In this first lecture we will first give an outlook about the whole lecture. The remaining part deals with bipartite matchings. First, we recap important definitions and results for unweighted bipartite matchings. Afterwards, we consider the weighted bipartite matching problem and give an algorithm that efficiently solves this problem. Finally, we state some further implications.

1 Overview of the Lecture

Most likely, we will have four topics: matchings, flows, linear programs, and matroids:

- matchings
  - recap: unweighted bipartite matchings
  - weighted bipartite matchings
  - unweighted non-bipartite matchings
  - outlook for weighted non-bipartite matchings
  - stable matchings
- flows
  - recap: flows, Ford-Fulkerson Algorithm, Edmonds-Karp Algorithm
  - blocking flow algorithm (with application for bipartite matchings)
  - push-relabel algorithm
  - min-cost flow algorithm
- linear programs
  - ellipsoid method ($\rightarrow$ polynomial running-time for LPs)
  - interior point method
- matroids
  - greedy algorithm for weighted independent set
  - matroid intersection

2 Recap on unweighted bipartite matching

Let $G = (V, E)$ be a graph. A *matching* $M \subseteq E$ is a subset of the edges such that no two edges of M share a vertex. A vertex $v \in V$ is *covered* by M if there is a $u \in V$ such that $uv \in M$. A matching M is *perfect* if every $v \in V$ is covered by M. A matching M is *maximal* if there is no $e \in E \setminus M$ such that $M + e$ is a matching. A matching m is maximum if there is no matching M' with $|M'| > |M|$. Note that a maximum matching is also maximal and that a perfect matching is a maximum matching.

A graph $G = (V, E)$ is *bipartite* if we can partition V into two disjoint sets L and R such that there are no edges with both endpoints being in one of the sets L or R. We usually denote bipartite graphs with bipartition L and R by $G = (L \cup R, E)$. The problem of finding a maximum matching is defined as follows.

Definition 1 (maximum bipartite matching)**.**
Input: A bipartite graph $G = (L \cup R, E)$.
Task: Compute a maximum matching in G.

In previous lectures we have already seen two variants to compute a maximum matching in a bipartite graph. In the first one, we reduce the problem to a *maximum flow problem*. In the second, we solve it directly using so-called *augmenting paths*.

2.1 Reducing maximum bipartite matching to a maximum flow problem

In the maximum flow problem we are given a directed graph $D = (V, A)$ with capacities $c_e \geq 0$ for each $e \in E$ and two distinct vertices $s, t \in V$. The function $f : A \to \mathbb{R}$ is a flow if the following is satisfied:

- $0 \leq f(e) \leq c_e$ for all $e \in A$
- $\sum_{e \in \delta^+(v)} f(e) = \sum_{e \in \delta^-(v)} f(e)$ for all $v \in V \setminus \{s, t\}$ (flow conservation)

We note that the *value* of a feasible flow f is defined as $v(f) = \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e)$ and it is equal to $\sum_{e \in \delta^-(t)} f(e) - \sum_{e \in \delta^+(t)} f(e)$. The maximum flow problem is defined as follows.

Definition 2 (maximum flow problem)**.**
Input: A directed graph $D = (V, A)$ with capacities $c_e \geq 0$ for all $e \in E$ and two distinct vertices $s, t \in V$.
Task: Compute an s-t flow in G of maximum value.

Theorem 1. *An s-t flow of maximum value can be computed in polynomial time. Furthermore, if c_e is integer for all $e \in E$, then there is an s-t flow of maximum value that is integer and such a flow can be computed in polynomial time.*

We can reduce the maximum bipartite matching problem to the maximum s-t flow problem as follows: We take $G = (L \cup R, E)$ and orient each edge towards R. Further, we add two vertices s and t and connect s to each vertex in L directed away from s and connect t to each vertex in R directed towards t. The edges incident to s and t have capacity 1, while all other edges have capacities $|L|$. Now all values of c are integer, one can use Theorem 1 to compute a maximum s-t flow f that is integer. We set M to be the set of edges corresponding to edges with flow value 1 which are not incident to s or t. Note that M is a matching in the original graph: By construction, for each $u \in L$ and each $u \in R$ there is at most one edge incident, since f is a maximum integer s-t flow.

