

6

Graph Matchings I: Combinatorial Algorithms

Another fundamental graph problem is to find *matchings*: these are subsets of edges that do not share endpoints. Matchings arise in various contexts: matching tasks to workers, or advertisements to slots, or roommates to each other. Moreover, matchings have a rich combinatorial structure. The classical results can be found in *Matching Theory* by Laci Lovász and Michael Plummer ¹, though Lex Schrijver's *Combinatorial Optimization: Polyhedra and Efficiency* might be easier to find, and contains more recent developments as well.

Several different and interesting algorithmic techniques can be used to find large matchings in graphs; we will discuss them over the next few chapters. This chapter discusses the simplest, combinatorial algorithms.



6.1 Notation and Definitions

Consider an undirected (simple and connected) graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ as usual. The graph is unweighted; we will consider weighted versions of matching problems in later chapters. When considering *bipartite* graphs, where the vertex set has parts $V = L \uplus R$ (the “left” and “right”, and the edges $E \subseteq L \times R$, we may denote the graph as $G = (L, R, E)$.

Definition 6.1 (Matching). A matching in graph G is a subset of the edges $M \subseteq E$ which have no endpoints in common. Equivalently, the edges in M are disjoint, and hence every vertex in (V, M) has maximum degree 1.

Given a matching M in G , a vertex v is *open* or *exposed* or *free* if no edge in the matching is incident to v , else the vertex is *closed* or *covered* or *matched*. Observe: the empty set of edges is a matching. Moreover, any matching can have at most $|V|/2$ edges, since each edge covers two vertices, and each vertex can be covered by at most one edge.

Definition 6.2 (Perfect Matching). A *perfect matching* M is a matching such that $|M| = |V|/2$. Equivalently, every vertex is matched in the matching M .

Definition 6.3 (Maximum Matching). A *maximum cardinality matching* (or simply maximum matching) in G is a matching with largest possible cardinality. The size of the maximum matching in graph G is denoted $MM(G)$.

Definition 6.4 (Maximal Matching). A *maximal matching* on a graph is a matching that is inclusion-wise maximal; that is, no additional edges can be added to M while maintaining the matching property. Hence, $M \cup \{e\}$ is not a matching for all edges $e \notin M$.

The last definition is given to mention something we will *not* be focusing on; our interest is in perfect and maximum matchings. That being said, it is a useful exercise to show that any maximal matching in G has at least $MM(G)/2$ edges.

6.1.1 Augmenting Paths for Matchings

Since we want to find a maximum matching, a question we may ask is: *given a matching M , can we (efficiently) decide if it is a maximum matching?* One answer to this was suggested by Berge, who gave a characterization of maximum matchings in terms of “augmenting” paths.

Definition 6.5 (Alternating Path). For matching M , an *M -alternating path* is a path in which edges in M alternate with those not in M .

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

Given sets S, T , their symmetric difference is denoted

$$S \Delta T := (S \setminus T) \cup (T \setminus S).$$

The following theorem explains the name for augmenting paths.

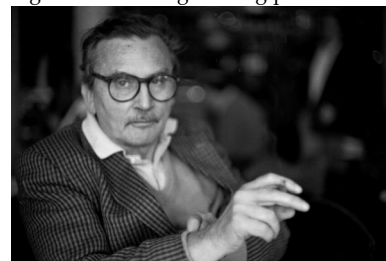
Theorem 6.7 (Berge’s Optimality Criterion). *A matching M is a maximum matching in graph G if and only if there are no M -augmenting paths in G .*

Proof. If there is an M -augmenting path P , then $M' := M \Delta P$ is a larger matching than M . (Think of getting M' by toggling the dashed edges in the path to solid, and *vice versa*). Hence if M is maximum matching, there cannot exist an M -augmenting path.

Conversely, suppose M is not a maximum matching, and matching M' has $|M'| > |M|$. Consider their symmetric difference $S := M \Delta M'$.

