15-850: Advanced Algorithms **CMU, Spring 2017**
Lecture #6: Graph matchings I: combinatorial algorithms February 6, 2017
Lecturer: Anupam Gupta Scribe: Raymond Kang, Evan Palmer, Yihan Sun

1 Contents

- Matching on bipartite graphs
  - Ford-Fulkerson Algorithm
  - Augmenting paths and König's Theorem
- Matching on non-bipartite graphs
  - Tutte-Berge Theorem
  - Edmonds Blossom Algorithm

2 Notation and Definitions

During this lecture, we will consider only simple graphs.

We will denote an undirected unweighted graph G with the vertex set V of size n and the edge set E of size m as:

$$G = (V, E)$$

We will denote an undirected unweighted bipartite graph H with left vertices L, right vertices R and edges E as:

$$H = (L, R, E)$$

Definition 6.1 (Matching)**.** A matching on a graph is some subset of the edges $M \subseteq E$ which have no endpoints in common. Equivalently, (V, M) has max degree 1.

Given a matching M on a graph, we will say that:
A vertex v is *open* or *exposed* if there is no edge in the matching adjacent to v
A vertex v is *closed* or *covered* if there exists an edge e in the matching such that v is an endpoint of e.

Definition 6.2 (Perfect Matching)**.** A perfect matching on a graph is a matching M such that $|M| = |V|/2$.

Definition 6.3 (Maximum Matching)**.** The maximum matching on a graph is the matching with the largest cardinality. We will denote the size of the maximum matching of graph G as $MM(G)$.

Definition 6.4 (Maximal Matching)**.** A maximal matching on a graph is a matching where no additional edges can be added without violating the shared endpoint constraint.

3 Bipartite Graphs

3.1 Ford-Fulkerson for matchings

Consider a bipartite graph $G = (L, R, E)$. We can find the maximum matching of it via a reduction to maximum flow.

We can do the following:

- Create a super source s and connect it to all of the vertices in L with a directed edge of capacity 1 from s to L.
- Create a super sink t and connect all of the vertices in R to it with a directed edge of capacity 1 from R to t.
- Change every edge in E into a directed edge which points from its endpoint in L to its endpoint in R, and give the edge capacity 1.

We can now run Ford-Fulkerson [FF56] on this graph. Due to the integral flow theorem, we know that we can construct a max flow which gives a capacity of either 1 or 0 to each edge in our graph.

We will choose the edges which are fully saturated in the maxflow graph and present in our original graph. These edges will form a maximum matching.

3.2 Augmenting paths for matchings

Alternatively, we can find the maximum matching via a more direct method, i.e. augmenting paths.

Definition 6.5 (Alternating Path)**.** Given a matching M, an M-alternating path has edges in M and not in M alternating.

Definition 6.6 (Augmenting Path)**.** Given a matching M, an M-augmenting path is an M-alternating path with both endpoints open.

For sets S, T, we define $S \triangle T := (S \setminus T) \cup (T \setminus S)$. Note that if P is M-augmenting, then $P \triangle M$ is another matching with cardinality $|M| + 1$.

Theorem 6.7. *[Ber57] M is a maximum matching in G if and only if there are no M-augmenting paths in G. G need not be bipartite.*

Proof. ($\Rightarrow$) If M is a maximum matching, there are clearly no M-augmenting paths P, or $M' = M \triangle P$ is a larger matching.

($\Leftarrow$) Suppose M is not maximum, and the maximum matching M' has $|M'| > |M|$. Consider $S = M \triangle M'$. Every vertex is incident to at most 2 edges in S, so S consists of only paths and cycles:

- Every cycle in S has even length because G is bipartite, and has alternating edges from M and M'.
- Every path in S has alternating edges from M and M'.
- Because $|M'| > |M|$, there exists a path in S that has more M'-edges than M edges. But this is an M-augmenting path.

