**CS 860: Modern Topics in Graph Algorithms** **University of Waterloo: Winter 2024**

# Lecture 2: Maximum Matchings

January 22, 2023

*Instructor: Sepehr Assadi*



## Topics of this Lecture



# 1 Maximum Matchings

In the last lecture, we examined the minimum spanning tree problem and saw a randomized linear (expected) time algorithm for it. We now switch to studying another fundamental combinatorial optimization problem: the *maximum matching* problem. As before, you have most likely seen this problem in your previous algorithms courses. We will provide a quick review in the following.

Let $G = (V, E)$ be an undirected graph. A *matching* $M$ in $G$ is any collection of vertex-disjoint edges, namely, a subgraph of $G$ with maximum degree at most one.

**Problem 1** (**Maximum Matching**)**.** The maximum matching problem is defined as follows: Given a graph $G = (V, E)$, find a matching $M$ of $G$ with maximum number of edges.

We note that in many applications of the maximum matching problem, we can additionally assume the input graph is *bipartite*, i.e., its vertices can be partitioned into two independent sets. We refer to this special case as the **bipartite matching** problem. In general, the bipartite matching problem tends to be an "algorithmically simpler" version of the problem compared to the general case, in that bipartite matching algorithms, for the most part, are considerably simpler than the algorithms for the general case[1].

[1]Although interestingly, from a "complexity point of view", the two problems are typically equivalent to each other, as in,

### 1.1 Basics of Maximum Matching Algorithms

We start by reviewing some basic background on maximum matching algorithms. A key concept when studying matching algorithms is that of *augmenting paths* (quite similar to the same notion for maximum flow algorithms).

**Definition 1.** For any matching $M$ in a graph $G = (V, E)$, an **augmenting path** $P$ is any path in $G$ starting from a vertex unmatched by $M$, alternatively taking edges in $E \setminus M$ followed by $M$ and continuing like this until ending in another unmatched vertex of $M$.

A basic fact about augmenting paths is that we can always increase size of a matching $M$ if we have an augmenting path $P$ for it (see Figure 1 for an illustration).

**Fact 2.** *Let $M$ be any matching in $G$ and $P$ be an augmenting path for $M$. Then, the set $M \triangle P$, namely, the symmetric difference of $M$ and $P$, is a matching of size one larger than $M$.*

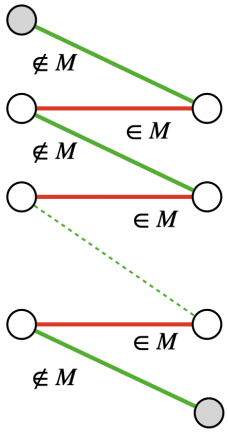



Figure 1: An illustration of the augmenting paths. The gray vertices are unmatched, the green edges are not-matching edges and the red ones are matching edges. By switching red edges to green edges in our matching, we can increase its size by one.

We can also show that as long as $M$ is *not* a maximum matching, it always admits an augmenting path.

**Proposition 3.** *Let $M$ be any matching in $G$ which is not a maximum matching. Then, there exists some augmenting path $P$ for $M$ in $G$.*

*Proof.* Let $M^*$ be a maximum matching in $G$. Consider the subgraph $H$ of $G$ on edges $M \cup M^*$. Every vertex in $H$ has maximum degree at most two and thus $H$ is a vertex-disjoint union of paths and cycles. Moreover, since $H$ is a union of two matchings, all cycles in $H$ should be of even length (because $H$ is 2-edge-colorable). Any even-cycle in $H$ has the same number of edges in $M$ and $M^*$. Thus, since $|M^*| > |M|$ by our assumption, there should be at least one path in $H$ that has more edges from $M^*$ compared to $M$. Such a path is now an augmenting path for $M$ (in $G$), concluding the proof. □

## 2 Augmenting Path Algorithms for Maximum Matching

Fact 2 and Proposition 3 gives us a simple recipe for designing maximum matching algorithms:

the results in the bipartite case can often be extended to the general case with "considerable additional work" (although there are notable exceptions as well).

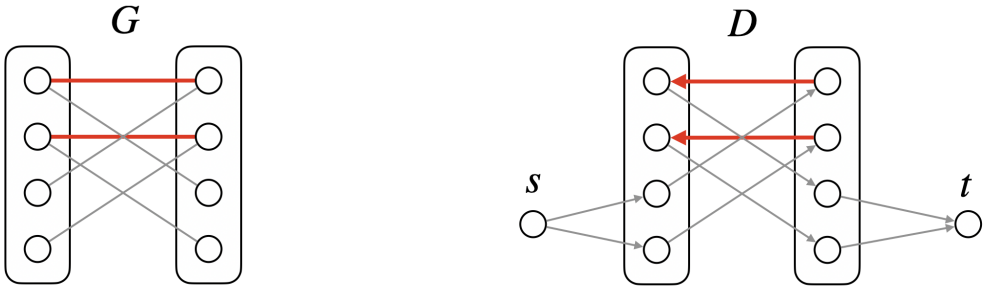
**Algorithm 1.**

1. Start with an empty matching $M = \emptyset$;
2. While $M$ is not a maximum matching:
   (a) Find an augmenting path $P$ for $M$ which always exist by Proposition 3;
   (b) Use Fact 2 to increase the size of $M$ using $P$ by updating $M \leftarrow M \triangle P$.

We now show how to implement this strategy for bipartite and general graphs separately.