Figure 6.1: An alternating path P (dashed edges are not in P , solid edges are in P)

Figure 6.2: An augmenting path



Berge (1957)

Every vertex is incident to at most 2 edges in S (at most one each from M and M'), so S consists of only paths and cycles, all of them having edges in M alternating with edges in M' . Any cycle with this alternating structure must be of even length, and any path has at most one more edge from one matching than from the other. Since $|M'| > |M|$, there must exist a path in S with one more edge from M' than from M . But this is an M -augmenting path. \square

If we could efficiently find an M -augmenting path (if one exists), we could repeatedly augment the current matching until we have a maximum matching. However, Berge's theorem does not immediately give an efficient algorithm: finding an M -augmenting path could naively take exponential time. We now give algorithms to efficiently find augmenting paths, first in bipartite graphs, and then in general graphs.

6.2 Bipartite Graphs

Finding an M -augmenting path (if one exists) in bipartite graphs is an easier task, though it still requires cleverness. A first step is to consider a "dual" object, which is called a vertex cover. **Check Schrijver for which paper proved what.**

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

Note that the entire set V is trivially a vertex cover, and the challenge is to find small vertex covers. We denote the size of the smallest cardinality vertex cover of graph G as $VC(G)$. Our motivation for calling it a "dual" object comes from the following fundamental theorem from the early 20th century:

Theorem 6.9 (König's Minimax Theorem). *In a bipartite graph, the size of the largest possible matching equals the cardinality of the smallest vertex cover:*

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

This theorem is a special case of the max-flow min-cut theorem, which you may have seen before. **And we'll see again in Chapter.** It is first of many min-max relationships, many of which lead to efficient algorithms. Indeed, the algorithm for finding augmenting paths will come out of the proof of this theorem.

Proof. In many such proofs, there is one easy direction. Here, it is proving that $MM(G) \leq VC(G)$. Indeed, the edges of any matching share no endpoints, so covering a matching of size $MM(G)$ requires

Dénes König (1916)



Exercise: Use König's theorem to prove P. Hall's theorem: *A bipartite graph has a matching that matches all vertices of L if and only for every subset $S \subseteq L$ of vertices, $|N(S)| \geq |S|$. Here $N(S)$ denotes the "neighborhood" of S , i.e., those vertices with a neighbor inside S .*

at least as many vertices. The minimum vertex cover size is therefore at least $MM(G)$.

Next, we prove that $MM(G) \geq VC(G)$. To do this, we give a linear-time algorithm that takes as input an arbitrary matching M , and either returns an M -augmenting path (if such a path exists), or else returns a vertex cover of size $|M|$. Since a maximum matching M admits no M -augmenting path by Berge's theorem, we would get back a vertex cover of size $MM(G)$, thereby showing $VC(G) \leq MM(G)$.

The proof is an "alternating" breadth-first search: it starts with all open nodes among the left vertex set L , and places them at level 0. Then it finds all the (new) neighbors of these nodes reachable using non-matching edges, and then all (new) neighbors of those nodes using matching edges, and so on. Formally, the algorithm is as follows, where we use $X_{\leq j}$ to denote $X_0 \cup \dots \cup X_j$.

```

9.1  $X_0 \leftarrow$  all open vertices in  $L$ 
9.2 for  $i = 0, 1, 2, \dots$  do
9.3    $X_{2i+1} \leftarrow \{v \mid \text{exists } u \in X_{2i} \text{ s.t. } uv \notin M, \text{ and } v \notin X_{\leq 2i}\}$ 
9.4    $X_{2i+2} \leftarrow \{v \mid \text{exists } u \in X_{2i+1} \text{ s.t. } uv \in M, \text{ and } v \notin X_{\leq 2i+1}\}$ 
    
```

Let us make a few observations about the procedure. First, since the graph is bipartite, X_i is a subset of L for even levels i , and of R for odd levels i . Next, all vertices in $X_2 \cup X_4 \cup \dots$ are matched vertices, since they are reached from the previous level using an edge in the matching. Moreover, if some odd level X_{2i+1} contains an open node v , we have found an M -alternating path from an open node in X_0 to v , and hence we can stop and return this augmenting path.

Hence, suppose we do not find an open node in an even level, and stop when some X_j is empty. Let $X = \cup_j X_j$ be all nodes added to any of the sets X_j ; we call these **marked** nodes. Define the set C to be the vertices on the left which are *not marked*, plus the vertices on the right which are **marked**. That is,

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

We claim that C is a vertex cover of size $|M|$.

Claim 6.10. C is a vertex cover.

Proof. G is a bipartite graph, and C hits all edges that touch $R \cap X$ and $L \setminus X$. Hence we must show there are no edges between $L \cap X$ and $R \setminus X$, i.e., between the top-left and bottom-right of the figure.