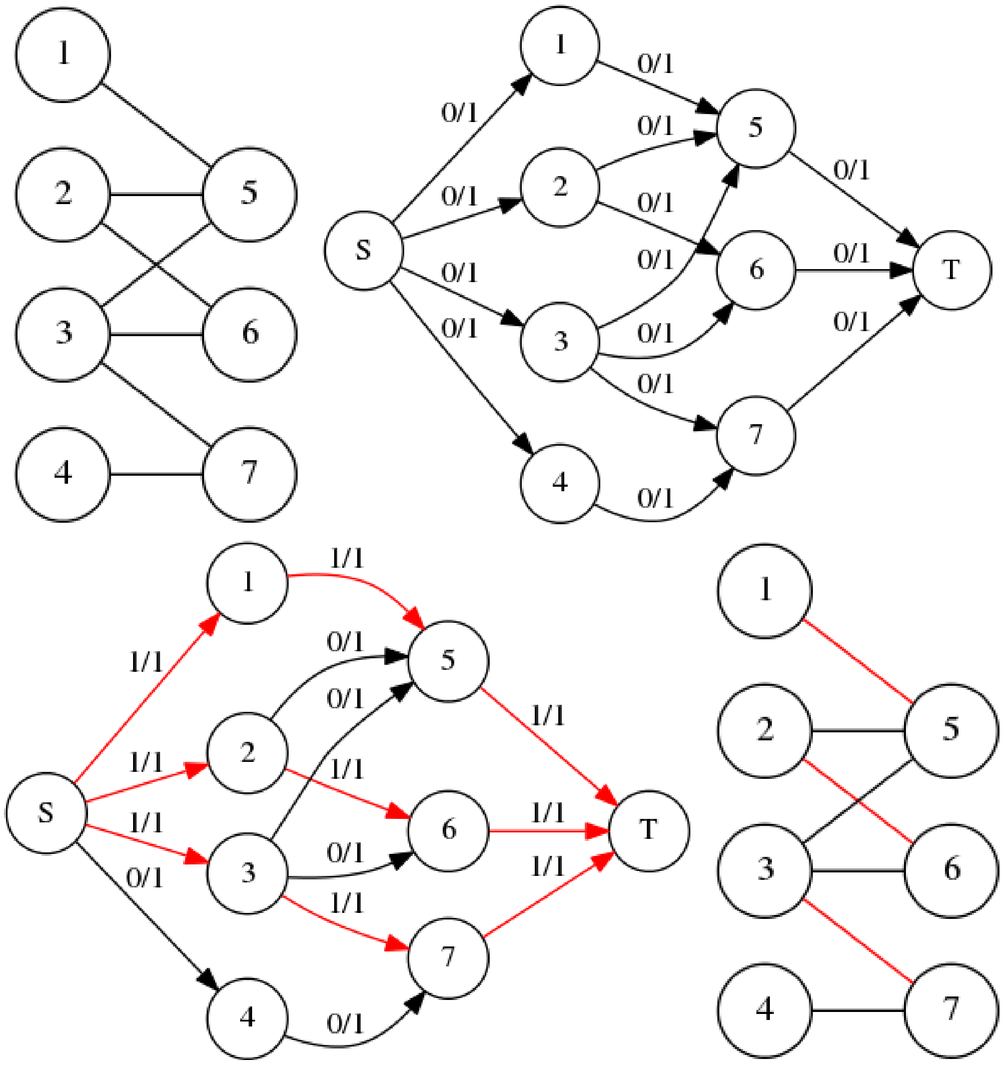


Figure 6.1: The use of Ford-Fulkerson to determine a matching

□

How do we find an augmenting path if one exists? An algorithm is given the proof of the following theorem. We recall the definition of a vertex cover:

Definition 6.8 (Vertex Cover)**.** A vertex cover of a graph is a set of vertices C such that every edge in the graph has at least one endpoint in C. We denote the size of the smallest vertex cover of graph G as $VC(G)$.

Theorem 6.9. *[Kön31] On a bipartite graph, the cardinality of the smallest vertex cover is the size of the largest possible matching.*

$$VC(G) = MM(G)$$

This is a special case of the max-flow min-cut theorem.

