```
B10              if w is not in F then
                     // w is matched, so add e and w's matched edge to F
B11                  x ← vertex matched to w in M
B12                  add edges { v, w } and { w, x } to the tree of v
B13              else
B14                  if distance( w, root( w ) ) is odd then
                         // Do nothing.
B15                  else
B16                      if root( v ) ≠ root( w ) then
                             // Report an augmenting path in F ∪ { e }.
B17                          P ← path ( root( v ) → ... → v ) → ( w → ... → root( w ) )
B18                          return P
B19                      else
                             // Contract a blossom in G and look for the path in the contracted graph.
B20                          B ← blossom formed by e and edges on the path v → w in T
B21                          G', M' ← contract G and M by B
B22                          P' ← find_augmenting_path( G', M' )
B23                          P ← lift P' to G
B24                          return P
B25                      end if
B26                  end if
B27              end if
B28              mark edge e
B29          end while
B30          mark vertex v
B31      end while
B32      return empty path
B33 end function
```
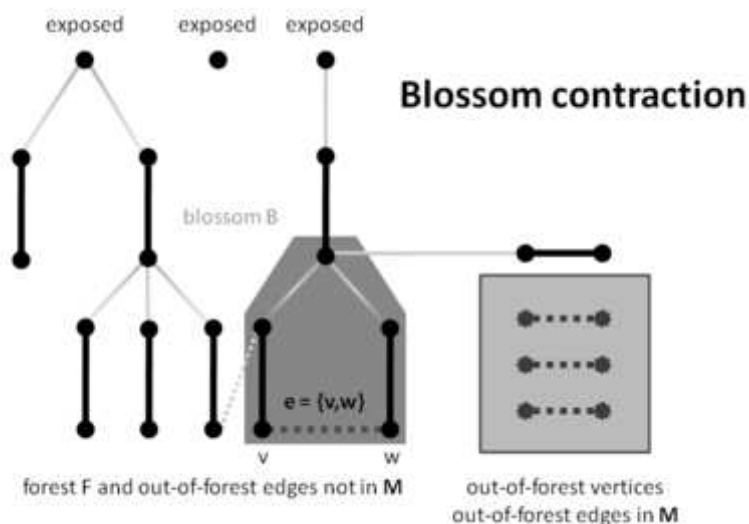
Examples

The following four figures illustrate the execution of the algorithm. Dashed lines indicate edges that are currently not present in the forest. First, the algorithm processes an out-of-forest edge that causes the expansion of the current forest (lines B10 – B12).
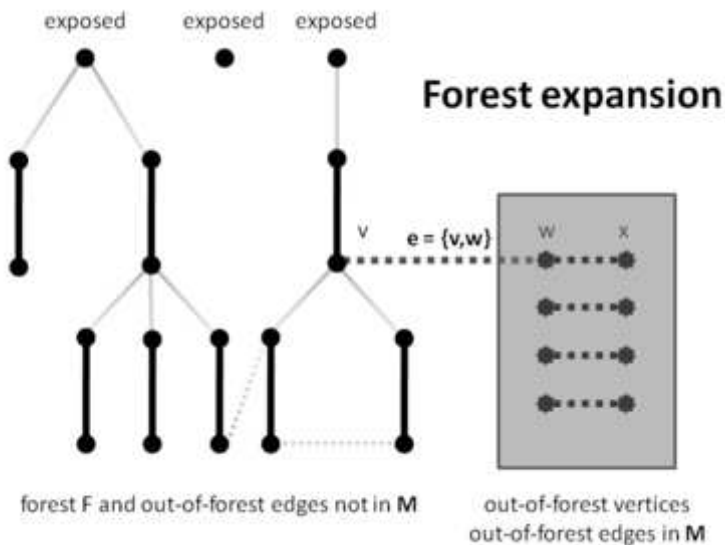
Next, it detects a blossom and contracts the graph (lines B20 – B21).

Blossom contraction

Finally, it locates an augmenting path P′ in the contracted graph (line B22) and lifts it to the original graph (line B23). Note that the ability of the algorithm to contract blossoms is crucial here; the algorithm cannot find *P* in the original graph directly because only out-of-forest edges between vertices at even distances from the roots are considered on line B17 of the algorithm.

