Chapter 31

Matchings II

By Sariel Har-Peled, November 1, 2015[①] Version: 1.01

31.1. Maximum Weight Matchings in A Bipartite Graph

31.1.1. On the structure of the problem

For an alternating path/cycle π, its ***weight***, in relation to a matching M, is

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \cap M} w(e). \tag{31.1}$$

Namely, it is the total weight of the free edges in π minus the weight of the matched edges. This is a natural concept because of the following lemma.

Lemma 31.1.1. *Let M be a matching, and let π be an alternating path/cycle with positive weight relative to M; that is $\gamma(\pi, M) > 0$. Furthermore, assume that*

$$M' = M \oplus \pi = (M \setminus \pi) \cup (\pi \setminus M)$$

is a matching. Then $w(M')$ is bigger; namely, $w(M') > w(M)$.

Proof: We have that

$$\begin{aligned} w(M') - w(M) &= \sum_{e \in M'} w(e) - \sum_{e \in M} w(e) = \sum_{e \in M' \setminus M} w(e) - \sum_{e \in M \setminus M'} w(e) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in M \setminus \pi} w(e) \\ &= \gamma(\pi, M). \end{aligned}$$

Just observe that $w(M') = w(M) + \gamma(\pi, M)$. ■

Definition 31.1.2. An alternating path is ***augmenting*** if it starts and ends in a free vertex.

Observation 31.1.3. *If M has an augmenting path π then M is not of maximum size matching (this is for the unweighted case), since $M \oplus \pi$ is a larger matching.*

Theorem 31.1.4. *Let M be a matching of maximum weight among matchings of size $|M|$. Let π be an augmenting path for M of maximum weight, and let T be the matching formed by augmenting M using π. Then T is of maximum weight among matchings of size $|M| + 1$.*

Proof: Let S be a matching of maximum weight among all matchings with $|M| + 1$ edges. And consider $H = (V, M \oplus S)$.

Consider a cycle σ in H. The weight $\gamma(\sigma, M)$ (see Eq. (31.1)) must be zero. Indeed, if $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ is a matching of the same size as M which is heavier than M. A contradiction to the definition of M as the maximum weight such matching.

Similarly, if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.

By the same argumentation, if σ is a path of even length in the graph H then $\gamma(\sigma, M) = 0$ by the same argumentation.

Let U_S be all the odd length paths in H that have one edge more in S than in M, and similarly, let U_M be the odd length paths in H that have one edge more of M than an edge of S.

We know that $|U_S| - |U_M| = 1$ since S has one more edge than M. Now, consider a path $\pi \in U_S$ and a path $\pi' \in U_M$. It must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$. Indeed, if $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ would have bigger weight than M while having the same number of edges. Similarly, if $\gamma(\pi, M) + \gamma(\pi', M) < 0$ (compared to M) then $S \oplus \pi \oplus \pi'$ would have the same number of edges as S while being a heavier matching. A contradiction.

Thus, $\gamma(\pi, M) + \gamma(\pi', M) = 0$. Thus, we can pair up the paths in U_S to paths in U_M, and the total weight of such a pair is zero, by the above argumentation. There is only one path μ in U_S which is not paired, and it must be that $\gamma(\mu, M) = w(S) - w(M)$ (since everything else in H has zero weight as we apply it to M to get S).

This establishes the claim that we can augment M with a single path to get a maximum weight matching of cardinality $|M| + 1$. Clearly, this path must be the heaviest augmenting path that exists for M. Otherwise, there would be a heavier augmenting path σ' for M such that $w(M \oplus \sigma') > w(S)$. A contradiction to the maximality of S. ∎

The above theorem imply that if we always augment along the maximum weight augmenting path, than we would get the maximum weight matching in the end.

31.2. Maximum Weight Matchings in A Bipartite Graph

Let $G = (L \cup R, E)$ be the given bipartite graph, with $w : E \to \mathbb{R}$ be the non-negative weight function. Given a matching M we define the graph G_M to be the directed graph, where if $rl \in M$, $l \in L$ and $r \in R$ then we add (r, l) to $E(G_M)$ with weight $\alpha((r, l)) = w(rl)$. Similarly, if $rl \in E \setminus M$ then add the edge $(l \to r) \in E(G_M)$ to G_M and set $\alpha((l, r)) = -w(rl)$

Namely, we direct all the matching edges from right to left, and assign them their weight, and we direct all other edges from left to right, with their negated weight. Let G_M denote the resulting graph.

An augmenting path π in G must have an odd number of edges. Since G is bipartite, π must have one endpoint on the left side and one endpoint on the right side. Observe, that a path π in G_M has weight $\alpha(\pi) = -\gamma(\pi, M)$.