1. There can be no unmatched edge from the *open* vertices in $L \cap X$ to $R \setminus X$, else that vertex would be reachable from X_0 and so belong to X_1 . Moreover, an open vertex has no unmatched edges, by

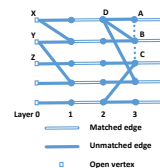


Figure 6.3: Illustration of the process to find augmenting paths in a bipartite graph. **Mistakes!**

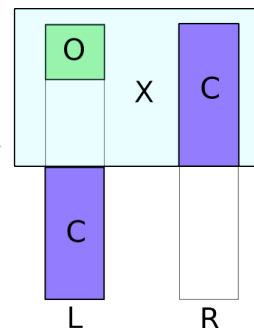


Figure 6.4: X = set of marked vertices, O = marked open vertices, C = claimed vertex cover of G . **Change**

definition. Hence, any “offending edges” out of $L \cap X$ must come from a covered vertex.

2. There can be no non-matching edge from a covered vertex in $L \cap X$ to some node u in $R \setminus X$, else this node u would have been added to some level X_{2i+1} .
3. Finally, there can be no matching edge between a covered vertex in $L \cap X$ and some vertex in $R \setminus X$. Indeed, every covered node in $L \cap X$ (i.e., those in X_2, X_4, \dots) was reached via a matching edge from some node in $R \cap X$. There cannot be another matching edge from some node in $R \setminus X$ incident to it.

This shows that C is a vertex cover. □

Claim 6.11. $|C| \leq |M|$.

We use a simple counting argument:

- Every vertex in $R \cap X$ has a matching edge incident to it; else it would be open, giving an augmenting path.
- Every vertex in $L \setminus X$ has an incident edge in the matching, since no vertices in $L \setminus X \subseteq L \setminus X_0$ are open.
- There are no matching edges between $L \setminus X$ and $R \cap X$, else they would have been explored and added to X .

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

Observe that the proof of König’s theorem is algorithmic, and can be implemented to run in $O(m)$ time. Now, starting from some trivial matching, we can use this linear-time algorithm to repeatedly augment until we have a maximum matching. This means that maximum matching on bipartite graphs has an $O(mn)$ -time algorithm.

Observe: this algorithm also gives a “proof of optimality” of the maximum matching M , in the form of a vertex cover of size $|M|$. By the easy direction of König’s theorem, this is a vertex cover of *minimum cardinality*. Therefore, while finding the smallest vertex cover is NP-hard for general graphs, we have just solved the minimum vertex cover problem on bipartite graphs.

One other connection: if you have seen the Ford-Fulkerson algorithm for computing maximum flows, the above algorithm may seem familiar. Indeed, modeling the maximum matching problem in bipartite graphs as that of finding a maximum integer s - t flow, and running the Ford-Fulkerson “augmenting paths” algorithm results in the same result. Moreover, the minimum s - t cut corresponds to a vertex cover, and the max-flow min-cut theorem proves König’s theorem. The figure to the right illustrates this on an example. Figure fix.

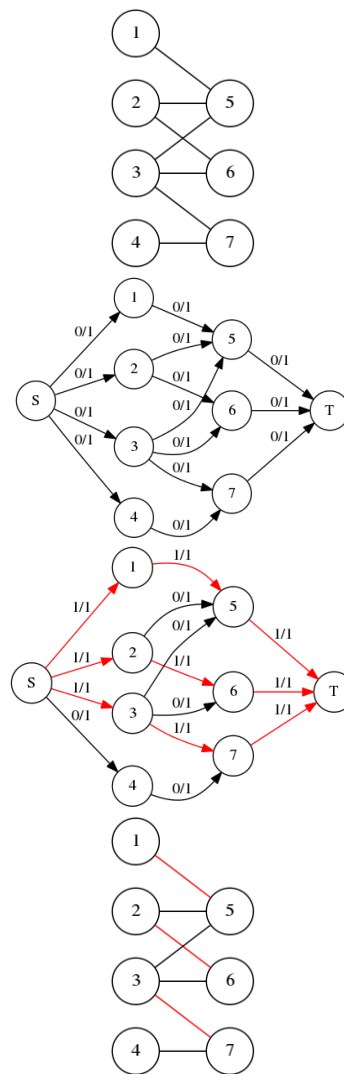


Figure 6.5: Use Ford-Fulkerson algorithm to find a matching

6.2.1 Other algorithms

There are faster algorithms to find maximum matchings in bipartite graphs. For a long time, the fastest one was an algorithm by John Hopcroft and Dick Karp, which ran in time $O(m\sqrt{n})$. It finds many augmenting paths at once, and then combines them in a clever way. There is also a related algorithm of Shimon Even and Bob Tarjan, which runs in time $O(\min(m\sqrt{m}, mn^{2/3}))$; in fact, they compute maximum flows on unit-capacity graphs in this running time.