## 2.1 Bipartite Graphs

Implementing Algorithm 1 for bipartite graphs is quite easy. Suppose $G = (L, R, E)$ is a bipartite graph with bi-partition $L, R$ of vertices and $M$ is some matching in $G$. We create the following directed graph (digraph) $D$ from $G$ and $M$. Orient all edges of $M$ from $R$ to $L$ and all other edges from $L$ to $R$. Add two more vertices $s$ and $t$ to $D$ where $s$ is connected to all unmatched vertices in $L$ and $t$ has an edge from all unmatched vertices in $R$. See Figure 2 for an illustration.




(a) A bipartite graph $G$ and a matching $M$.

(b) The corresponding digraph $D$ of $G$ and $M$

Figure 2: An illustration of the digraph created for finding augmenting paths in bipartite graphs.

**Proposition 4.** *Let $G$ be a bipartite graph and $D$ be the digraph obtained from it as described above. Suppose $M$ has an augmenting path in $G$. Then, there is a path from $s$ to $t$ in $D$. Conversely, any path from $s$ to $t$ in $D$ contains an augmenting path for $M$ in $G$.*

*Proof.* The proof of both statements is straightforward. Let $P = (u_1, u_2, \ldots, u_k)$ be an augmenting path for $M$ in $G$. Then, the path $(s, u_1, u_2, \ldots, u_k, t)$ is a path in $D$ since for $i \geqslant 1$, $(u_{2i-1}, u_{2i})$ is connected via $E \setminus M$ in $P$ (and thus the corresponding directed edge exists in $D$) and $(u_{2i}, u_{2i+1})$ is connected via $M$ in $P$ (and thus, again, the corresponding directed edge exists in $D$).

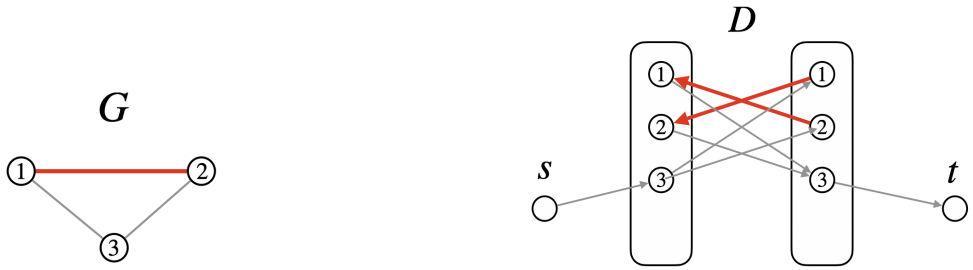
For the other direction, any $s$-$t$ path in $D$ (ignoring the edges out of $s$ and coming to $t$) alternates between using edges from $E \setminus M$ and $M$ and thus is an augmenting path for $M$ in $G$. □

Proposition 4 suggests that finding an augmenting path $P$ for a matching $M$ in a *bipartite* graph $G$ can be simply done using a DFS/BFS (or any other standard graph search algorithm). In particular, this means that we can implement each iteration of Algorithm 1 in $O(m)$ time. As we can also find at most $n$ augmenting paths, this results in a simple $O(m \cdot n)$ time algorithm for bipartite matching.

## 2.2 General (Non-Bipartite) Graphs

Finding augmenting paths in general graphs however is considerably more tricky. This is, intuitively speaking, because of the following: to find an augmenting path in bipartite graphs, we always know *which direction* to traverse a matching edge (which is dictated by the bipartition of the input graph); this is no longer true in general graphs – an augmenting path may only exist if we traverse matching edges with a particular direction but we simply do not know these directions and cannot simply "guess" them[2].

A simple "fix" given that we do not know which direction to traverse matching edges is to include *both* directions when finding augmenting paths. Suppose $G = (V, E)$ is an arbitrary (not necessarily bipartite) graph and $M$ is some matching in $G$. We create the following directed graph (digraph) $D$ from $G$ and $M$. We create *two* copies of all vertices in two sets $V_L$ and $V_R$. We use $v_L$ and $v_R$ to refer to the copy of $v$ in $V_L$ and $V_R$, respectively. Then, for any edge $(u, v) \in E$, we add two edges $(u_L, v_R)$ and $(u_R, v_L)$ to $D$ (the graph created so far is called the **bipartite double cover** of $G$). We then orient both copies of each edge of $M$ from $V_R$ to $V_L$ and all the remaining edges from $V_L$ to $V_R$. Finally, we add two new vertices $s$ and $t$ and connect $s$ to all unmatched vertices in $V_L$ and connect all unmatched in $V_R$ to $t$. See Figure 3.




(a) A general graph $G$ and a matching $M$.

(b) The corresponding digraph $D$ of $G$ and $M$

Figure 3: An illustration of the digraph created for finding augmenting paths in bipartite graphs. Notice that even though there is a path from $s$ to $t$ in $D$, this does not mean $M$ admits an augmenting path in $G$.

It is easy to see—using exactly the same analysis as in Proposition 4—that having an augmenting path for $M$ in $G$ implies that $s$ can reach $t$ in $D$ as well. However, the converse is no longer true here: not every $s$-$t$ path in $D$ can be turned into an augmenting path for $M$ in $G$ (see Figure 3 for an example).

A brilliant fix to this problem was discovered by Edmonds in his famous paper "Paths, trees, and flowers" [Edm65]. We will go over this algorithm—called Edmonds' *Blossom algorithm*—after a quick detour.

**Remark.** Edmonds paper [Edm65] put forward the notion of *polynomial-time* algorithms (and the complexity class P) as the "most natural" notion of efficiency for algorithms and the rest is history. You are strongly encouraged to check Section 2 of this paper for a detailed account of Edmonds' opinion on polynomial-time algorithms as a measure of practicality. The following quote from this paper is also quite relevant (although somewhat overlooked):

> "It would be unfortunate for any rigid criterion to inhibit the practical development of algorithms which are either not known or known not to conform nicely to the criterion. Many of the best algorithmic ideas known today would suffer by such theoretical pedantry."