Let U_L be all the unmatched vertices in L and let U_R be all the unmatched vertices in R.

Thus, what we are looking for is a path π in G_M starting U_L going to U_R with maximum weight $\gamma(\pi)$, namely with minimum weight $\alpha(\pi)$.

Lemma 31.2.1. *If M is a maximum weight matching with k edges in G, than there is no negative cycle in G_M where $\alpha(\cdot)$ is the associated weight function.*

Proof: Assume for the sake of contradiction that there is a cycle C, and observe that $\gamma(C) = -\alpha(C) > 0$. Namely, $M \oplus C$ is a new matching with bigger weight and the same number of edges. A contradiction to the maximality of M. ∎

The algorithm. So, we now can find a maximum weight in the bipartite graph G as follows: Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.

Thus, we need to find a minimum weight path in G_M between U_L and U_R (because we flip weights). This is just computing a shortest path in the graph G_M which does not have negative cycles, and this can just be done by using the **Bellman-Ford** algorithm. Indeed, collapse all the vertices of U_L into a single vertex, and all the uncovered vertices of U_R into a single vertex. Let H_M be the resulting graph. Clearly, we are looking for the shortest path between the two vertices corresponding to U_L and U_R in H_M and since this graph has no negative cycles, this can be done using the **Bellman-Ford** algorithm, which takes $O(nm)$ time. We conclude:

Lemma 31.2.2. *Given a bipartite graph G and a maximum weight matching M of size k one can find a maximum weight augmenting path for G in $O(nm)$ time, where n is the number of vertices of G and m is the number of edges.*

We need to apply this algorithm $n/2$ times at most, as such, we get:

Theorem 31.2.3. *Given a weight bipartite graph G, with n vertices and m edges, one can compute a maximum weight matching in G in $O(n^2 m)$ time.*

31.2.1. Faster Algorithm

It turns out, in fact, that the graph here is very special, and one can use the Dijkstra algorithm. We omit any further details, and just state the result. The interested student can figure out the details (warning: this is not easy).

Theorem 31.2.4. *Given a weight bipartite graph G, with n vertices and m edges, one can compute a maximum weight matching in G in $O(n(n \log n + m))$ time.*

31.3. The Bellman-Ford Algorithm - A Quick Reminder

The **Bellman-Ford** algorithm computes the shortest path from a single source s in a graph G that has no negative cycles to all the vertices in the graph. Here G has n vertices and m edges. The algorithm works by initializing all distances to the source to be ∞ (formally, for all $u \in \mathsf{V}(\mathsf{G})$, we set $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$). Then, it n times scans all the edges, and for every edge $(u, v) \in \mathsf{E}(\mathsf{G})$ it performs a **Relax**(u, v) operation. The relax operation checks if $x = d[u] + w((u, v)) < d[v]$, and if so, it updates $d[v]$ to x, where $d[u]$ denotes the current distance from s to u. Since **Relax**(u, v) operation can be performed in constant time, and we scan all the edges n times, it follows that the overall running time is $O(mn)$.

We claim that in the end of the execution of the algorithm the shortest path length from s to u is $d[u]$, for all $u \in \mathsf{V}(\mathsf{G})$. Indeed, every time we scan the edges, we set at least one vertex distance to its final value (which is its shortest path length). More formally, all vertices that their shortest path to s have i edges, are being set to their shortest path length in the ith iteration of the algorithm, as can be easily proved by induction. This implies the claim.

Notice, that if we want to detect negative cycles, we can ran **Bellman-Ford** for an additional iteration. If the distances changes, we know that there is a negative cycle somewhere in the graph.

31.4. Maximum Size Matching in a Non-Bipartite Graph

The results from the previous lecture suggests a natural algorithm for computing a maximum size (i.e., matching with maximum number of edges in it) matching in a general (i.e., not necessarily bipartite) graph. Start from an empty matching M and repeatedly find an augmenting path from an unmatched vertex to an unmatched vertex. Here we are discussing the unweighted case.

Notations. Let $\mathcal{T}$ be a given tree. For two vertices $x, y \in V(\mathcal{T})$, let τ_{xy} denote the path in $\mathcal{T}$ between x and y. For two paths π and π' that share an endpoint, let $\pi \parallel \pi'$ denotes the path resulting from concatenating π to π'. For a path π, let $|\pi|$ denote the number of edges in π.

31.4.1. Finding an augmenting path

We are given a graph G and a matching M, and we would to compute a bigger matching in G. We will do it by computing an augmenting path for M.

