Combinatorial Optimization

Frank de Zeeuw

February 18, 2016

This is .....

Chapter 1

Trees

1.1 Minimum Spanning Trees • 1.2 Breadth-First Search Trees • 1.3 Directed Trees

1.1 Minimum Spanning Trees

In this subsection graphs will be undirected and weighted (by a function $w : E(G) \to \mathbb{R}$).

Definitions
A graph T is a *tree* if it satisfies one of the following equivalent conditions:

- T is connected and acyclic;
- T is minimally connected (removing any edge disconnects it);
- T is maximally acyclic (adding any edge creates a cycle);
- T is connected and $|E(T)| = |V(T)| - 1$;
- for any $u, v \in V(T)$, there is a unique path in T between u and v.

A *spanning tree* of a graph G is a subgraph T with $V(T) = V(G)$ which is a tree.
A *minimum spanning tree* of G is a spanning tree T such that $w(T) = \sum_{e \in E(T)} w(e)$ is minimum among all spanning trees of G.
A graph is a *forest* if it is acyclic; equivalently, it is a disjoint union of trees.
Finally for $S \subset V(G)$ we define $\delta(S) = \{e \in E(G) : e \cap S = 1\}$ (recall that an edge is a set of two vertices, so $\delta(S)$ is the set of edges with exactly one vertex in S).

Greedy Tree-Growing Algorithm
We want an algorithm that, given a weighted undirected graph G, finds a minimum spanning tree, or determines that there is none (which implies the graph is disconnected).
The easiest thing to try is a greedy approach: repeatedly add a smallest edge, while preserving connectedness and acyclicity (i.e. you maintain a tree all along). More precisely:

Tree-Growing Algorithm

1. Start with T being a single vertex;
2. Find $e \in \delta(T)$ with minimum $w(e)$; if $\delta(T) = \emptyset$, go to 4;

3. Add e to T; go to 2;

4. If $|E(T)| = |V(G)| - 1$, return T, else return "disconnected".

This algorithm is usually called *Prim's algorithm.*

Theorem 1.1.1. *The Tree-Growing Algorithm returns a minimum spanning tree, if it exists.*

Proof 1. Suppose it returns T. Let T^* be a minimum spanning tree that has as many edges as possible in common with T. If $T = T^*$, then we are done, so assume it isn't.
Let e be the first edge not in T^* that is picked by the algorithm, and let T' be the tree that the algorithm had before it picked e. Since $e \notin T^*$, adding it creates a cycle. Some of the edges in that cycle must not be in T', otherwise the algorithm couldn't have picked e. Let f be such an edge with one endpoint in T' and the other not (this is possible because one endpoint of e is in T'). Then since the algorithm chose e over f, we must have $w(e) \leq w(f)$.
Now define $T^{**} = T^* - f + e$, so $w(T^{**}) \leq w(T^*)$, hence T^{**} is also a minimum spanning tree. But it has one more edge in common with T than T^*, contradicting the definition of T^*. It follows that $T = T^*$, and T is minimum. □

Proof 2. Call a tree *good* if it is contained in a minimum spanning tree. Clearly the empty graph is good. We will inductively show that all the trees occurring during the algorithm are good, hence so is the final tree T. Since T is clearly a spanning tree, it follows that it is minimum.
So suppose we are in step 2 with a good tree T', contained in a minimum spanning tree T^*. Then we want to show that $T' + e$ is also good, where $e \in \delta(T')$ with minimum $w(e)$. If $e \in T^*$, then this is obvious, so suppose $e \notin T^*$. Adding e to T^* creates a cycle, which must contain some edge $f \in \delta(T')$. Since the algorithm chose e over f, we have $w(f) \geq w(e)$. Then $T^{**} = T^* - f + e$ is a minimum spanning tree containing T' and e, which means $T' + e$ is good. □

Greedy Forest-Growing Algorithm
There are different greedy approaches for finding a minimum spanning tree, where a different property is preserved in the process. For instance, one can just preserve acyclicity (i.e. maintain a forest), and in the end the connectedness will follow just from the number of edges. This leads to:

Forest-Growing Algorithm

1. Start with empty graph F, and set $S = E(G)$ (edges still to be done);

2. Find $e \in S$ with minimum $w(e)$; if $S = \emptyset$ or $|E(F)| = |V(G)| - 1$, go to 4;

3. Add e to F, unless that creates a cycle; remove e from S; go back to 2;

4. If $|E(F)| = |V(G)| - 1$, return F, else return "disconnected".

Theorem 1.1.2. *The Forest-Growing Algorithm returns a minimum spanning tree, if it exists.*

Proof. See the problems at the end of this chapter. □

Equivalent problems
Two optimization problems are *equivalent* if one can translate one problem into the other, and the other way around, such that an algorithm for one will give an algorithm for the other. Technically the translation should not have too large running time in some sense, but we won't make that precise here (below the translations actually have linear running time).

Theorem 1.1.3. *Given a weighted undirected graph, the following* 4 *optimization problems are equivalent:*

- Minimum/Maximum Spanning Tree Problem: *Find a spanning tree with minimum/maximum weight, if one exists.*
- Minimum/Maximum Forest Problem: *Find a forest of minimum/maximum weight.*

Proof. The equivalence between the minimum and maximum versions follows simply by multiplying all weights by -1. For instance, to find a maximum spanning tree in a graph with weights $w(e)$, instead look for a minimum spanning tree in the same graph but with weights $-w(e)$. (Note that this trick happens to work for trees, but will not for other objects, for instance for shortest paths).
We will show that the maximum spanning tree and maximum forest problems are also equivalent, which completes the theorem.
Given a graph G in which to find a maximum forest, modify it to G' as follows: remove all edges with negative weight (would never be used in a maximum forest), then add edges with weight 0 between any two vertices that are not connected. Then G' is connected, so has a spanning tree. Find a maximum spanning tree T' in G'. Then removing the edges of T' with weight 0 gives a maximum forest F in G.
Conversely, given a graph G with weights w in which to find a maximum spanning tree, give it new weights w' as follows. Let W be the largest weight of any edge, and define $w'(e) = w(e) + W + 1$, so that all $w(e) > 0$. Find a maximum forest F in G' with w'. Because all weights are positive, F will be connected, so a maximum spanning for w'. Then it is also a minimum spanning tree for w, because $w(T) = w'(T) - (|V(G)| - 1) \cdot (W + 1)$. □

Linear Programming Formulation
The Minimum Spanning Tree Problem has a good LP formulation, but since we didn't really need it to find an algorithm, I will give it here without proof.
It is good in the sense that the polytope described by its inequalities is *integral*, i.e. the linear program and the integer program have the same optimum solutions (regardless of what linear function is to be optimized). In other words, the vertices of that polytope correspond exactly to the spanning trees of G. That too I won't prove here, but it will follow from a more general theorem that we will very likely see later in the course??? (the matroid polytope theorem).

$$\begin{aligned} &\text{minimize } \sum w(e)x_e && \text{subject to} \\ &\sum_{e \in E(G[X])} x_e \leq |X| - 1 && \text{for } \emptyset \neq X \subsetneq V(G), \\ &\sum_{e \in E(G)} x_e = |V(G)| - 1, && \text{and } 0 \leq x \leq 1. \end{aligned}$$

1.2 Breadth-First Search Trees

In this subsection the graphs are directed but unweighted.
An edge of a direct graph has a head vertex $h(e)$ (where the arrowhead is), and a tail vertex

$t(e)$. I will always write a directed edge with its tail first, so if $e = uv$, then $t(e) = u$, $h(e) = v$.