Hopcroft and Karp (1973)

Even and Tarjan (1975)

There was remarkably little progress on the maximum matching problem until 2016, when Aleksander Madry gave an algorithm that runs in time $\tilde{O}(m^{10/7})$ time—in fact the algorithm also solves the unit-capacity maximum-flow problem in that time. It takes an interior-point algorithm for solving general linear programs, and specializes it to the case of maximum matchings. We may discuss this max-flow algorithm in a later chapter. The current best runtime for the unit-capacity maximum-flow problem is $m^{4/3+o(1)}$, due to Yang Liu and Aaron Sidford². [Read these. Hw problem: equivalence between unit-capacity max-flow and matching.](#)

Madry (2016)

6.3 General Graphs: The Tutte-Berge Theorem

The matching problem on general (non-bipartite) graphs gets more involved, since the structure of matchings is richer. For example, the flow-based approaches do not work any more. And while Berge's theorem (Theorem 6.7) still holds in this case, König's theorem (Theorem 6.9) is no longer true. Indeed, the 3-cycle C_3 has a maximum matching of size 1, but the smaller vertex cover is of size 2. However, we can still give a min-max relationship, via the Tutte-Berge theorem.

To state it, let us give a definition: for a subset $U \subseteq V$, suppose deleting the nodes of U and their incident edges from G gives connected components $\{K_1, K_2, \dots, K_t\}$. The quantity $\text{odd}(G \setminus U)$ is the number of such pieces with an odd number of vertices.

Theorem 6.12 (The Tutte-Berge Max-Min Theorem). *Given a graph G , the size of the maximum matching is described by the following equation.*

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

Tutte (1947), Berge (1958)

Tutte showed that the graph has a *perfect matching* precisely if for every $U \subseteq V$, $\text{odd}(G \setminus U) \leq |U|$. Berge gave the generalization to *maximum matchings*.

The expression on the right can seem a bit confusing, so let's consider some cases.

- If $U = \emptyset$, we get that if $|V|$ is even then $MM(G) \leq n/2$, and if $|V|$ is odd, the maximum matching cannot be bigger than $(n-1)/2$. (Or if G is disconnected with k odd-sized components, this gives $n/2 - k/2$.)

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

Trying special cases is a good way to understand the

Proof of the \leq direction of Theorem 6.12. The easy direction is to show that $MM(G)$ is at most the quantity on the right. Indeed, consider a maximum matching M . At most $|U|$ of the edges in M can be hit by nodes in U ; the other edges must lie completely within some connected component of $G \setminus U$. The maximum size of a matching within K_i is $\lfloor K_i/2 \rfloor$, and it are these losses from the odd components that gives the expression on the right. Indeed, we get

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

We can prove the “hard” direction using induction (see the webpage for several such proofs). However, we defer it for now, and derive it later from the proof of the Blossom algorithm. \square

Can we prove Konig from TB?

6.4 The Blossom Algorithm

The Blossom algorithm for finding the maximum matching in a general graph is by Jack Edmonds. Recall: the algorithm for minimum-weight arborescences in §?? was also due to him, and you may see some similarities in these two algorithms.

Edmonds (1965)

Theorem 6.13. *Given a graph G , the Blossom algorithm finds a maximum matching M in time $O(mn^2)$.*

The rest of this section defines the algorithm, and proves this theorem. The essential idea of the algorithm is simple, and similar to the one for the bipartite case: if we have a matching M , Berge’s characterization from Theorem 6.7 says that if M is not optimal, there exists an M -augmenting path. So the natural idea would be to find such an augmenting path. However, it is not clear how to do this directly. The clever idea in the Blossom algorithm is to either find an M -augmenting path, or else find a structure called a “blossom”.