We first observe that if G has any edge with both endpoints being free, we can just add it to the current matching. Thus, in the following, we assume that for all edges, at least one of their endpoint is covered by the current matching M. Our task is to find an augmenting path in M.

We start by collapsing the unmatched vertices to a single vertex s, and let H be the resulting graph. Next, we compute an ***alternating BFS*** of H starting from s. Formally, we perform a BFS on H starting from s such that for the even levels of the tree the algorithm is allowed to traverse only edges in the matching M, and in odd levels the algorithm traverses the unmatched edges. Let $\mathcal{T}$ denote the resulting tree.

An augmenting path in G corresponds to an odd cycle in H with passing through the vertex s.

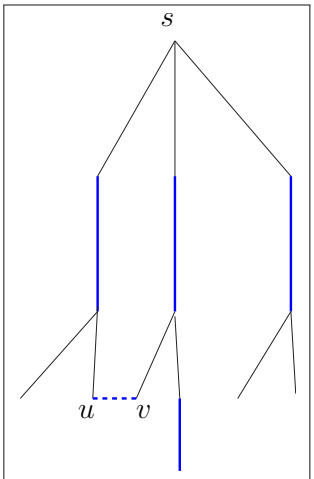


Figure 31.1: A cycle in the alternating BFS tree.

Definition 31.4.1. An edge $uv \in E(G)$ is a ***bridge*** if the following conditions are met: (i) u and v have the same depth in $\mathcal{T}$, (ii) if the depth of u in $\mathcal{T}$ is even then uv is free (i.e., $uv \notin M$, and (iii) if the depth of u in $\mathcal{T}$ is odd then $uv \in M$.

Note, that given an edge uv we can check if it is a bridge in constant time after linear time preprocessing of $\mathcal{T}$ and G.

The following is an easy technical lemma.

Lemma 31.4.2. *Let v be a vertex of G, M a matching in G, and let π be the shortest alternating path between s and v in G. Furthermore, assume that for any vertex w of π the shortest alternating path between w and s is the path along π.*

Then, the depth $d_{\mathcal{T}}(v)$ of v in $\mathcal{T}$ is $|\pi|$.

Proof: By induction on $|\pi|$. For $|\pi| = 1$ the proof trivially holds, since then v is a neighbor of s in G, and as such it is a child of s in $\mathcal{T}$.

For $|\pi| = k$, consider the vertex just before v on π, and let us denote it by u. By induction, the depth of u in $\mathcal{T}$ is $k - 1$. Thus, when the algorithm computing the alternating BFS visited u, it tried to hang v from it in the next iteration. The only possibility for failure is if the algorithm already hanged v in earlier iteration of the algorithm. But then, there exists a shorter alternating path from s to v, which is a contradiction. ■

Lemma 31.4.3. *If there is an augmenting path in G for a matching M, then there exists an edge $uv \in E(G)$ which is a bridge in $\mathcal{T}$.*

Proof: Let π be an augmenting path in G. The path π corresponds to a an odd length alternating cycle in H. Let σ be the shortest odd length alternating cycle in G (note that both edges in σ that are adjacent to s are unmatched).

For a vertex x of σ, let $d(x)$ be the length of the shortest alternating path between x and s in H. Similarly, let $d'(x)$ be the length of the shortest alternating path between s and x along σ. Clearly, $d(x) \leq d'(x)$, but we claim that in fact $d(x) = d'(x)$, for all $x \in \sigma$. Indeed, assume for the sake of contradiction that $d(x) < d'(x)$, and let π_1, π_2 be the two paths from x to s formed by σ. Let η be the shortest alternating path between s and x. We know that $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$. It is now easy to verify that either $\pi_1 \parallel \eta$ or $\pi_2 \parallel \eta$ is an alternating cycle shorter than σ involving s, which is a contradiction.

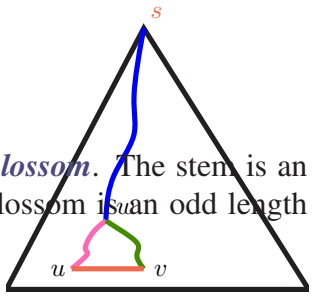
But then, take the two vertices of σ furthest away from s. Clearly, both of them have the same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$. By Lemma 31.4.2, we now have that $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Establishing the first part of the claim. See Figure 31.1.

As for the second claim, observe that it easily follows as σ is created from an alternating path. ∎

Thus, we can do the following: Compute the alternating BFS $\mathcal{T}$ for H, and find a bridge uv in it. If M is not a maximal matching, then there exists an augmenting path for G and by Lemma 31.4.3 there exists a bridge. Computing the bridge uv takes $O(m)$ time.