**Flowers and Blossoms.** We now get to describe Edmond's fix. Consider a path $(s, u_1, u_2, \ldots, u_k, t)$ in $D$. If all of $u_1, \ldots, u_k$ correspond to *distinct* vertices in $G$, then, exactly as in Proposition 4, we also obtain

[2] That being said, a beautiful algorithm due to McGregor [McG05] for $(1-\varepsilon)$-*approximation* of maximum matchings in $(1/\varepsilon)^{O(1/\varepsilon)} \cdot O(m)$ time precisely achieves this using randomization; but, this runtime is too slow in terms of dependence on $\varepsilon$.

an augmenting path for $M$ in $G$. However, it is possible that these vertices are not all distinct. In that case, let $u_j$ be the first repeated vertex and let $u_i$ for $i < j$ be the copy of this vertex visited previously. Consider the sequence of vertices $u_1, u_2, \ldots, u_i, \ldots, u_j (= u_i)$ in $G$ (here, with a slight abuse of notation, we "reverted back" the vertices in $D$ to their copies in $G$). This sequence forms what Edmonds called an ***M*-flower** as defined below. See Figure 4 for an example.

**Definition 5.** Let $G = (V, E)$ be any graph and $M$ be a matching in $G$. An ***M*-flower** in $G$ (with respect to $M$) is a sequence of vertices $(u_1, u_2, \ldots, u_k)$ with the following properties:

1. For any $i \geqslant 1$, $(u_{2i-1}, u_{2i})$ is an edge in $E \setminus M$ and $(u_{2i}, u_{2i+1})$ is an edge in $M$.
2. $k$ is even and the vertex $u_k$ is the same as $u_{2i^*+1}$ for some $i^* \geqslant 0$.

The path $(u_1, \ldots, u_{2i^*+1})$ (which might be empty) is called the **stem** of the $M$-flower and the *odd-cycle* $(u_{2i^*+1}, \ldots, u_k)$ (which is never empty) is called the **blossom** of the $M$-flower. The vertex $u_{2i^*+1}$ is also called the **base** of the blossom.

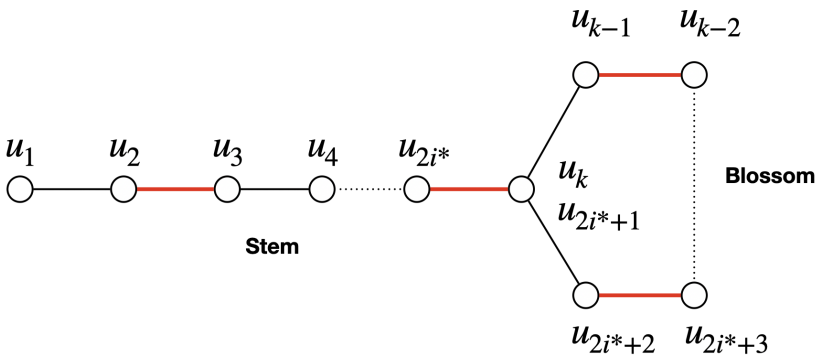



Figure 4: An illustration of an $M$-flower with its stem and blossom. Here, the thick (red) edges are in the matching $M$ and remaining edges are in $E \setminus M$.

**Proposition 6.** *Let $G$ be an arbitrary (not necessarily bipartite) graph and $D$ be the digraph obtained from it as described above. Suppose $M$ has an augmenting path in $G$. Then, there is a path from $s$ to $t$ in $D$. On the other hand, any path from $s$ to $t$ in $D$ contains either an augmenting path for $M$ or an $M$-flower in $G$.*

*Proof.* The proof is identical to that of Proposition 4 for bipartite graphs except for the following case: the path $(s, u_1, u_2, \ldots, u_k, t)$ in $D$ contains two copies of the same vertex in $G$.

In that case, we consider the sequence of vertices $(u_1, \ldots, u_k)$ in $D$ (and, with a slight abuse of notation in $G$ as well) where $u_k$ is the first duplicated vertex. Notice that by the construction of $D$, the edges of $(u_1, \ldots, u_k)$ are alternating between $E \setminus M$ and $M$. Moreover, $k$ needs to be an even number because $u_{2i+1}$ for every $i \geqslant 1$ corresponds to $M(u_{2i})$; so, if $u_{2i+1}$ is duplicated, we already have $u_{2i}$ also duplicated and thus the first index $k$ is always an even number. Finally, because we started with a path in $D$, the edge $(u_k, u_{k+1})$ in the path in $D$ should go in the "opposite" direction of the matching edge compared to the last time we saw this edge (otherwise, we are hitting the same vertex of $D$ twice in a path that cannot happen); thus, $(u_k, u_{k+1})$ should be an odd- to even-index matching edge, which means $u_k = u_{2i^*+1}$ for some $i^* \geqslant 0$.

All in all, these mean that $(u_1, \ldots, u_k)$ form an $M$-flower (Definition 5) in $G$, concluding the proof. □

**Contracting Blossoms.** Notice that a blossom is an odd-cycle and thus cannot contain a perfect matching (namely, a matching that matches all vertices). But, it also has this additional property: for any vertex $v$

in the blossom, we can find a matching that only leaves $v$ unmatched (such a (sub)graph is called **factor-critical** in graph theory). This allows us to *contract* the blossom for now to continue our search for an augmenting path and once we found the path, we can *expand* the blossom to find an augmenting path for the entire graph also. This is the content of the following main lemma.

