MATCHINGS

CSCI 4113/6101
INSTRUCTOR: NORBERT ZEH
OCTOBER 31, 2025

In this topic, we introduce matchings. As a warm-up exercise, we discuss how to find a maxim*al* matching in linear time. A maxim*um* matching is much harder to find.[1] We will discuss algorithms for finding different types of matchings in upcoming topics. We end the introduction to matchings in this topic by showing that the maximum matching problem on bipartite graphs can be expressed as a maximum flow problem. Thus, as a starting point, we can use the maximum flow algorithms we discussed already to compute maximum matchings, at least in bipartite graphs.

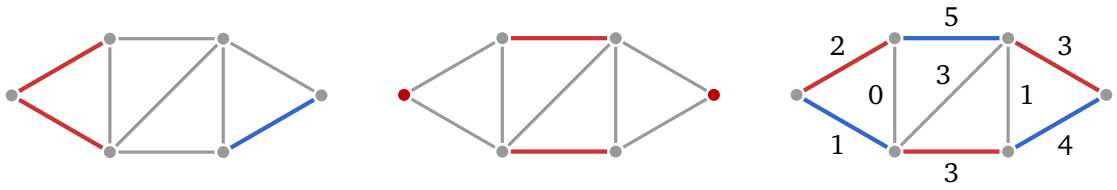
1 MATCHINGS

Matching problems are also known as assignment problems because the basic problem is to pair entities. For example, consider a set of compute jobs to be run and a set of machines they can be run on. Each machine completes each job in a particular amount of time based on the demands of the job in terms of CPU speed, memory bandwidth, and amount of memory available, and on the machine's specifications. You are renting these machines from Amazon at some cost per hour. In this scenario, it is not hard to imagine that the cost of running each job can differ significantly from machine to machine. We want to assign jobs to available machines so that the total cost of running all jobs is minimized.

We can model this as a bipartite graph G whose vertex set is the union of two sets J and M representing the jobs and machines, respectively. If a job $j \in J$ can run on a machine $m \in M$ (some jobs may not be possible to run on a given machine at all, for example, if the job requires more memory than the machine has), then there is an edge $\{j, m\} \in G$ with weight equal to the cost of running job j on machine m. If we assume that we want to run every job on a different machine, our goal is to pick a subset of the edges in G such that no two edges share an endpoint (no two jobs run on the same machine and no job can be split over multiple machines), every vertex in J is the endpoint of a chosen edge (we run every job), and the total weight of the chosen edges is minimized.

A set $M \subseteq E$ of edges in a graph $G = (V, E)$ such that no two edges in M share an endpoint is called a **matching** (see Figs. 1a,b). For every edge $\{u, v\} \in M$, we call u and v **mates**. If every vertex of G is the endpoint of an edge in M, then M is called a **perfect matching** (see Fig. 1c). In the above scenario of scheduling jobs on machines, if the number of jobs equals the number of available machines, then the

[1] I am not sure whether mathematicians use this terminology, but in the algorithms literature, the terms "maximal" and "maximum" are both used as adjectives with rather distinct meanings. A maxim*al* solution is a local optimum in the sense that we cannot make it bigger by adding to it. For example, a maximal matching has the property that it does not have a proper superset that is also a matching. In contrast, a maxim*um* solution is one with maximum objective function value. A matching is maximum if there is no matching containing more edges. A maximal matching M may not be maximum because there may be larger matchings in the given graph, only they are not supersets of M. The same distinction applies to the terms "minimal" and "minimum". A minimum solution is one of minimum objective function value, while a minimal solution has no proper subset that is also a feasible solution.



(a) (b) (c)

Figure 1: (a) A set of edges (red and blue) that are not a matching because the two red edges share an endpoint. (b) A matching in the same graph that is not a perfect matching. The two red vertices are unmatched. (c) Two perfect matchings (red and blue) in the same graph. The red one is a minimum-weight perfect matching, the blue one is not.

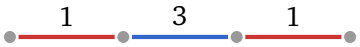


Figure 2: A graph whose maximum matching (red) is not a maximum-weight matching (blue) and vice versa.

requirement that every job has an incident edge in the matching implies that every machine also has an incident edge in the matching. Thus, we are looking for a **minimum-weight perfect matching** (see Fig. 1c). Such a matching may not exist since not every pair of vertices is connected by an edge. Thus, we may also consider the simpler problem of deciding whether a perfect matching exists. This decision problem can be generalized to the **maximum matching** problem, which is to find a matching containing the maximum number of edges. A matching is perfect if and only if its cardinality is $n/2$, where n is the number of vertices of the graph. In the same way that we went from the minimum-weight perfect matching problem to the simpler problem of finding a perfect matching, we can *generalize* the maximum matching problem to the **maximum-weight matching** problem where edges have weights and we want to find a matching of maximum weight. Note that a maximum-weight matching may contain fewer edges than a maximum matching for the same graph if the edge weights in the graph vary sufficiently. An example is shown in Fig. 2. Exer. 1 asks you to prove that we can use any maximum-weight matching algorithm to find a matching of maximum weight among all maximum matchings or a matching of maximum cardinality among all maximum-weight matchings, in an attempt to maximize both the size and the weight of the matching. Tbl. 1 lists the different types of matchings we are interested in in this course. There are many other quality measures for matchings, which arise in different applications, but we won't discuss them here.