Proof. $(VC \geq MM)$

Since edges in the matching MM share no common endpoints, in order to have a vertex cover, we must have at least one vertex for each edge. □

Proof. $(VC \leq MM)$

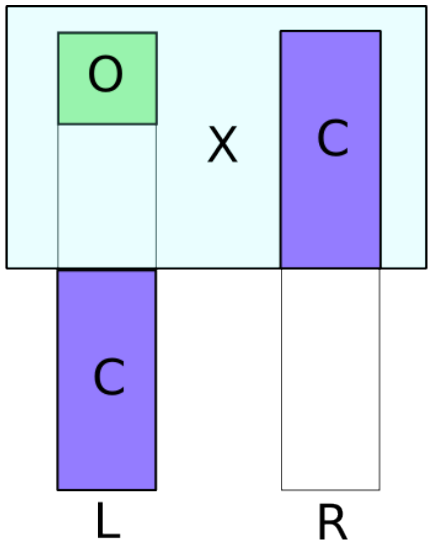
Given a matching M, we will either find an M-augmenting path in $O(m)$ if such a path exists, or find a vertex cover of size $|M|$. This shows that $VC \leq MM$. Essentially we perform a breadth-first search, as follows:

```
Mark all open vertices in $L$ (that are non-isolated) as being in layer 0.
For $i = 0, 1, 2, \ldots$:
  Mark all unmarked nodes with a *non-matching* edge to level $2i$ as being in layer $2i+1$.
  Mark all unmarked nodes with a *matching* edge to level $2i+1$ as being in layer $2i+2$.
```

If there is a vertex at level $2i+1$ with no matching edge, then we have found an M-augmenting path going from level 0 to level $2i+1$ in $O(m)$ time. Otherwise, let X be the set of marked vertices.

We will take our vertex cover C to be the vertices on the left side which are not in X, and the vertices on the right side which are in X.

$$C = (L \setminus X) \cup (R \cap X)$$



Claim 6.10. *C is a vertex cover.*

To show that this is a vertex cover, we must demonstrate that there can be no edges between $L \cap X$ and $R \setminus X$. Since this is a bipartite graph all other possible edges must have at least one endpoint in C

- There can be no unmatched edge from the open vertices $L \cap X$ to $R \setminus X$ since otherwise that vertex would be reachable from O and therefore in X. There cannot be a matched edge from an open vertex by definition.
- There can be no unmatched edge from a closed vertex in $L \cap X$ to $R \setminus X$ as this would make the endpoint in $R \setminus X$ reachable from O.
- There can be no matched edge from $R \setminus X$ to $L \cap X$. Since everything in $L \cap X$ is reachable from O, it must have a matched edge from something in $R \cap X$. Accordingly it cannot have another matched edge due to matching constraints.

Claim 6.11. *$|C| \leq MM$.*

- Every vertex in $R \cap X$ must have an edge in the matching as otherwise it would be open, and there would be an augmenting path
- Every vertex in $L \setminus X$ must have an edge in the matching since no vertices in $L \setminus X$ are open.
- There can be no edges between $L \setminus X$ and $R \cap X$ as this would mean that things in $L \setminus X$ were reachable from X and therefore O

So, every vertex in $(L \setminus X) \cup (R \cap X)$ corresponds to a unique edge in the matching, and $|C| \leq MM$. □

The algorithm in the proof above shows that an augmenting path can be found in $O(m)$ time. Hence maximum matching can be done in $O(mn)$ time.

3.3 Other algorithms

Hopcroft-Karp [HK73]: $O(m\sqrt{n})$ - This is the fastest current algorithm for matching on bipartite graphs. It finds many augmenting paths at once and then combines them in a clever way.

Even, Tarjan [ET75]: $O(\min(m\sqrt{m}, mn^{2/3}))$ - An algorithm for computing maximum flows on unit graphs.

4 Non-Bipartite Graphs

4.1 Tutte-Berge Theorem

Theorem 6.12. *[Ber58] Given a graph G, the size of the maximum matching is described by the following equation.*

$$MM(G) = \min_{U \subseteq V} \frac{n + |U| - odd(G \setminus U)}{2}$$

Here U is a set of vertices such that if U is removed from G, the remainder of G becomes disjoint with pieces $\{K_1, K_2, ..., K_t\}$. The quantity $odd(G \setminus U)$ is the number of such pieces with odd cardinality.

It is clear that the size of any matching M must be bounded by this quantity.