**Lemma 7.** *Let $G$ be any arbitrary graph, $M$ be a matching in $G$, and $B$ be a blossom for $M$. Let $G/B$ be the graph obtained from $G$ by contracting vertices of $B$ into a single vertex $b$ (and removing self-loops but keeping parallel edges). Define $M/B$ similarly for the matching $M$. Then:*

1. *If $M/B$ is a maximum matching of $G/B$, then, $M$ is also a maximum matching of $G$.*
2. *Conversely, if $M/B$ admits an augmenting path $P$ in $G/B$, then, there is also an augmenting path $Q$ for $M$ in $G$ that can be obtained from $P$ by expanding vertices of $G$.*

*Proof.* We prove each part separately.

1. Suppose $M$ is not a maximum matching of $G$. Consider the matching $M'$ obtained by taking the symmetric difference of $M$ with the edges in the stem of the $M$-flower with blossom $B$. See Figure 5 that corresponds to applying this step to the $M$-flower in Figure 4.

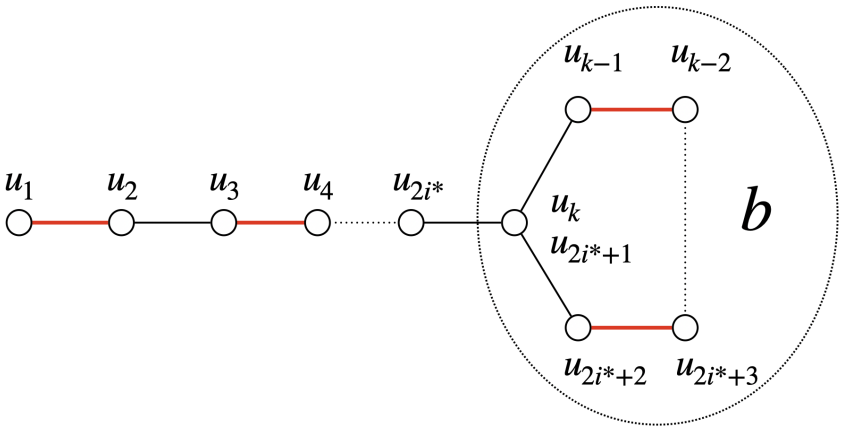
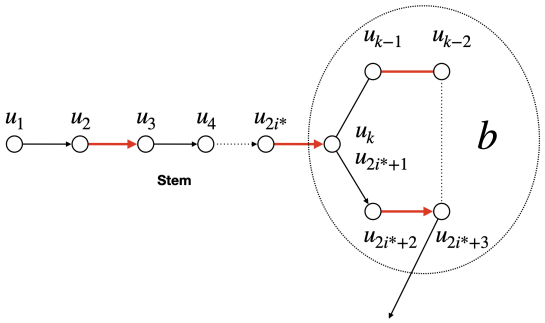



Figure 5: An illustration of the matching $M'$ obtained by applying the alternating path in the step of the $M$-flower with blossom $B$ (to the $M$-flower of Figure 4).

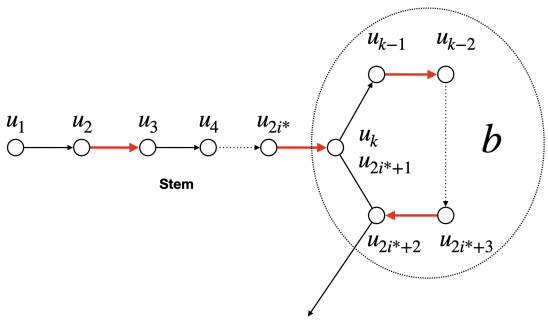
   Notice that $M'$ has the same size as $M$ and is thus also not a maximum matching. Moreover, $M'$ leaves the base of the $M$-blossom $B$ unmatched. Now by Proposition 3, there is an augmenting path $P$ for $M'$ in $G$. Let $(u_1, \ldots, u_k)$ be the vertices of $P$. If none of these vertices are in $B$, then $P$ is also an augmenting path for $M'/B$ in $G/B$. But, $M'/B$ has the same size as $M/B$ which is a maximum matching, thus a contradiction. Otherwise, let $u_j$ be the first vertex in $P$ that belongs to $B$. Then, the path $(u_1, \ldots, u_{j-1}, b)$ is an augmenting path for $M'/B$ in $G/B$ (since $b$ is unmatched by $M'/B$) and thus again a contradiction. This means that $M$ is also a maximum matching.

2. Let $P$ be an augmenting path for $M/B$ in $G/B$ and let $u \in B$ be the vertex in $B$ which is incident on the edge leaving $b$ in $P$ (if no such vertex exist, $P$ is already an augmenting path for $M$ in $G$). Then, if $u$ is an odd-indexed vertex in $B$, then we can create $Q$ by following the same direction as that of vertices in $B$ and taking the edge out of $u$ from $B$. Otherwise, if $u$ is an even-indexed vertex in $B$, we "rotate" the blossom and follow the opposite direction of $B$ until we take the edge out of $u$ from $B$. See Figure 6 for an illustration. In both cases, this leads to an augmenting path $Q$ for $M$ in $G$, concluding the proof.

This concludes the proof of the lemma. □




(a) The edge out of $B$ here respects the ordering of the blossom and we can find the augmenting path with the same order.




(b) The edge out of $B$ here is the opposite of the ordering of the blossom and we can find the augmenting path by following the blossom from the other direction.

Figure 6: An illustration of finding an augmenting path for $M$ in $G$ from an augmenting path for $M/B$ in $G/B$. The blossom corresponds to the one in Figure 4.