2.2 An augmenting path style algorithm for maximum bipartite matching

In this section we follow a different approach to obtain a polynomial time algorithm for maximum bipartite matching. For a graph $G = (V, E)$ and a given matching M, we say that a vertex v is *exposed* with respect to M if it is not covered by M, i.e., there is no edge in M that is incident to v. An *alternating path* is a path in G that alternatingly uses edges in M and edges in $E \setminus M$. An alternating path is called *M-augmenting* if additionally the endvertices of the path are exposed.

M-augmenting paths are useful in the following sense: Let W be an M-augmenting path. If we exchange the edges on W, we can obtain a larger matching M'. This is called the symmetric difference:

$$M' = M \Delta E(W) \coloneqq (M \cup E(W)) \setminus (M \cap E(W)) = M \setminus E(W) \cup E(W) \setminus M$$

Observe that M' is a matching and $|M'| = |M| + 1$. Hence, if we can find an M-augmenting path, then we can obtain the matching M' that is larger than M. One can show that a matching M is maximum if and only if there is no M-augmenting path. The proof is an exercise.

Theorem 2. *Let M be a matching in a graph $G = (V, E)$. M is maximum if and only if there is no M-augmenting path.*

Note that the above theorem holds for arbitrary (and not necessarily bipartite) graphs.

Hence, if we want to find a maximum matching, the following algorithm seems to be a good idea: As long as there exists an M-augmenting path W, update M to $M \Delta E(W)$. The only question is: How do we find an M-augmenting path?

We use that $G = (L \cup R, E)$ is bipartite. Let M be some matching in G. We describe an algorithm that finds an M-augmenting path if one exists. We construct a directed graph D_M obtained from G by orienting each edge $e \in M$ from L to R, and each edge $e \in E \setminus M$ from R to L. Now it is easy to see that a path from an exposed vertex $u \in V$ to an exposed vertex $v \in V$, $u \neq v$, in D_M corresponds to an M-augmenting path. The proof is an exercise.

Lemma 1. *Let M be a matching in a bipartite graph $G = (L \cup R, E)$. There is an M-augmenting path in G if and only if there is a directed path in D_M from an exposed vertex $u \in V$ to an exposed vertex $v \in V$, $u \neq v$.*

Hence, it remains to find such a path in D_M. This can simply be done by a Breadth First Search (BFS) algorithm.

Theorem 3. *We can compute a maximum matching in a bipartite graph in time $O(|V| \cdot |E|)$.*

Proof. We start with an empty matching $M = \emptyset$ and find M-augmenting paths until we obtain a maximum matching. Clearly, there are at most $|V|$ iteration, since a matching has size at most $|V|$. In each iteration, we construct D_M and search for a path using a BFS. This can be done in time $O(|E|)$. Hence, we obtain an overall running time of $O(|V| \cdot |E|)$. □

Later in this lecture we will see how to improve the running time for computing a maximum bipartite matching. We now shift our focus to finding a maximum *weight* bipartite matching.

3 Maximum weight bipartite matching

In this section we generalize the maximum bipartite matching problem and consider weights on the edges. That is, each edge has some weight and the goal is to compute a maximum *weight* bipartite matching. Note that such a matching might neither be maximum, nor maximal. The problem is formally defined as follows.

Definition 3 (maximum weight bipartite matching)**.**
Input: A bipartite graph $G = (L \cup R, E)$, weights $w_e \in \mathbb{R}$ for all $e \in E$.
Task: Compute a maximum weight matching in G, i.e., a matching M that maximizes $\sum_{e \in M} w_e$.