At most everything in U can be in the matching. Similarly, since the partitions K_i are disjoint, vertices in K_i not matched with U can only be matched within the partition. This gives us:

$$\begin{aligned} |M| &\leq |U| + \sum_{i=1}^{t} \left\lfloor \frac{k_i}{2} \right\rfloor \\ &= |U| + \frac{n - |U|}{2} - \frac{\text{odd}(G \setminus U)}{2} \\ &= \frac{|U| + n - \text{odd}(G \setminus U)}{2}. \end{aligned}$$

To make some sense of this formula, we can consider the trivial case when $U = \emptyset$. Here the formula states that if the graph has an even size, then the maximum matching cannot be bigger than $n/2$, and if it has an odd size, then the maximum matching cannot be bigger than $(n-1)/2$.

Another special case is when U is any vertex cover with size V. Then the K_i's must be isolated vertices, so $\text{odd}(G \setminus U) = n - V$. This gives us $MM \leq \frac{V+n-(n-V)}{2} = V$, i.e. the size of the maximum matching is at most the size of any vertex cover.

4.2 Edmonds Blossom Algorithm

The blossom algorithm is an algorithm for finding the maximum matching in a general graph.

4.2.1 Definitions and Notations

A **flower** is a set of nodes and edges starting at an open vertex, with a "stem" of even number of edges (matched and unmatched edges alternating), and a cycle of odd number of edges (again with alternating matched and unmatched edges, but with the two edges adjacent to the stem unmatched). The cycle is called the **blossom**. A illustration is shown in Figure 6.2(a).

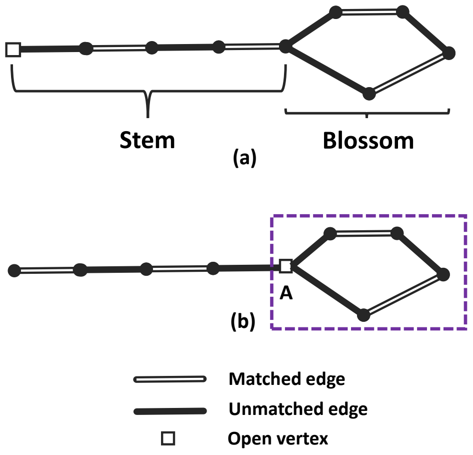


Figure 6.2: An example of blossom and the toggling of the stem.

4.2.2 Overview

The idea of Edmonds Blossom Algorithm [Edm65] comes from Berge's Theorem.

Theorem 6.13. *Given a graph G and a matching M, M is a maximum matching if and only if there exists no M-augmenting path.*

For a blossom, we can always toggle the stem part, such that the open vertex moves to the blossom. As an illustration, after toggling the stem from Figure 6.2(a), we get the blossom in (b), where the vertex A is now the open vertex. After this toggling, the length of stem is zero, and the blossom is now the part in the purple rectangle.

Here we introduce an algorithm `FindAugPath` to explore augmenting paths in a graph given a matching M. There are three possible results of this algorithm:

1. **It returns "No M-augmenting path".** In this case, M is the maximum matching.
2. **An augmenting path P is found.** We can now augment along P, by setting $M \leftarrow M \triangle P$.

3. **A blossom is found.** Denote the blossom as B. Toggle the stem and shrink the blossom, getting the graph G/B. Since the blossom contains an open vertex at the base (after the toggle), the new vertex v_B obtained by shrinking is an open vertex in the new graph G/B. (illustrated in Figure 6.3). Call `FindAugPath` on G/B and M/B and get an M/B-augmenting path P' in G/B. Finally, extend P' to M-augmenting path P in G. (See Figure 6.4.)