The good thing about blossoms is that we can use them to contract the graph in a certain way, and make progress. Let us now give some definitions, and details.

A *flower* is a subgraph of G that looks like the object to the right: it has a open vertex at the base, then a *stem* with an even number of edges (alternating between matched and unmatched edges), and then a cycle with an odd number of edges (again alternating, though naturally having two unmatched edges adjacent to the stem). The cycle itself is called the *blossom*.

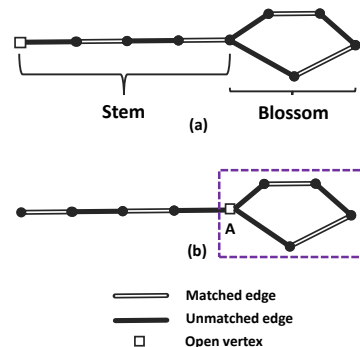


Figure 6.6: An example of blossom and the toggling of the stem.

6.4.1 The Main Procedure

The algorithm depends on a subroutine called `FindAugPath`, which has the following guarantee.

Lemma 6.14. *Given graph G and matching M , the subroutine `FindAugPath`, runs in $O(m)$ time. If G has an M -augmenting path, then it returns either (a) a flower F , or (b) an M -augmenting path.*

Note that we have not said what happens if there is no M -augmenting path. Indeed, we cannot find an augmenting path, but we show that the `FindAugPath` returns either a flower, or says “no M -augmenting path, and returns a Tutte-Berge set U achieving equality in Theorem 6.12 with respect to M . **Add this to the theorem?** We can now use this `FindAugPath` subroutine within our algorithm as follows.

1. Says “no M -augmenting path” and a set U of nodes. In this case, M is the maximum matching.
2. Finds augmenting path P . We can now augment along P , by setting $M \leftarrow M \Delta P$.
3. Finds a flower F . In this case, we don’t yet know if M is a maximum matching or not. But we can shrink the blossom down to get a smaller graph G' (and a matching M' in it), and recurse. Either we will find a proof of maximality of M' in G' , or an M' -augmenting path. This we can extend to the matching M in G . That’s the whole algorithm!

Let’s give some more details for the last step. Suppose we find a flower F , with stem S and blossom B . First, toggle the stem (by setting $M \leftarrow M \Delta S$): this moves the open node to the blossom, without changing the size of the matching M . (It makes the following arguments easier, with one less case to consider.) **(Change figure.)** Next, contract the blossom down into a single vertex v_B , which is now open. Denote the new graph $G' \leftarrow G/B$, and $M' \leftarrow M/B$.

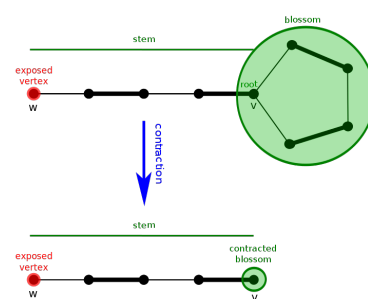


Figure 6.7: The shrinking of a blossom. Image found at http://en.wikipedia.org/wiki/Blossom_algorithm.

Given a graph and a subset $C \subseteq V$, recall that G/C denotes the *contraction* of C in G .

Since all the nodes in blossom B , apart from perhaps the base, were matched by edges within the blossom, M' is also a matching in G' .

Next, we recurse on this smaller graph G' with matching M' . Finally, if we get back an M' -augmenting path, we “lift” it to get an M -augmenting path (as we see soon). Else if we find that M' is a maximum matching in G' , we declare that M is maximum in G . To show correctness, it suffices to prove the following theorem.

Lemma 6.15. *Given graph G and matching M , suppose we shrink a blossom to get G' and M' . Then there exists an M -augmenting path in G if and only if there exists an M' -augmenting path in G' .*

Moreover, given an M' -augmenting path in G' , we can lift it back to an M -augmenting path P in G in $O(m)$ time.

Proof. Since we toggled the stem, the vertex v at the base of the blossom B is open, and so is the vertex v_B created in G' by contracting B . Moreover, all other nodes in the blossom are matched by edges within itself, so all edges leaving B are non-matching edges. The picture essentially gives the proof, and can be used to follow along.