Extract the paths from s to u and from s to v in $\mathcal{T}$, and glue them together with the edge uv to form an odd cycle μ in H; namely, $\mu = \tau_{su} \parallel uv \parallel \tau_{vs}$. If μ corresponds to an alternating path in G then we are done, since we found an alternating path, and we can apply it and find a bigger matching.

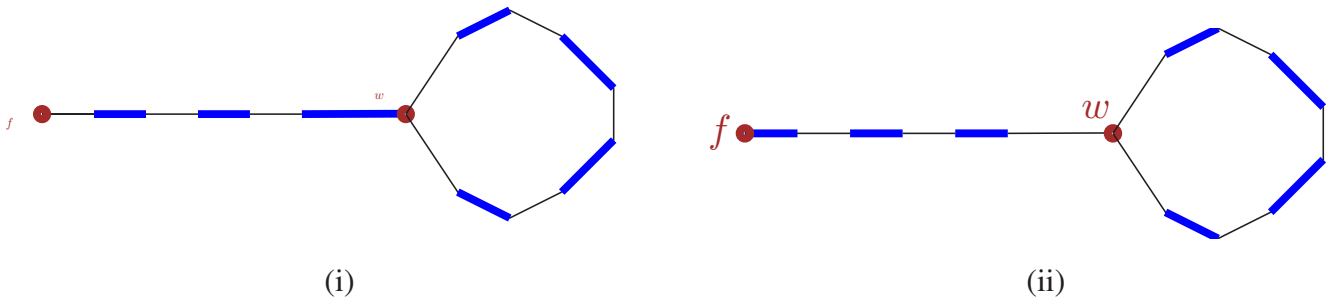
But μ, in fact, might have common edges. In particular, let π_{su} and π_{sv} be the two paths from s to u and v, respectively. Let w be the lowest vertex in $\mathcal{T}$ that is common to both π_{su} and π_{sv}.



Definition 31.4.4. Given a matching M, a ***flower*** for M is formed by a ***stem*** and a ***blossom***. The stem is an even length alternating path starting at a free vertex v ending at vertex w, and the blossom is an odd length (alternating) cycle based at w.

Lemma 31.4.5. *Consider a bridge edge $uv \in G$, and let w be the least common ancestor (LCA) of u and v in $\mathcal{T}$. Consider the path π_{sw} together with the cycle $C = \pi_{wu} \parallel uv \parallel \pi_{vw}$. Then π_{sw} and C together form a flower.*

Proof: Since only the even depth nodes in $\mathcal{T}$ have more than one child, w must be of even depth, and as such π_{sw} is of even length. As for the second claim, observe that $\alpha = |\pi_{wu}| = |\pi_{wv}|$ since the two nodes have the same depth in $\mathcal{T}$. In particular, $|C| = |\pi_{wu}| + |\pi_{wv}| + 1 = 2\alpha + 1$, which is an odd number. ∎



(i) (ii)

Figure 31.3: (i) the flower, and (ii) the invert stem.

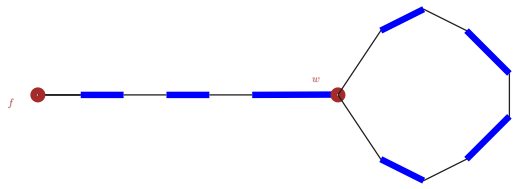


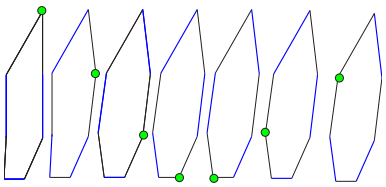
Figure 31.2: A blossom is made out of a ***stem*** (the path fw), and an odd length cycle which is the blossom. Together they form a flower.

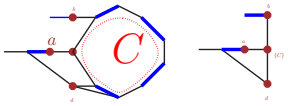
Let us translate this blossom of H back to the original graph G. The path s to w corresponds to an alternating path starting at a free vertex f (of G) and ending at w, where the last edge is in the stem is in the matching, the cycle $w \ldots u \ldots v \ldots w$ is an alternating odd length cycle in G where the two edges adjacent to w are unmatched.

We can not apply a blossom to a matching in the hope of getting better matching. In fact, this is illegal and yield something which is not a matching. On the positive side, we discovered an odd alternating cycle in the graph G. Summarizing the above algorithm, we have:

Lemma 31.4.6. *Given a graph G with n vertices and m edges, and a matching M, one can find in $O(n+m)$ time, either a blossom in G or an augmenting path in G.*

To see what to do next, we have to realize how a matching in G interact with an odd length cycle which is computed by our algorithm (i.e., blossom). In particular, assume that the free vertex in the cycle is unmatched. To get a maximum number of edges of the matching in the cycle, we must at most $(n-1)/2$ edges in the cycle, but then we can rotate the matching edges in the cycle, such that any vertex on the cycle can be free. See figure on the right.