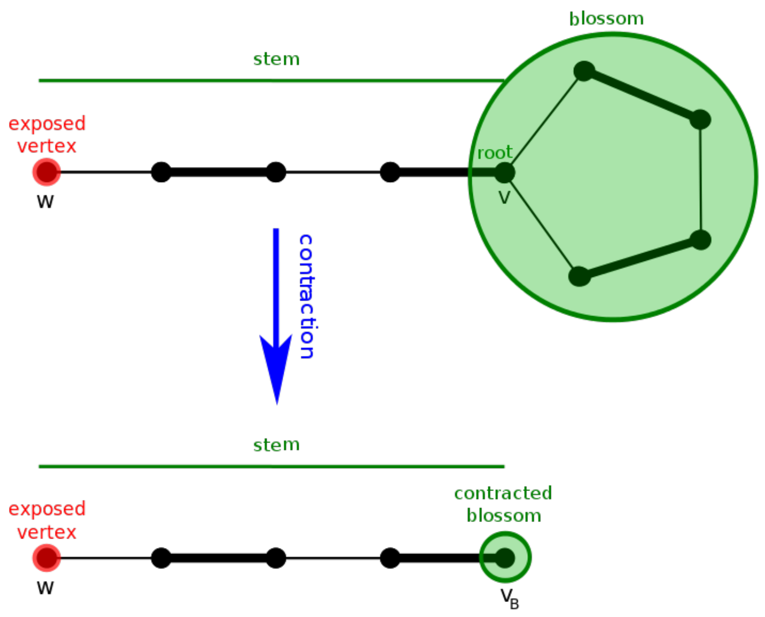


Figure 6.3: The shrinking of a blossom[1]

[1]Image found at http://en.wikipedia.org/wiki/Blossom_algorithm

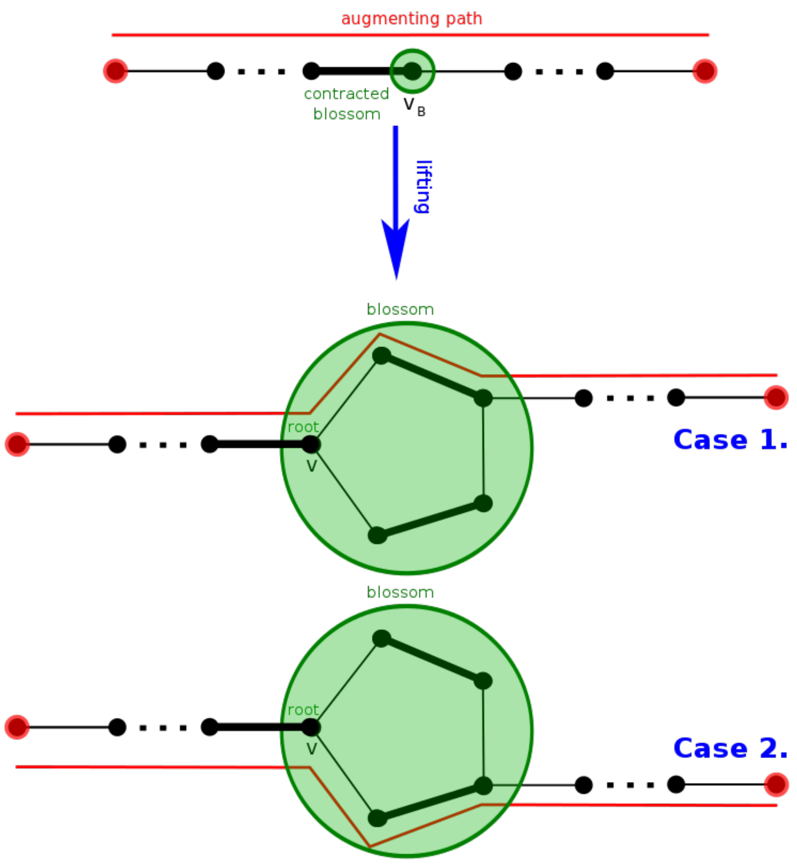


Figure 6.4: The translation of augmenting paths from $G \setminus B$ to G and back[2]

The correctness of `FindAugPath` are captured in the following theorem:

Theorem 6.14. *The algorithm `FindAugPath` runs in $O(m)$ time, and given graph G and matching M, if there exists an M-augmenting path in G, it returns either (a) a blossom B, or (b) an M-augmenting path.*

And to ensure that if there were an M-augmenting path in G, we will indeed find an M/B-augmenting path in G/B, and can lift this back to an M-augmenting path in the original graph, we need the second theorem.

Theorem 6.15. *Given graph G and matching M, let B be a blossom in G. There exists an M-augmenting path in G if and only if there exists an M/B augmenting path in G/B.*

Moreover, given P', an M/B augmenting path in G/B, we can get back an M-augmenting path P in G in $O(m)$ time.

We will prove these theorems in the following sections. But before that, let us show why this is an $O(mn^2)$ time algorithm.