**The Blossom Algorithm.** Equipped with Lemma 7 we can now finally implement Algorithm 1 in general graphs as well.

At each step of finding the augmenting path for a matching $M$, we create the digraph $D$ as described earlier and find an $s$-$t$ path in $D$. By Proposition 6, we either find an augmenting path and we will be done for this iteration or we find a $M$-flower and thus we can contract the blossom $B$ and recurse on the matching $M/B$ and $G/B$ instead. Eventually, we either contract the entire graph to a single vertex and find no augmenting path, which, by Lemma 7 means there is no augmenting path for $M$ in $G$ also (inductively). Or, we find an augmenting path and we can expand the blossoms to find an augmenting path in $G$ as well.

Each graph search on $D$ takes $O(m)$ time and each blossom contraction reduces the number of vertices by one at least, so we can only have $O(n)$ blossom contraction steps in each iteration before finding an augmenting path. This takes $O(mn)$ time for finding a single augmenting path and thus $O(mn^2)$ time for computing a maximum matching.

In conclusion, Edmond's Blossom algorithm finds a maximum matching in $O(mn^2)$ time in any arbitrary (not necessarily bipartite) graphs.

**Remark.** The running time of Edmond's blossom algorithm has been improved considerably over the years. Firstly, a relatively easy modification allows us to only spend $O(m + n^2)$ time for finding each single augmenting path by not redoing the entire search from scratch each time. This runtime was further improved to $O(m)$ time by Gabow and Tarjan [GT91]. This line of work leads to an algorithm for general matching with $O(mn)$ time.

The fastest algorithm for general matching is due to Micali and Vazirani [MV80] and achieves $O(m\sqrt{n})$ time (see also [GT91]). You can read more about the history of matching algorithms in [DP14].

# 3 A Primal-Dual Algorithm for Bipartite Matching

We now examine a different way of obtaining an algorithm for *bipartite* matchings that does not involve *directly* computing augmenting paths (although, to some extent, it still does it implicitly). This is obtained via an application of *linear programming (LP)* for *approximating* combinatorial optimization problems. This lecture is self-contained and you do not need to know any background on LPs for the rest of this lecture (although such background can further demystify certain aspects of this algorithm).

### 3.1 Bipartite Matching as an LP

Let $G = (L, R, E)$ be a bipartite graph with the bipartition $L$ and $R$ of vertices. Consider the following *integer linear program (ILP)* that models the maximum matching problem:

$$\begin{aligned} \max_{x} \quad & \sum_{e \in E} x_e \\ \text{subject to} \quad & \forall u \in L: \quad \sum_{e \ni u} x_e \leqslant 1 \\ & \forall v \in R: \quad \sum_{e \ni v} x_e \leqslant 1 \\ & \forall e \in E: \quad x_e \in \{0, 1\}. \end{aligned}$$

In words, we assign each edge $e$ a value $x_e \in \{0, 1\}$ such that no vertex has more than one incident edge with value $x_e = 1$, thus, the set $\{e \mid x_e = 1\}$ is always a matching. By maximizing $\sum_e x_e$ also, we are maximizing the size of the matching picked. Thus, the maximum bipartite matching problem is equivalent to solving the above ILP. Now, to obtain an LP, we simply relax the constraint that $x_e$ needs to be an integer and let it be any value $x_e \geqslant 0$ (namely, allow for picking edges *fractionally* in the matching). This way, we obtain the following LP for bipartite matching.

$$\begin{aligned} \max_{x} \quad & \sum_{e \in E} x_e \\ \text{subject to} \quad & \forall u \in L: \quad \sum_{e \ni u} x_e \leqslant 1 \\ & \forall v \in R: \quad \sum_{e \ni v} x_e \leqslant 1 \\ & \forall e \in E: \quad x_e \geqslant 0. \end{aligned}$$

Now, we will take a quick detour to do a basic "duality" step. Suppose our goal is to simply find an upper bound on the value of this LP. Let us define variables $y_u$ for $u \in L$ and $z_v$ for $v \in R$ such that $y_u, z_v \geqslant 0$ for all $u \in L, v \in R$. By multiplying these variables to their corresponding constraints in the matching LP and summing them up, we obtain,

$$\sum_{u \in L} y_u \cdot \sum_{e \ni u} x_e + \sum_{v \in R} z_v \cdot \sum_{e \ni v} x_e \leqslant \sum_{u \in L} y_u + \sum_{v \in R} z_v,$$

because $y_u$'s and $z_v$'s are non-negative and thus the multiplication preserves the order of the inequalities in the matching LP. But, we can re-write the LHS as:

$$\sum_{u \in L} y_u \cdot \sum_{e \ni u} x_e + \sum_{v \in R} z_v \cdot \sum_{e \ni v} x_e = \sum_{e=(u,v) \in E} (y_u + z_v) \cdot x_e,$$

by re-ordering the terms for each edge. Thus, we obtain the following inequality:

$$\sum_{e=(u,v) \in E} (y_u + z_v) \cdot x_e \leqslant \sum_{u \in L} y_u + \sum_{v \in R} z_v.$$

Finally, suppose we add the additional constraints that for every edge $(u, v) \in E$, we need $y_u + z_v \geqslant 1$. Then, the LHS of the above satisfies

$$\sum_{e=(u,v) \in E} (y_u + z_v) \cdot x_e \geqslant \sum_{e=(u,v) \in E} x_e.$$

Thus, if we find any set of variables $y_u$'s and $z_v$'s such that they are non-negative and for every edge $(u, v) \in E$, they satisfy $y_u + z_v \geqslant 1$, we obtain that for any feasible solution $x$ to the matching LP,

$$\sum_{e \in E} x_e \leqslant \sum_{u \in L} y_u + \sum_{v \in R} z_v.$$

This means that we can write the task of upper bounding the matching LP as the following LP itself (it becomes a minimization because we want to find the best upper bound):

$$\begin{aligned} \min_{y,z \in \mathbb{R}^L \times \mathbb{R}^R} \quad & \sum_{u \in L} y_u + \sum_{v \in R} z_v \\ \text{subject to} \quad & y_u + z_v \geqslant 1 \qquad \forall\ (u,v) \in E, \\ & y_u, z_v \geqslant 0 \qquad \forall\ u \in L, v \in R. \end{aligned}$$

This LP is called the **dual** of the matching LP (and we refer to the matching LP in this context as the **primal** LP).