The idea for the algorithm that computes a maximum weight bipartite matching is very similar to our previous algorithm: We start with an empty matching $M = \emptyset$. If we found a matching M, let D_M be the directed graph obtained from G by orienting each edge $e \in M$ from L to R, with *length* $\ell_e := w_e$, and each edge $e \in E \setminus M$ from R to L, with length $\ell_e := -w_e$. Let L_M and R_M denote the exposed vertices of M contained in L and R, respectively. If there is a L_M-R_M path, find a shortest such path, say P, and set $M' := M \Delta E(P)$.

We iterate this procedure until there exists no L_M-R_M path in D_M (hence, M has maximum-size). Among all matchings found in this algorithm, we output the one with maximum weight.

We claim that the matching output by this algorithm computes a maximum weight matching. To see this, we call a matching M *extreme* if it has maximum weight among all matching of size $|M|$. We inductively show that each matching we compute in our algorithm is extreme.

Lemma 2. *Each matching M found in the above algorithm is extreme.*

Proof. The statement is clearly true if $M = \emptyset$. Suppose now that M is extreme, and let P and M' be the augmenting path and matching found in the next iteration. We want to show that M' is also extreme. This then proves the statement by induction.

Consider any extreme matching N of size $|M| + 1$. As $|N| > |M|$, $M \Delta N$ has a component Q that is an M-augmenting path. As P is a shortest M-augmenting path, we know $\ell(Q) \geq \ell(P)$. As $N \Delta Q$ is a matching of size $|M|$, and as M is extreme, we have $w(N \Delta Q) \leq w(M)$. Hence,

$$w(N) = w(N \Delta Q) - \ell(Q) \leq w(M) - \ell(P) = w(M') \ ,$$

and therefore M' is also extreme. □

Hence, Lemma 2 implies that the above algorithm correctly outputs a maximum weight matching in a bipartite graph. We can bound the running time as follows.

Theorem 4. *We can compute a maximum-weight matching in a bipartite graph in time $O(|V|^2 \cdot |E|)$.*

Proof. We start with an empty matching $M = \emptyset$ and find M-augmenting paths until we obtain a maximum-size matching. Clearly, there are at most $|V|$ iteration, since a matching has size at most $|V|$. In each iteration, we construct D_M and search for a shortest path. Note that the graph has edges of negative length and hence we can use the Bellman-Ford Algorithm to find a shortest L_M-R_M path in D_M. This algorithm has a running time of $O(|V| \cdot |E|)$. Hence, we obtain an overall running time of $O(|V|^2 \cdot |E|)$. □

We note that one can also use a version of Dijkstra's Algorithm to compute the shortest L_M-R_M path in D_M, even though the graph has negative edge-cost. Since Dijkstra's Algorithm has a running time of $O(|E| + |V| \log(|V|))$, the running time of the overall algorithm can be reduced to $O(|V| \cdot (|E| + |V| \log(|V|)))$.

Theorem 5. *We can compute a maximum-weight matching in a bipartite graph in time* $O(|V| \cdot (|E| + |V| \log(|V|)))$.

Actually, we can observe that one can stop the above algorithm as soon as the new matching M' has no larger weight than M, that is, no L_M-R_M path in D_M has negative length. We obtain the following improved running-time. Its proof will be an exercise.

Theorem 6. *We can compute a maximum-weight matching in a bipartite graph in time* $O(n' \cdot (|E| + |V| \log(|V|)))$, *where* n' *is the minimum size of a maximum-weight matching.*

A closely related problem is the *assignment problem*, the problem of finding a minimum-weight perfect matching in an edge-weighted (bipartite) graph G.

Definition 4 (assignment problem)**.**
Input: A bipartite graph $G = (L \cup R, E)$, *weights* $w_e \in \mathbb{R}$ *for all* $e \in E$.
Task: Compute a perfect matching in G *of minimum weight, i.e., a perfect matching* M *that minimizes* $\sum_{e \in M} w_e$.

Adapting our algorithm from above yields a polynomial time algorithm for finding an optimum solution for the assignment problem. Proving this is also an exercise.

Theorem 7. *We can compute a minimum-weight perfect matching in a bipartite graph in time* $O(|V| \cdot (|E| + |V| \log(|V|)))$.