Path detection in G′

Path lifting
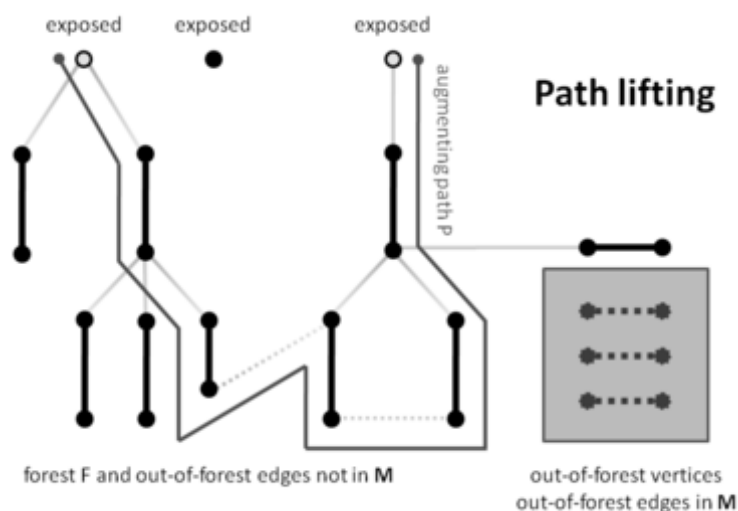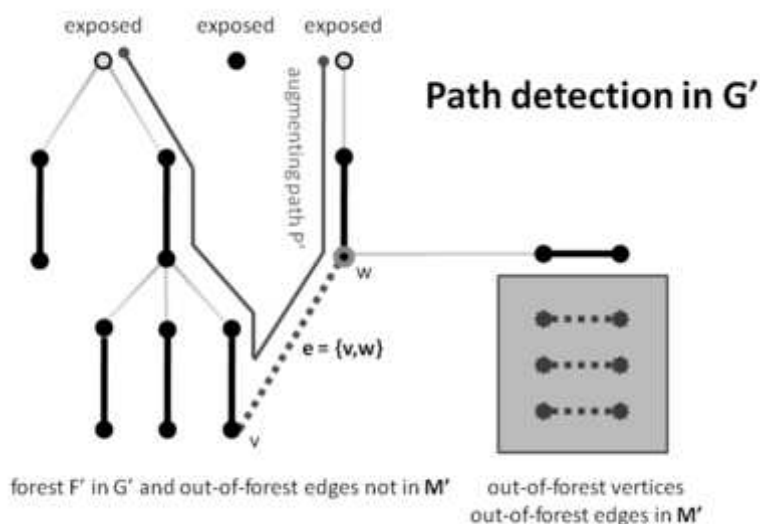
Analysis

The forest *F* constructed by the *find_augmenting_path()* function is an alternating forest.[8]

- a tree *T* in *G* is an **alternating tree** with respect to *M*, if
  - *T* contains exactly one exposed vertex *r* called the tree root
  - every vertex at an odd distance from the root has exactly two incident edges in *T*, and
  - all paths from *r* to leaves in *T* have even lengths, their odd edges are not in *M* and their even edges are in *M*.
- a forest *F* in *G* is an **alternating forest** with respect to *M*, if
  - its connected components are alternating trees, and
  - every exposed vertex in *G* is a root of an alternating tree in *F*.

Each iteration of the loop starting at line B09 either adds to a tree *T* in *F* (line B10) or finds an augmenting path (line B17) or finds a blossom (line B20). It is easy to see that the running time is $O(|V|^4)$. Micali and Vazirani[9] show an algorithm that constructs maximum matching in $O(|E||V|^{1/2})$ time.

Bipartite matching

The algorithm reduces to the standard algorithm for matching in bipartite graphs[6] when *G* is bipartite. As there are no odd cycles in *G* in that case, blossoms will never be found and one can simply remove lines B20 – B24 of the algorithm.

Weighted matching

The matching problem can be generalized by assigning weights to edges in *G* and asking for a set *M* that produces a matching of maximum (minimum) total weight. The weighted matching problem can be solved by a combinatorial algorithm that uses the unweighted Edmonds's algorithm as a subroutine.[5] Kolmogorov provides an efficient C++ implementation of this.[10]

References

1. Edmonds, Jack (1991), "A glimpse of heaven", in J.K. Lenstra; A.H.G. Rinnooy Kan; A. Schrijver, *History of Mathematical Programming --- A Collection of Personal Reminiscences*, CWI, Amsterdam and North-Holland, Amsterdam, pp. 32–54
2. Edmonds, Jack (1965). "Paths, trees, and flowers". *Canad. J. Math.* **17**: 449–467. doi:10.4153/CJM-1965-045-4.
3. Edmonds, Jack (1965). "Maximum matching and a polyhedron with 0,1-vertices". *Journal of Research of the National Bureau of Standards Section B*. **69**: 125–130.
4. Schrijver, Alexander. *Combinatorial Optimization: Polyhedra and Efficiency*. Algorithms and Combinatorics. **24**. Springer.
5. Lovász, László; Plummer, Michael (1986). *Matching Theory*. Akadémiai Kiadó. ISBN 963-05-4168-8.
6. Karp, Richard, "Edmonds's Non-Bipartite Matching Algorithm", *Course Notes. U. C. Berkeley* (PDF)
7. Tarjan, Robert, "Sketchy Notes on Edmonds' Incredible Shrinking Blossom Algorithm for General Matching", *Course Notes, Department of Computer Science, Princeton University* (PDF)
8. Kenyon, Claire; Lovász, László, "Algorithmic Discrete Mathematics", *Technical Report CS-TR-251-90, Department of Computer Science, Princeton University*
9. Micali, Silvio; Vazirani, Vijay (1980). *An O(V$^{1/2}$E) algorithm for finding maximum matching in general graphs*. 21st Annual Symposium on Foundations of Computer Science,. IEEE Computer Society Press, New York. pp. 17–27.
10. Kolmogorov, Vladimir (2009), "Blossom V: A new implementation of a minimum cost perfect matching algorithm", *Mathematical Programming Computation*, **1** (1): 43–67

Categories: Graph algorithms │ Matching