Definitions
We define $\delta^{\text{in}}(v) = \{e : h(e) = v\}$ and $\delta^{\text{out}}(v) = \{e : t(e) = v\}$, and more generally $\delta^{\text{in}}(S) = \{e : h(e) \in S,\ t(e) \notin S\}$ and $\delta^{\text{in}}(S) = \{e : t(e) \in S,\ h(e) \notin S\}$ for $S \subset V(G)$.
A *directed path* is directed graph whose underlying undirected graph is a path, and $\delta^{\text{in}}(v) \leq 1$ and $\delta^{\text{out}}(v) \leq 1$ for all its vertices v.
A *directed cycle* is a directed path such that the underlying undirected graph is a cycle; equivalently, it is a connected directed graph such that $\delta^{\text{in}}(v) = \delta^{\text{out}}(v) = 1$ for all its vertices v.
A *directed tree with root* r is a directed graph T satisfying one of the following equivalent definitions:

- The underlying undirected graph is a tree, $|\delta^{\text{in}}(r)| = 0$, and $|\delta^{\text{in}}(v)| = 1$ for all $v \in V(T)\backslash\{r\}$.
- For each $v \in V(T)\backslash\{r\}$, there is a unique directed path from r to v.

A *directed forest* is a directed graph F such that the underlying undirected graph is a forest, and $|\delta^{\text{in}}(v)| \leq 1$ for all $v \in V(F)$.
These definitions can be summarized as follows: a *directed X* is an X in the underlying undirected graph with all edge oriented "in the same direction".

Breadth-First Algorithm
This is not an optimization algorithm, but it will be important for finding shortest paths.

BFS-Tree Algorithm (given root r)

1. Start with the graph T consisting just of r;
2. Set $S = \delta^{out}(T)$.
3. For each $e \in S$, add e to T if it does not create a cycle (in the undirected sense).
4. If $\delta^{\text{out}}(T) \neq \emptyset$, go back to 2.
5. If $|E(T)| = |V(G)| - 1$, return T; else return "disconnected".

Note that this is not the standard way of describing this algorithm (usually one would use a *queue*), but it will suit our purposes better.

Theorem 1.2.1. *Given a graph G and root r, the BFS-Tree Algorithm returns a spanning directed tree T rooted at r, if one exists.*
The unique path from r to a v is a shortest such path; in other words, $dist_T(r, v) = dist_G(r, v)$.

Proof. The algorithm clearly maintains a directed tree rooted at r, and if it returns T it must have $|E(T)| = |V(G)| - 1$, which implies that it is spanning.
A vertex v is added to T (along with an entering edge) at the kth loop of the algorithm if and only if $k = \text{dist}_T(r, v) = \text{dist}_G(r, v)$. Indeed, if there were a shorter path, v would have been added in an earlier loop. □

1.3 Directed Trees

In this subsection the graphs are directed and weighted.
We will write SDT for *spanning directed tree*.

Equivalent problems

Theorem 1.3.1. *Given a weighted undirected graph, the following* 6 *optimization problems are equivalent:*

- Minimum/Maximum SDT Problem: *Find a spanning directed tree with minimum/ maximum weight.*
- Minimum/Maximum SDT Problem with Root r: *Find a spanning directed tree with root r and minimum/maximum weight.*
- Minimum/Maximum Directed Forest Problem: *Find a directed forest with minimum/maximum weight.*

Proof. The equivalence of the directed tree and directed forest problems is analogous to that for undirected trees and forests, and I will omit it here, as well as the equivalence of minimum/maximum versions. Assuming those, I will show that the rooted minimum SDT problem is equivalent to the others.
Given a graph G in which to find a minimum directed forest, add a new root r to it as follows:

$$V(G') = V(G) \cup \{r\}, \quad E(G') = E(G) \cup \{rv : v \in V(G)\}.$$

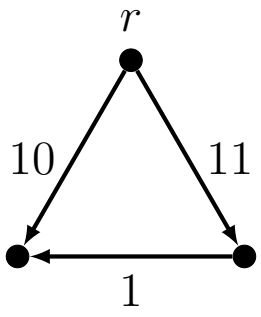
Then find a minimum SDT with root r in G'. Removing r will give a minimum directed forest.
Given a graph G in which to find a minimum SDT with given root r, add a new vertex as follows:

$$V(G') = V(G) \cup \{s\}, \quad E(G') = E(G) \cup \{sr\}.$$

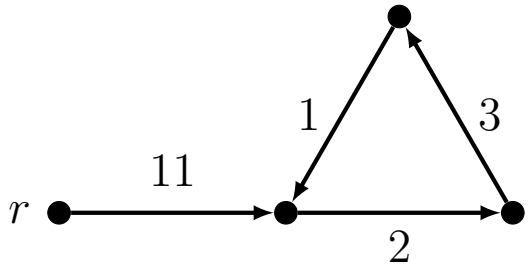
Find a minimum SDT in G'; it must be rooted at s. Removing s will give a minimum SDT with root r. □