The final generalization we consider is to drop the requirement that the given graph be bipartite. A graph $G = (V, E)$ is **bipartite** if its vertex set can be partitioned into two subsets U and W—that is, $U \cup W = V$ and $U \cap W = \emptyset$—such that every edge in E has one endpoint in U and one endpoint in W. We often specify this partition of V explicitly by writing $G = (U, W, E)$. Many natural applications of matching problems work with bipartite graphs and, as we will see, solving matching problems on bipartite graphs can be significantly easier than on arbitrary graphs. Nevertheless, it is useful to be able to find various kinds of matchings in arbitrary graphs, particularly as a building block for other algorithms.

Name	Input	Output
Maximal matching	Graph $G = (V, E)$	Matching $M \subseteq E$ such that there does not exist a matching $M' \subseteq E$ with $M' \supset M$
Maximum matching	Graph $G = (V, E)$	Matching $M \subseteq E$ such that there does not exist a matching $M' \subseteq E$ with $\|M'\| > \|M\|$
Maximum-weight matching	Graph $G = (V, E)$ Edge weights $w : E \to \mathbb{R}$	Matching $M \subseteq E$ such that there does not exist a matching $M' \subseteq E$ with $w(M') > w(M)$
Minimum-weight perfect matching	Graph $G = (V, E)$ Edge weights $w : E \to \mathbb{R}$	Perfect matching $M \subseteq E$ such that there does not exist a perfect matching $M' \subseteq E$ with $w(M') < w(M)$

Table 1: The different types of matchings we study in this course

2 Maximal Matching

As a warm-up exercise, this section discusses the maximal matching problem. Given a graph $G = (V, E)$, the problem is to find a matching $M \subseteq E$ such that there is no matching M' that satisfies $M \subset M' \subseteq E$.

The algorithm is extremely simple. We start by setting $M = \emptyset$. Then we inspect every edge $e \in E$ in turn. When inspecting e, we add e to M if and only if e's endpoints are unmatched at this time. This algorithm can clearly be implemented in $O(n + m)$ time using an adjacency list representation of the graph, which allows us to enumerate the vertices and edges of G in linear time and find the endpoints of each edge in constant time: First, we mark every vertex as unmatched; this takes $O(n)$ time. Then we inspect every edge $e \in E$. For each edge $\{u, v\}$, we check whether both u and v are unmatched. If so, we add $\{u, v\}$ to M and mark u and v as matched. This takes constant time per edge, $O(m)$ time in total.

Lemma 1. *A maximal matching of a graph G can be computed in $O(n + m)$ time.*

Proof. We already argued that the above algorithm takes $O(n + m)$ time. To see that its output is a matching, assume the contrary. Then there exist two edges e and f in M that share an endpoint v. If w.l.o.g. e is added to M before f, then v is marked as matched after adding e to M. Thus, when inspecting f, v is matched, and we do not add f to M, a contradiction.

To see that the computed matching M is maximal, assume there exists an edge $\{u, v\} \in E$ such that $M \cup \{\{u, v\}\}$ is also a matching. Then both u and v are unmatched at the end of the algorithm and, therefore, also when the algorithm inspects the edge $\{u, v\}$. Thus, we would have added $\{u, v\}$ to M, a contradiction again. □

A related problem is the independent set problem. An **independent set** in a graph $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent (see Fig. 3). Again, we can distinguish between a *maximal* independent set, which has no independent proper superset, and a *maximum* independent set, which is an independent set of maximum cardinality. Exer. 2 asks you to verify that a maximal

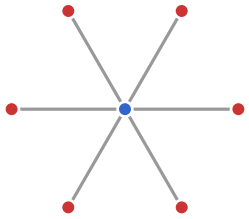
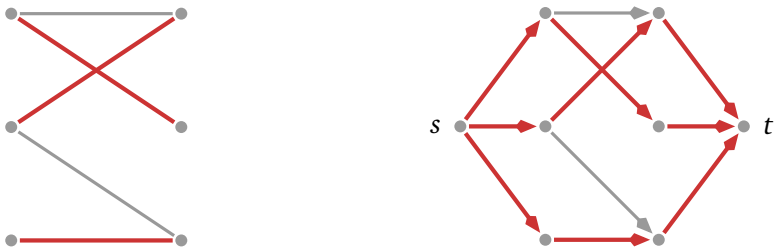


Figure 3: A maximal but not maximum independent set (blue) and a maximum independent set (red). Both sets are vertex covers. The blue set is a minimum vertex cover, the red one is not.

independent set can be computed in $O(n+m)$ time using an algorithm very similar to the maximal matching algorithm above. The maximum-cardinality variants of matching and independent set, on the other hand, are of greatly different difficulty. While the focus of the next three topics is to demonstrate that a wide range of matching problems can be solved in polynomial time, finding a maximum independent set is NP-hard and in fact $W[1]$-hard. $W[1]$-hard problems are unlikely to be fixed-parameter tractable in the same sense that NP-hard problems are unlikely to be solvable in polynomial time. We will discuss fast exponential-time algorithms for finding a maximum independent set later in this course. Matchings are also closely related to another classical NP-hard problem, the **vertex cover** problem, which is to find a minimum-size subset C of G's vertices such that every edge of G has at least one endpoint in C (see Fig. 3). Based on this connection, we will use the matching algorithms discussed in this course as building blocks for parameterized algorithms for the vertex cover problem.