Finally, suppose we further change this LP to an ILP by requiring each $y_u \in \{0,1\}$ and $z_v \in \{0,1\}$; then, we are trying to find the minimum number of vertices in a bipartite graph such that each edge is incident on at least one of these vertices. This corresponds to the following problem.

**Problem 2** (**Minimum Bipartite Vertex Cover**)**.** The minimum bipartite vertex cover problem is defined as follows: Given a bipartite graph $G = (L, R, E)$, find a smallest set of vertices that cover all edges of the graph.

Let $\mathsf{opt}_P$ denote the optimal value of the matching LP (the primal LP) and $\mathsf{opt}_D$ for the vertex cover LP (the dual LP). Then, the following fact is immediate from the above calculations (and by noticing that relaxing an ILP to an LP can only improve the optimal value).

**Fact 8.** *In any bipartite graph $G = (L, R, E)$,*

$$\textit{maximum matching size} \leqslant \mathsf{opt}_P \leqslant \mathsf{opt}_D \leqslant \textit{minimum vertex cover size}.$$

We shall only use Fact 8 to design our algorithm.

## 3.2 A Primal-Dual $(1-\varepsilon)$-Approximation Algorithm

We are going to design an algorithm that for every given $\varepsilon > 0$, outputs a $(1-\varepsilon)$-approximation to maximum bipartite matching. We will then see how to use this algorithm to obtain a faster algorithm for finding maximum matchings as well.

The general strategy of primal-dual algorithms (with some abuse of their precise definitions in other texts) is as follows. The algorithm starts with a feasible primal solution (here, a matching) and an infeasible dual solution (here, a fractional vertex cover which is infeasible) – throughout the algorithm, we maintain the invariant that the value of the primal is always equal to the value of this infeasible dual solution. Then, we gradually increase the primal value and the corresponding dual solution by focusing on the most violated constraints of the dual (here, the edges that are least covered). Eventually, we show that the dual becomes (approximately) feasible and in that point we can say that the primal-dual pair we maintained are approximately optimal for their respective problems (by applying Fact 8).

The algorithm is formally as follows (see Figure 7 for an illustration of the 'update rule' of the algorithm).

**Algorithm 2. A $(1-\varepsilon)$-approximation algorithm for bipartite matching.**

1. Let $y_u = 0$ for all $u \in L$, $z_v = 0$ for all $v \in R$, and $M = \emptyset$ initially.
2. Let $U = L$ be the set containing all unmatched vertices in $L$ initially[a].
3. While $U \neq \emptyset$:
   (a) Pick any vertex $u$ from $U$ and remove it from $U$.
   (b) Find any vertex $v \in \arg\min_{w \in N(u)} z_w$.
   (c) If $z_v = 1$, skip to the next iteration of the while-loop; otherwise, let $w$ be the matched neighbor of $v$ in $M$ (which potentially may not even exist).
   (d) Change the matching $M$ such that $u$ is matched to $v$ instead and $w$ is now unmatched; insert $w$ to the set $U$.
   (e) Update $z_v \leftarrow z_v + \varepsilon$ and $y_u = 1 - z_v$ and $y_w = 0$.
4. Output $M$ as the final matching.

[a] Although some unmatched vertices later will be removed from $U$, so $U$ may not contain *all* unmatched vertices.

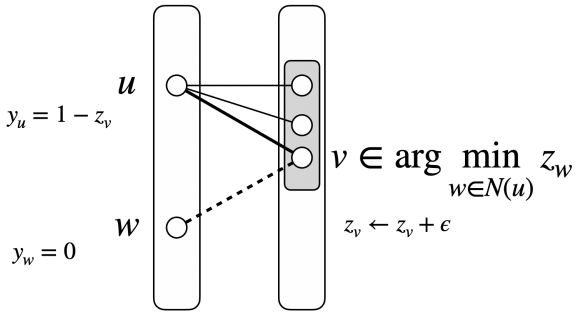



Figure 7: An illustration of the 'update rule' of the algorithm. The neighborhood of $u$ is denoted by the shaded (gray) region. Vertex $v$ has the minimum value $z_v$ in the neighborhood of $u$. Moreover, $v$ may have been matched to some vertex $w$ (dashed line) but the matching is now updated to include the edge $(u, v)$ (solid line) instead.

Before getting to the analysis of this algorithm, let us provide a quick remark.

**Remark.** This primal-dual algorithm can also be cast as an **auction algorithm** (cf. [Ber88]). Think of each vertex $u \in L$ as a buyer and each $v \in R$ as an item. Each buyer has a value of 1 for any of the items in its neighborhood but is *unit-demand*, i.e., only wants one item. Then, the above algorithm can be seen as an auction. At each step, $M$ shows a *tentative* assignment, $z_v$ is the *price* of the item $v$, and $y_u$ is the *utility* or *happiness* of the buyer $u$ (which is equal to 1 (the valuation of the buyer) minus the price it has to pay ($z_v$)). Thus, at each step, the algorithm finds an unmatched/unhappy buyer, and allows to "trade" for the lowest price item in its neighborhood by increasing the price of that item by $\varepsilon$.

We can indeed analyze the algorithm purely with this point of view (see, e.g., [Ber88]) but for our purpose, it will be easier to stick with the primal-dual interpretation directly. That being said, still seeing $y$-values as 'happiness' and $z$-values as 'prices' may provide some more intuition.