(\Rightarrow) Consider an M -augmenting path in G , denoted by P . If P does not go through the blossom B , the path still exists in G' . Else if P goes through the blossom, we can assume that one of its endpoints is the base of the blossom (which is the only open node on the blossom)—indeed, any other M -augmenting path P can be rerouted to the base. (Figure!) So suppose this path P starts at the base and ends at some v' not in B . Because v_B is open in G' , the path from v_B to v' is an M' -augmenting path in G' .

(\Leftarrow) Again, an M' -augmenting path P' in G' that does not go through v_B still exists in G . Else, the M' -augmenting path P' passes through v_B , and because v_B is open in G' , the path starts at v_B and ends at some node t . Let the first edge on P' be $e' = v_B y$ for some node y , and let it correspond to edge $e = xy$ in G , where $x \in B$. Now, if v is the open vertex at the base of the blossom, following one of the two paths (either clockwise or counter-clockwise) along the blossom from v to x , using the edge xy and then following the rest of the path P' from y to t gives an M -augmenting path in G . (This is where we use the fact that the cycle is odd, and is alternating except for the two edges incident to v .)

The process to get from P' in G' to the M -augmenting path in G be done algorithmically in $O(m)$ time, completing the proof. \square

We can now analyze the runtime, and prove Theorem 6.13:

Proof of Theorem 6.13. We first call FindAugPath, which takes $O(m)$ time. We are either done (because M is a maximum matching, or else

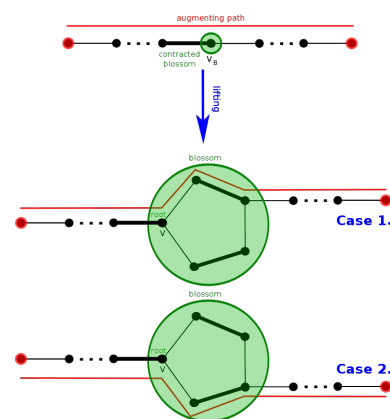


Figure 6.8: The translation of augmenting paths from $G \setminus B$ to G and back.

we have an augmenting path), or else we contract down in another $O(m)$ time to get a graph G' with at most $n - 3$ vertices and at most m edges. Inductively, the time taken in the recursive call on G' is $O(m(n - 3))$. Now lifting an augmenting path takes $O(m)$ time more. So the total runtime to find an augmenting path in G (if one exists) is $O(mn)$.

Finally, we start with an empty matching, so its size can be augmented at most $n/2$ times, giving us a total runtime of $O(mn^2)$. \square

6.4.2 The FindAugPath Subroutine

The subroutine FindAugPath is very similar to the analogous procedure in the bipartite case, but since there is no notion of left and right vertices, we start with level X_0 containing *all* vertices that are unmatched in M_0 , and try to grow M -alternating paths from them, in the hope of finding an M -augmenting path.

```

9.1  $X_0 \leftarrow$  all open vertices in  $V$ 
9.2 for  $i = 0, 1, 2, \dots$  do
9.3    $X_{2i+1} \leftarrow \{v \mid \text{exists } u \in X_{2i} \text{ s.t. } uv \notin M, \text{ and } v \notin X_{\leq 2i}\}$ 
9.4    $X_{2i+2} \leftarrow \{v \mid \text{exists } u \in X_{2i+1} \text{ s.t. } uv \in M, \text{ and } v \notin X_{\leq 2i+1}\}$ 
9.5 if exists a "cross" edge between nodes of same level then
9.6   return augmenting path or flower
9.7 else
9.8   say "no  $M$ -augmenting path"

```

As before, let $X_{\leq j}$ denote $X_0 \cup \dots \cup X_j$, and let nodes added to some level X_j be called *marked*.

To argue correctness, let us look at the steps above in more detail. In line 9.2, for each vertex $u \in X_{2i}$, we consider the possible cases for each non-matching edge uv incident to it:

1. If v is not in $X_{\leq 2i+1}$ already (i.e., not marked already) then we add it to X_{2i+1} . Note that $v \in X_{2i+1}$ now has an M -alternating path to some node in X_0 , that hits each layer exactly once.
2. If $v \in X_{2i}$, then uv is an unmatched edge linking two vertices in the same level. This gives an augmenting path or a blossom! Indeed, by construction, there are M -alternating paths P and Q from u and v to open vertices in X_0 . If P and Q do not intersect, then concatenating path P , edge uv , and path Q gives an M -augmenting path. If P and Q intersect, they must first intersect some vertex $w \in X_{2j}$ for some $j \leq i$, and the cycle containing u, v, w gives us the blossom, with the stem being a path from w back to an open vertex in X_0 .
3. If $v \in X_{2j}$ for $j < i$, then u would have been added to the odd level X_{2j+1} , which is impossible.