3 Bipartite Maximum Matching via Maximum Flow*

Consider the problem of finding a maximum matching in a **bipartite graph** $G = (U, W, E)$. To find such a matching, we augment G with two new vertices s and t. Every vertex in U becomes an out-neighbour of s; every vertex in W becomes an in-neighbour of t; and every edge of G gets directed from its endpoint in U to its endpoint in W. Finally, we give every edge a capacity of 1. Let us call the resulting graph $\vec{G}$. This construction is illustrated in Fig. 4.



(a) (b)

Figure 4: Reduction of bipartite maximum matching to maximum flow. (a) A bipartite graph $G = (U, W, E)$. The vertices in U are the vertices on the left. The vertices in W are the vertices on the right. Every edge has an endpoint in U and an endpoint in W. (b) The network $\vec{G}$ we construct from G. Edge capacities are not shown because all edges have capacity 1. The red paths in $\vec{G}$ are the ones carrying one unit of flow from s to t and correspond to the red matching in G.

LEMMA 2. *M is a matching in G if and only if there exists an integral st-flow f in $\vec{G}$ such that $M = \{e \in G \mid f_e = 1\}$.*

Proof. First, assume that M is a matching. Then set $f_{s,u} = f_{u,w} = f_{w,t} = 1$, for every edge $\{u,w\} \in M$, and $f_e = 0$, for any other edge of $\vec{G}$. Clearly, f is integral and satisfies the capacity constraints of all edges in $\vec{G}$. For every vertex $u \in G$, if u is unmatched, then $f_{s,u} = 0$ and $f_{u,w} = 0$, for every edge $\{u,w\} \in G$. If u is matched, then $f_{s,u} = 1$ and $f_{u,w} = 1$, for exactly one edge $\{u,w\} \in G$, because M is a matching. Thus, f satisfies u's flow conservation constraint whether u is matched or not. A similar argument shows that f satisfies the flow conservation constraints of all vertices in W. Thus, f is an st-flow.

Now, assume that f is an integral st-flow in $\vec{G}$. By the flow conservation constraint, we have $\sum_{w \in W} f_{u,w} = f_{s,u} \leq 1$, for every vertex $u \in U$. Since f is integral, this shows that there exists at most one edge $\{u,w\} \in G$ such that $f_{u,w} = 1$. By an analogous argument, there exists at most one edge $\{u,w\} \in G$ such that $f_{u,w} = 1$, for every vertex $w \in W$. Thus, the set $M = \{e \in G \mid f_e = 1\}$ is a matching in G. □

PROPOSITION 3. *A maximum matching of a bipartite graph G can be found in $O\left(n^2 m\right)$ time.*

Proof. Constructing $\vec{G}$ from G takes $O(n+m)$ time. We can use the push-relabel algorithm to find a maximum flow in $\vec{G}$ in $O\left(n^2 m\right)$ time. As shown by Exer. 1 in the discussion of the push-relabel algorithm, the flow f this algorithm finds in $\vec{G}$ is an integral flow because all edge capacities in $\vec{G}$ are integers. Thus, by Lem. 2, the set $M = \{e \in G \mid f_e = 1\}$ is a matching in G. Constructing M from f takes $O(n+m)$ time. Overall, computing M thus takes $O\left(n^2 m\right)$ time.

Since $U \cup \{s\}$ is an st-cut in $\vec{G}$ and every edge in G crosses this cut, we have $|M| = F_s$. If M is not a maximum matching, then there exists a matching M' such that $|M'| > |M|$. By Lem. 2, $M' = \{e \in G \mid f'_e = 1\}$, for some integral st-flow f' in $\vec{G}$. Again, since $U \cup \{s\}$ is an st-cut, we have $F'_s = |M'| > |M| = F_s$, a contradiction because f is a maximum flow in $\vec{G}$. This proves that M is a maximum matching in G. □

Maximum-weight matching and minimum-weight perfect matching in bipartite graphs can also be solved via network flows, but the flow problem is not a simple maximum flow problem: we need to solve a minimum-cost flow problem in a network with capacities and costs on its edges. Since we did not discuss minimum-cost flows in this course, we won't discuss this reduction from maximum-weight matching and minimum-weight perfect matching to flow problems here.

EXERCISES

EXERCISE 1. Prove that it is possible to adjust the weights in a weighted graph so that a maximum-weight matching with respect to the adjusted weights is

(a) A matching of maximum weight (with respect to the original weights) among all maximum matchings of the graph or
(b) A matching of maximum cardinality among all maximum-weight matchings (with respect to the original weights) of the graph.

EXERCISE 2. Show that a maximal independent set of an arbitrary graph can be computed in $O(n+m)$ time.