Below we will deal only with the Minimum Spanning Directed Tree Problem with Root r. For convenience, we will assume that $\delta^{\text{in}}(r) = 0$, without loss of generality. Since edges entering r will never be used in a directed tree rooted at r, we can always just remove those.

Greedy doesn't work
It turns out that for directed trees, the greedy approach doesn't work (it doesn't give a directed tree, or not a spanning one, or it doesn't give a minimum one).
For instance, try to grow a directed tree, by repeatedly adding a minimum edge that preserves connectedness, acyclicity (in the undirected sense), and the single-entry condition. That will fail, for instance for:



You would take the edge with weight 10, then the one with weight 11. That does give an SDT, but there is a smaller one, consisting of the edges with weights 11 and 1.
One could take a different greedy approach that would work for the example above, by growing a directed forest, so preserving only acyclicity and the single-entry condition. But that would fail for other graphs, like



You would end up with the edges of weight 1 and 2, which do not make up a directed tree. Yet there is a directed tree, with the edges of weight 11, 2, and 3.
Of course, it's hard to prove that no alternative greedy approach couldn't work. But below we will see a "almost-greedy" algorithm: it first takes edges greedily, preserving only the single-entry condition, then fixes that to get a directed tree. ???

1.4 Problems

1. Prove carefully that the Forest-Growing Algorithm given in class returns an MST, if it exists.

2. Give a greedy algorithm that finds a minimum spanning tree by removing edges while preserving connectedness. Prove that it works. Give an example to show that the same approach does not work for minimum spanning directed trees.

3. Prove that if its weight function w is injective, then a graph has at most one minimum spanning tree.

4. Give an algorithm that finds the second-smallest spanning tree in a weighted undirected graph (in other words, given a graph and an MST T, find the smallest among all spanning trees distinct from T). Prove that it works.

(Solutions on page 73.)

Chapter 2

Paths

2.1 BFS • 2.2 Dijkstra's Algorithm • 2.3 LP Formulation • 2.4 Ford's Algorithm

SHORTEST PATH PROBLEM: Given a weighted directed graph G and $a, b \in V(G)$, find a path from a to b of minimum weight, if one exists.

2.1 Breadth-First Search

Below we will go through successively better algorithms, but we won't prove correctness for all of them, only for the last one.

Unweighted graphs
Note first that the greedy approach does badly for shortest path: building a path by greedily adding shortest available edge will probably not even give a path from a to b, and if you're lucky and it does, it still need not be minimal. Fortunately, in the last lecture we saw that a BFS-tree gives shortest paths to its root. We will start from that idea, but phrase it in terms of distance sets: D_k will be the set of vertices at distance k from a.

Notation: In a directed graph, we define the *neighborhoods* of $S \subset V(G)$ by

$$N^{\text{out}}(S) = \{v \in V(G) \backslash S : \exists u \in S \text{ with } uv \in E(G)\},$$

$$N^{\text{in}}(S) = \{v \in V(G) \backslash S : \exists u \in S \text{ with } vu \in E(G)\}.$$

BFS Algorithm for distance sets from a for unweighted graphs

1. Set $D_0 = \{a\}$, $k = 1$.
2. Compute $D_k = N^{\text{out}}(D_{k-1}) \backslash \left(\cup_{i=0}^{k-1} D_k\right)$.
3. If $D_k = \emptyset$, go to 4; else set $k := k + 1$ and go back to 2;
4. Return the sets D_i for $i = 0, 1, \ldots, k-1$.