**The analysis.** Let us now start the analysis of the algorithm. A key to the analysis of the algorithm is the following *invariants* that hold throughout the algorithm.

**Invariant 9.** For any vertex $u \in L$:

- If $u$ is matched by $M$, then $y_u = 1 - z_{M(u)}$ where $M(u) \in R$ is the matched neighbor of $u$.
- If $u$ is unmatched by $M$, then $y_u = 0$.

*Proof.* This follows directly from the update rule in Line (3e) of the algorithm. □

**Invariant 10.** For any vertex $v \in R$, $z_v > 0$ iff $v$ is matched by $M$.

*Proof.* This follows because when we match $v$ for the first time, we increase $z_v$ from zero, and that a matched vertex $v \in R$ always remain matched by the algorithm, although its matched pair in $L$ may change. □

Combining these invariants allows us to prove a key property of the algorithm as a primal-dual algorithm, namely, that $M$ and $(y, z)$ always have the same value.

**Claim 11.** *At every step of the algorithm,*

$$|M| = \sum_{u \in L} y_u + \sum_{v \in R} z_v.$$

*Proof.* We have,

$$\begin{aligned}
\sum_{u \in L} y_u + \sum_{v \in R} z_v &= \sum_{(u,v) \in M} y_u + z_v + \sum_{u \in L \setminus V(M)} y_u + \sum_{v \in R \setminus V(M)} z_v \\
&\qquad \text{(by pairing up the vertices } u \in L \text{ and } v \in R \text{ that are matched together by } M\text{)} \\
&= \sum_{(u,v) \in M} y_u + z_v \\
&\qquad \text{(the second sum is zero by Invariant 9 and the third sum is zero by Invariant 10)} \\
&= \sum_{(u,v) \in M} 1 \qquad \text{(as } y_u = 1 - z_v \text{ by Invariant 9)} \\
&= |M|,
\end{aligned}$$

as desired. □

The other next key step is to show that once the algorithm finishes, $(y, z)$ becomes an approximately feasible dual solution.

**Claim 12.** *At the end of the algorithm, the vectors* $(\frac{y}{1-\varepsilon}, \frac{z}{1-\varepsilon})$ *form a feasible dual solution.*

*Proof.* To prove the feasibility for the dual LP, given that all $y, z \geqslant 0$ at all times, we only need to show that for every edge $(u, v) \in E$, we have,

$$\frac{y_u}{1-\varepsilon} + \frac{z_v}{1-\varepsilon} \geqslant 1 \qquad \equiv \qquad y_u + z_v \geqslant 1 - \varepsilon.$$

Let us consider the following cases:

- $u$ is unmatched by $M$: the only way this can happen is if we remove $u$ from $U$ at some point and do not consider it further. But then it means that for every neighbor $w \in N(u)$, we have $z_w = 1$ by the criteria in Line (3c). Thus, for $v \in N(u)$, we have $z_v = 1$ and thus $y_u + z_v \geqslant 1$ in this case.

- $u$ is matched by $M$ to $v$: by Invariant 9, we have $y_u = 1 - z_v$ and thus $y_u + z_v = 1$ in this case.
- $u$ is matched by $M$ to some other vertex $w \neq v$: At the time that $u$ was matched to $w$, we had $z_w \leqslant z_v + \varepsilon$ (because $w$ had the minimum $z$-value in the neighborhood of $u$ and we only increased it by $\varepsilon$ after matching it to $u$). The value of $z_w$ does not change after its last time being matched to $u$ by the construction of the algorithm, and $z_v$ can only increase further (because $z$-values are increasing in the algorithm). Thus, at the end of the algorithm also, we have $z_w \leqslant z_v + \varepsilon$. But, by Invariant 9, we have $y_u + z_w = 1$ and thus we also have $y_u + z_v \geqslant 1 - \varepsilon$.

This concludes the proof. □

We can now conclude the analysis as follows.

**Lemma 13.** *Algorithm 2 outputs a $(1-\varepsilon)$-approximate matching $M$.*

*Proof.* We have,

$$\mathsf{opt}_D \geqslant \text{maximum matching size} \geqslant |M| = \sum_{u \in L} y_u + \sum_{v \in R} z_v \geqslant (1-\varepsilon) \cdot \mathsf{opt}_D \geqslant (1-\varepsilon) \cdot \text{maximum matching size},$$

where the first two inequalities are by Fact 8, the next is because $M$ is a feasible matching, the next equality is by Claim 11, the next inequality is by Claim 12 since $(\frac{y}{1-\varepsilon}, \frac{z}{1-\varepsilon})$ is a feasible dual solution, and the last is again by Fact 8. □

**Runtime.** Let us also analyze the runtime of Algorithm 2.

Firstly, see that in each iteration of the while-loop, the $z$-value of some vertex in $R$ increases by $\varepsilon$, but in total, there can only be $O(n/\varepsilon)$ increments to $z$-values before they all become 1 and thus the algorithm terminates. This means that the number of iterations is at most $O(n/\varepsilon)$.

Secondly, it is easy to see that we can implement $U$ via a list which allows us to insert and delete to it in $O(1)$ time. Thus, the only step of the while-loop that takes more than $O(1)$ time is Line (3b) for iterating over the neighbors of $u$. However, this step also takes at most $O(\deg(u)) = O(n)$ time each time.

Putting the above two steps together implies that the algorithm takes $O(n^2/\varepsilon)$ time. However, we are going to make a slightly more careful analysis and implementation of the algorithm to reduce the runtime to $O(m/\varepsilon)$ time.

**Faster implementation of Algorithm 2.** The faster implementation is as follows. For every vertex $u \in L$, maintain a list $D(u)$ called the *demand list* of $u$ and a value $d(u)$ called the *demand value* of $u$.