4.2.3 Runtime Analysis

Observe that if we get back an M-augmenting path, we can increase the size of the matching; this takes only $O(mn)$ time. Else, if we get back a blossom B, we recurse on a smaller graph. This

[2]Image found at http://en.wikipedia.org/wiki/Blossom_algorithm

may happen repeatedly, but after at most $O(n)$ recursions we either return an augmenting path P on some smaller graph or get back "No augmenting path" for an answer. In the former case, we can lift this augmenting path back to an M-augmenting path for G; in the latter case we can use Theorem 6.15 to claim that there was no M-augmenting in the original graph. This takes a total of $O(mn)$ time. And we would need to do this at most $n/2$ times, since each time we augment the size of the matching by 1. The total runtime is $O(mn^2)$.

An aside: this was improved to a runtime of $O(m\sqrt{n})$ by Micali and V. Vazirani [MV80]. The resulting algorithm is quite involved; see a recent paper of Vijay Vazirani [Vaz13] giving a cleaner explanation. Another algorithm by Mucha and Sankowski [MS06], based on fast matrix multiplication, gives $O(n^\omega)$ complexity, where $\omega \approx 2.376$.

4.2.4 Proof of Theorem 6.15

We first give the proof of Theorem 6.15 as follows.

Proof. Since we can toggle the stem as stated in Section 4.2.2, we assume that the stem has length zero, and the only one vertex in the cycle which links to two unmatched edges is open. Since all other nodes in the blossom are matched, any edges going to vertices outside B are non-matching edges.

($\Rightarrow$) Suppose there is an M-augmenting path in G, denoted as P. If P does not go through the blossom B, the path still exists in G/B. If P goes through B, because there is only one open vertex in B, there must exist at least open vertex v' in G which is an end of P but is not in B. Because v_B is open in G/B, the path from v' to v_B is an M/B-augmenting path in G/B.

($\Leftarrow$) Suppose there is an M/B-augmenting path P' in G/B. If P' does not go through v_B, P' still exists in G.

If P' goes through v_B, because v_B is open, it must be an end of P'; denote the other end of P' by s. Suppose the edge (denoted as e) which connects v_B in P' is originally connecting some vertex v in B. If v is the open vertex in B, then s to v is an augmenting path in G. Otherwise, there is a matched edge and an unmatched edge connecting v in B respectively. Since e is unmatched, going from s to v to the open vertex in B via the direction of the matched edge forms an M-augmenting path in G.

Since the process to get from P' to the M-augmenting path in G be done algorithmically in $O(m)$ time, this proves the second part of the theorem. □

4.2.5 The Algorithm `FindAugPath`

The algorithm `FindAugPath` put all vertices in G in a layered structure to find an M-augmenting path or a blossom, or correctly report that there is no M-augmenting path. The construction is quite similar in spirit to the bipartite case, though with some differences since we may have odd cycles.

At a high level, here's the construction:

```
Mark all open vertices as being in layer 0.
For $i = 0, 1, 2, \ldots$:
  Mark all unmarked nodes with a *non-matching* edge to level $2i$ as being in layer $2i+1$.
  Mark all unmarked nodes with a *matching* edge to level $2i+1$ as being in layer $2i+2$.
```

If there exists a "cross" edge between two nodes of the same layer, output a blossom or augmenting path, else say "No M-augmenting path".

An illustration of the algorithm is given in Figure 6.5. In order to argue correctness, it is worth going over the steps again in more detail. Let L_i denote the i-th layer vertices in the following discussion.