Let G/C denote the graph resulting from collapsing such an odd cycle C into single vertex. The new vertex is marked by $\{C\}$.

Lemma 31.4.7. *Given a graph G, a matching M, and a flower B, one can find a matching M' with the same cardinality, such that the blossom of B contains a free (i.e., unmatched) vertex in M'.*

Proof: If the stem of B is empty and B is just formed by a blossom, and then we are done. Otherwise, B was as stem π which is an even length alternating path starting from from a free vertex v. Observe that the matching $M' = M \oplus \pi$ is of the same cardinality, and the cycle in B now becomes an alternating odd cycle, with a free vertex.

Intuitively, what we did is to apply the stem to the matching M. See Figure 31.3. ∎

Theorem 31.4.8. *Let M be a matching, and let C be a blossom for M with an unmatched vertex v. Then, M is a maximum matching in G if and only if $M/C = M \setminus C$ is a maximum matching in G/C.*

Proof: Let G/C be the collapsed graph, with $\{C\}$ denoting the vertex that correspond to the cycle C.

Note, that the collapsed vertex $\{C\}$ in G/C is free. Thus, an augmenting path π in G/C either avoids the collapsed vertex $\{C\}$ altogether, or it starts or ends there. In any case, we can rotate the matching around C such that π would be an augmenting path in G. Thus, if M/C is not a maximum matching in G/C then there exists an augmenting path in G/C, which in turn is an augmenting path in G, and as such M is not a maximum matching in G.

Similarly, if π is an augmenting path in G and it avoids C then it is also an augmenting path in G/C, and then M/C is not a maximum matching in G/C.

Otherwise, since π starts and ends in two different free vertices and C has only one free vertex, it follows that π has an endpoint outside C. Let v be this endpoint of π and let u be the first vertex of π that belongs to C. Let σ be the path $\pi[v, u]$.

Let f be the free vertex of C. Note that f is unmatched. Now, if $u = f$ we are done, since then π is an augmenting path also in G/C. Note that if u is matched in C, as such, it must be that the last edge e in π is unmatched. Thus, rotate the matching M around C such that u becomes free. Clearly, then σ is now an augmenting path in G (for the rotated matching) and also an augmenting path in G/C. ■

Corollary 31.4.9. *Let M be a matching, and let C be an alternating odd length cycle with the unmatched vertex being free. Then, there is an augmenting path in G if and only if there is an augmenting path in G/C.*

31.4.2. The algorithm

Start from the empty matching M in the graph G.

Now, repeatedly, try to enlarge the matching. First, check if you can find an edge with both endpoints being free, and if so add it to the matching. Otherwise, compute the graph H (this is the graph where all the free vertices are collapsed into a single vertex), and compute an alternating BFS tree in H. From the alternating BFS, we can extract the shortest alternating cycle based in the root (by finding the highest bridge). If this alternating cycle corresponds to an alternating path in G then we are done, as we can just apply this alternating path to the matching M getting a bigger matching.

If this is a flower, with a stem ρ and a blossom C then apply the stem to M (i.e., compute the matching $M \oplus \rho$). Now, C is an odd cycle with the free vertex being unmatched. Compute recursively an augmenting path π in G/C. By the above discussing, we can easily transform this into an augmenting path in G. Apply this augmenting path to M.

Thus, we succeeded in computing a matching with one edge more in it. Repeat till the process get stuck. Clearly, what we have is a maximum size matching.

31.4.2.1. Running time analysis

Every shrink cost us $O(m + n)$ time. We need to perform $O(n)$ recursive shrink operations till we find an augmenting path, if such a path exists. Thus, finding an augmenting path takes $O(n(m + n))$ time. Finally, we have to repeat this $O(n)$ times. Thus, overall, the running time of our algorithm is $O(n^2(m + n)) = O(n^4)$.

Theorem 31.4.10. *Given a graph G with n vertices and m edges, computing a maximum size matching in G can be done in $O\left(n^2 m\right)$ time.*

31.5. Maximum Weight Matching in A Non-Bipartite Graph

This the hardest case and it is non-trivial to handle. There are known polynomial time algorithms, but I feel that they are too involved, and somewhat cryptic, and as such should not be presented in class. For the interested

student, a nice description of such an algorithm is presented in

Combinatorial Optimization - Polyhedral and efficiency by Alexander Schrijver Vol. A, 453–459.

The description above also follows loosely the same book.