At the beginning of the algorithm, we go over all neighbors of $u$ and place them in $D(u)$ and set $d(u) = 0$. Then, whenever we need to find $v \in \arg\min_{w \in N(u)} z_v$ in Line (3b) we simply iterate over the list $D(u)$ and for each $w \in D(u)$, if $z_w > d(u)$, we remove $w$ from $D(u)$ and if $z_w = d(u)$, we return $w$ as a choice of $v$. Once the list $D(u)$ becomes empty, we increase $d(u)$ by $\varepsilon$ and again insert all neighbors of $u$ back to $D(u)$.

Inductively, and since $z$-values are only increasing in the algorithm, we have that each vertex returned from $D(u)$ indeed has the minimum price in the neighborhood of $u$ and thus is a valid choice for returning as $v$. This means that this is a correct implementation of the algorithm still.

Finally, in terms of runtime, we will iterate $O(1/\varepsilon)$ time in total over $D(u)$ as once $d(u)$ becomes 1, we will stop considering $u$ anymore in the algorithm by Line (3c). This means that the total time spent for vertex $u$ throughout the entire algorithm is $O(\deg(u)/\varepsilon)$. Thus, the total runtime of the algorithm is proportional to $\sum_{u \in L} \deg(u)/\varepsilon = m/\varepsilon$, hence, the algorithm takes $O(m/\varepsilon)$ time in total.

**Remark.** A purely graph theoretic implication of Algorithm 2 is the following. By setting $\varepsilon \to 0$ (say, $\varepsilon < 1/n^2$ certainly suffices), we obtain a pair of matchings and fractional vertex covers which have the same value which actually implies a stronger version of Fact 8:

$$\text{maximum bipartite matching size} = \mathsf{opt}_P = \mathsf{opt}_D = \text{minimum bipartite vertex cover size}.$$

Here, the middle equality about the value of primal and dual is *always true* for all LPs and their duals (this is called the **strong duality** as opposed to the **weak duality** that upper bounds the primal via dual), and the equality of maximum bipartite matching and minimum bipartite matching size is Konig's theorem. Finally, this result also shows that there is no gap between the LP relaxation of bipartite matching and its ILP and the same holds for bipartite vertex cover (you are encouraged to prove these directly also, which have pretty simple proofs).

### 3.3 A Faster Exact Algorithm for Bipartite Matching

Let us conclude this section by showing how to also obtain an exact algorithm for the bipartite matching problem. There are already two easy ways of doing this.

**Strategy 1:** Run Algorithm 2 with parameter $\varepsilon = 1/(n+1)$, which gives a matching of size at least maximum matching size minus $n/(n+1)$ since max-matching size is at most $n$. But, since matching sizes are always integral, this means that this is a matching of size as large as the maximum matching itself.

The runtime of this algorithm is $O(mn)$ time.

**Strategy 2:** We can simply run the **augmenting path** algorithm described earlier.

The runtime of this algorithm is also $O(mn)$ time.

**Strategy 3:** Interestingly, even though both strategies above have $O(mn)$ time in the worst-case, we can combine them in a simple way to obtain a faster algorithm!

Set $\varepsilon = 1/\sqrt{n}$ and run Algorithm 2 first. This takes $O(m\sqrt{n})$ time and at the end of it, we find a matching $M$ of size

$$(1 - 1/\sqrt{n}) \cdot \mathsf{opt}_P \geqslant \mathsf{opt}_P - \sqrt{n},$$

as $\mathsf{opt}_P \leqslant n$ always. This means that at this point we have at most $\sqrt{n}$ unmatched vertices. Switch to running the augmenting path algorithm from now on. As we can only find $O(\sqrt{n})$ augmenting paths (before matching all vertices), this step also takes another $O(m\sqrt{n})$ time.

As a result, the runtime of this combined algorithm is $O(m\sqrt{n})$ time.

**Remark.** The $O(m\sqrt{n})$ time obtained via the algorithm outlined above matches the runtime of the celebrated *Hopcroft-Karp Algorithm* for bipartite matching [HK73] from 1973, but via a simpler algorithm and analysis.

It is also worth mentioning that while the $O(m\sqrt{n})$ time for bipartite matching was improved over the years for various ranges of parameters, only the very recent breakthroughs on almost-linear time algorithms for maximum flow resulted in completely improving this bound for all ranges of parameters. Discussing this line of work is beyond the scope of our course, and you are referred to [vdBLN+20, CKL+22, vdBCK+23] to learn more about these exciting developments.

# References


[Ber88] Dimitri P Bertsekas. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of operations research*, 14(1):105–123, 1988. 10

[CKL+22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 612–623. IEEE, 2022. 13

[DP14] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1):1:1–1:23, 2014. 7

[Edm65] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965. 4

[GT91] Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for general graph-matching problems. *J. ACM*, 38(4):815–853, 1991. 7

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. 13

[McG05] Andrew McGregor. Finding graph matchings in data streams. In Chandra Chekuri, Klaus Jansen, José D. P. Rolim, and Luca Trevisan, editors, *Approximation, Randomization and Combinatorial Optimization, Algorithms and Techniques, 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2005 and 9th InternationalWorkshop on Randomization and Computation, RANDOM 2005, Berkeley, CA, USA, August 22-24, 2005, Proceedings*, volume 3624 of *Lecture Notes in Computer Science*, pages 170–181. Springer, 2005. 4

[MV80] Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*, pages 17–27. IEEE Computer Society, 1980. 7

[vdBCK+23] Jan van den Brand, Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. *CoRR*, abs/2309.16629, 2023. 13

[vdBLN+20] Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 919–930. IEEE, 2020. 13