4. Finally, v may belong to some previous odd level, which is fine. Observe that this “backward” non-matching edge uv is also an *even-to-odd* edge, like the “forward” edge in the first case.

Now for the edges out of the odd layers considered in line 9.3.

Given $u \in X_{2i+1}$ and *matching* edge $uv \in M$, the cases are:

1. If v is not in $X_{\leq 2i+1}$ then add it to X_{2i+2} . Notice that v cannot be in X_{2i+2} already, since nodes in even layers are matched to nodes in the preceding odd layer, and there cannot be two matching edges incident to v .

Again, observe inductively that v has a path to some vertex in X_0 that hits each intermediate layer once.

2. If v is in X_{2i+1} , there is an matching edge linking two vertices in the same odd level. This gives an augmenting path or a blossom, as in case 2 above. (Success!)
3. The node v cannot be in a previous level, because all those vertices are either open, or are matched using other edges.

Observe that if the algorithm does not succeed, all the matching edges we explored are *odd-to-even*, whereas all the non-matching edges are *even-to-odd*. Now we can prove Lemma 6.14.

Proof of Lemma 6.14. Let P be an M -augmenting path in G . For a contradiction, suppose we do not succeed in finding an augmenting path or blossom. Starting from one of the endpoints of P (which is in X_0 , an even level), trace the path in the leveled graph created above. The next vertex should be in an odd level, the next in an even level, and so forth. Since the path P is alternating, FindAugPath ensures that all its edges will be explored. (Make sure you see this!) Now P has an odd number of edges (i.e., even number of vertices), so the last vertex has an opposite parity from the starting vertex. But the last vertex is open, and hence in X_0 , an even level. This is a contradiction. \square

6.4.3 Finding a Tutte-Berge Set*

If FindAugPath did not succeed, all the edges we explored form a bipartite graph. This does not mean that the entire graph is bipartite, of course—there can be non-matching edges incident to nodes in odd levels that lead to nodes that remain unmarked. But these components have no open vertices (which are all in X_0 and marked). Now define $U = X_{\text{odd}} := X_1 \cup X_3 \cup \dots$ be the vertices in odd levels. Since there are no cross edges, each of these nodes has a distinct

matching edge leading to the next level. Now $G \setminus U$ has two kinds of components:

- (a) the marked vertices in the even levels, X_{even} which are all singletons since there are no cross edges, and
- (b) the unmarked components, which have no open vertices, and hence have even size.

Hence

$$\begin{aligned} \frac{n + |U| - \text{odd}(G \setminus U)}{2} &= \frac{n + |X_{\text{odd}}| - |X_{\text{even}}|}{2} \\ &= \frac{2|X_{\text{odd}}| + (n - |X|)}{2} \\ &= |X_{\text{odd}}| + \frac{(n - |X|)}{2} = |M|. \end{aligned}$$

The last equality uses that all nodes in $V \setminus X$ are perfectly matched among themselves, and all nodes in X_{odd} are matched using unique edges.

The last piece is to show that a Tutte-Berge set U' for a contracted graph $G' = G/B$ with respect to $M' = M/B$ can be lifted to one for G with respect to M . We leave it as an exercise to show that adding the entire blossom B to U' gives such an U .

6.5 Subsequent Work

The best runtime of combinatorial algorithms for maximum matching in general graphs is $O(m\sqrt{n})$ by an algorithm of Silvio Micali and Vijay Vazirani³. The algorithm is based on finding augmenting paths much faster; it is quite involved, though a recent paper of Vijay Vazirani⁴ giving a more approachable explanation. In a later chapter, we will see a very different “algebraic” algorithm based on fast matrix multiplication. This algorithm due to Marcin Mucha and Piotr Sankowski gives a runtime of $O(n^\omega)$, where $\omega \approx 2.376$. Coming up next, however, is a discussion of weighted versions of matching, where edges have weights and the goal is to find the matching of maximum weight.

3

4

Mucha and Sankowski (2006)