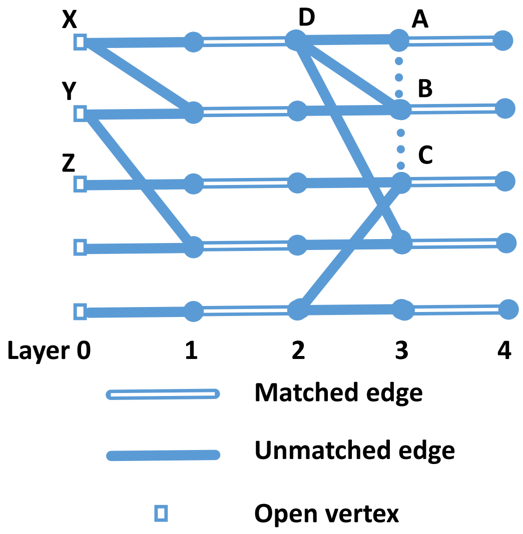


Figure 6.5: An illustration of `FindAugPath` algorihtm

1. Put all open vertices in L_0 and mark them.
2. Given the even layer L_{2i}, construct L_{2i+1} as follows. For each vertex $u \in L_{2i}$ and edge $(u, v) \in E \backslash M$, do the following:
   (a) *v is unmarked.* Then put v in L_{2i+1}, and mark it.
   (b) v is marked. Let us look at the layer containing v.
      i. $v \in L_{2i}$. This means that there is an unmatched edge linking two vertices in the same layer. This must result in an augmenting path or a blossom! Indeed, the way we construct the layered graph, there are alternating paths P and Q from both u and v to open vertices in L_0. If P and Q do not intersect, then $P \circ (u, v) \circ Q$ gives an M-augmenting path. If P and Q intersect, they must first intersect some vertex w at an even layer, and the cycle containing u, v, w gives us the blossom, with the stem being one path from w back to an open vertex in L_0. (Success!)
      ii. v is in a previous even layer L_{2j}. If so, u should have been explored in the odd layer L_{2j+1}, which is impossible.
      iii. v is in an odd layer. This is fine, we just observe that (u, v) is an *even-odd* edge.
3. Given the odd layer L_{2i+1}, construct L_{2i+2} as follows. For each vertex $u \in L_{2i+1}$ and *matching* edge $(u, v) \in M$, do the following:
   (a) v is unmarked. Then add v to L_{2i+2}, and mark v.
   (b) v is marked. Again, consider what layer v belongs to.

i. $v \in L_{2i+1}$. This means that there is an matching edge linking two vertices in the same layer. This must result in an augmenting path or a blossom. The analysis is similar to Case b(i). (Success!)
As examples, in Figure 6.5, if there is an edge connecting A and B, we find a blossom cycle containing three vertices A, B, D, with the stem from X to D. On the other hand, an edge connecting C and B gives the augmenting path $Y \to B \to C \to Z$.

ii. v is in previous layers. This is impossible because all previous explored vertices are either open or matched using other edges. There cannot be a matching edge connecting v to u.

From this construction, we observe that if the algorithm does not succeed, all edges are even-odd edges. Now we can prove Theorem 6.14.

Proof. We now prove that if there is an M-augmenting path in G, the `FindAugPath` algorithm will either return an M-augmenting path or a blossom.

For sake of contradiction, suppose that there exists an M-augmenting path P, and the algorithm does not succeed. From the observation above, this means that when we constructed the layered graph, each edge in G was an even-odd edge. Consider the augmenting path P, the starting open vertex is in L_0, which is an even layer. Thus the next vertex should be in an odd layer, and the next one should be in an even layer, and so forth. Because there are odd number of edges in P, there are an even number of vertices, and hence the last vertex must be in an odd layer. However, the last vertex is open, and hence in L_0, an even layer. We get a contradiction. □

References

[Ber57] Claude Berge. Two theorems in graph theory. *Proc. Nat. Acad. Sci. U.S.A.*, 43:842–844, 1957. 6.7

[Ber58] Claude Berge. Sur le couplage maximum d'un graphe. *C. R. Acad. Sci. Paris*, 247:258–259, 1958. 6.12

[Edm65] Jack Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17:449–467, 1965. 4.2.2

[ET75] Shimon Even and R. Endre Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975. 3.3

[FF56] L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canad. J. Math.*, 8:399–404, 1956. 3.1

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973. 3.3

[Kön31] D König. Graph and matrices. *Mat. Fig. Lapok (Hungarian)*, 38:116–119, 1931. 6.9

[MS06] Marcin Mucha and Piotr Sankowski. Maximum matchings in planar graphs via Gaussian elimination. *Algorithmica*, 45(1):3–20, 2006. 4.2.3

[MV80] Silvio Micali and Vijay V Vazirani. An $O(\sqrt{|V|}|E|)$ algoithm for finding maximum matching in general graphs. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 17–27. IEEE, 1980. 4.2.3

[Vaz13] Vijay V. Vazirani. A simplification of the MV matching algorithm and its proof, 2013. 4.2.3