Distance from a to b: We have $\text{dist}(a, b) = i$ iff $b \in D_i$. More generally, if we define $d : V(G) \to \mathbb{Z}_{\geq 0}$ by $d(v) = i$ iff $v \in D_i$, then $d(v) = \text{dist}(a, v)$ for any $v \in V(G)$.
Of course, if you only cared about b, you could let the algorithm stop as soon as $b \in D_k$.

Shortest ab-path: We can find the shortest path from a to b by going backwards from b. If $b \in D_m$, set $v_m = b$, and repeatedly choose

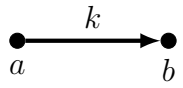
$$v_i \in N^{\text{in}}(v_{i+1}) \cap D_i,$$

from $i = m-1$ to $i = 0$. Then $v_0 = a$ and $P = av_1v_2 \cdots v_{m-1}b$ is a shortest ab-path.
If you want all shortest ab-paths, go through all possible such sequences of choices of v_i.

Nonnegative weights
Can we do the same with arbitrary weights? Think of an unweighted graph as having weight 1 at every edge. Then we could handle any graph with nonnegative integral weights by splitting an edge with weight k into k edges of weight 1. That gives an unweighted graph with corresponding shortest paths.
Doing this more directly gives the following not-so-efficient algorithm:

Set $D_0 = \{a\}$, and iteratively compute $D_k = \{v : \exists u \in D_{k-m} \text{ such that } w(uv) = m\}$ until $\bigcup D_i = V(G)$.

Clearly this is not very efficient, since every time we compute D_k we have to look through all the D_i we have so far. But what is worse is that many D_k will be empty, and many steps will be pointless. In fact, for the graph



the algorithm will go through k steps, for arbitrarily large k. That implies that the running time is not $O(f(n))$ for any function $f(n)$ of $n = |V(G)|$, so it certainly isn't polynomial.
Fortunately, this is not too hard to fix, by skipping the empty D_k as follows.

BFS Algorithm for distance sets from a for nonnegative weights

1. Set $D_0 = \{a\}$, $d(a) = 0$, $S = D_0$;
2. Compute $m = \min(d(u) + w(uv))$ for $uv \in \delta^{\text{out}}(S)$; if $\delta^{\text{out}}(S) = \emptyset$, go to 5;
3. Compute $D_m = \{v : \exists uv \in \delta^{\text{out}}(S) \text{ with } d(u) + w(uv) = m\}$;
4. Add D_m to S, set $d(v) = m$ for $v \in D_m$, go to 2;
5. Return the sets D_i.

Correctness: We won't prove correctness here, because it will follow from correctness of the more general algorithm that we see later.??? But it should be plausible: a vertex is only added to D_m if we have a path of length m to it, and there is no shorter path because then it would have been added to an earlier D_i.

Polynomial running time: In each step 2 and 3, we do $2|\delta^{\text{out}}(S)| \leq 2|E(G)|$ computations, and we will pass through at most $|V(G)|$ loops, so the algorithm is polynomial.

Nonintegral weights: Unlike for the previous attempt, this will work fine for nonintegral weights.

Shortest ab-path: We can still determine a shortest ab-path as before: If $b \in D_s$, find some $u \in D_r$ such that $r + w(ub) = s$, and repeat this. This is not very efficient, one could do much better by storing a "predecessor" for every vertex during the algorithm, but the idea is the same.

2.2 Dijkstra's Algorithm

Dijkstra's algorithm is essentially the algorithm above, but made a bit more efficient, and described more concisely. It gets rid of the D_ks and only uses the distance function $d(v)$.
It also computes temporary estimates $d(v)$ for v that are not yet in S, so that $d(u)+w(uv)$ does not have to be computed over and over. It does this in such a way that always $d(v) \geq \text{dist}(a, v)$, and we put v into S once we have $d(v) = \text{dist}(a, v)$.

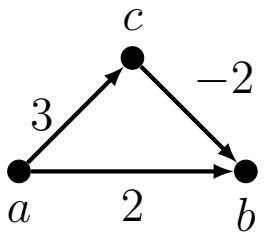
Dijkstra's Algorithm for shortest paths from a for nonnegative weights

1. Set $d(a) = 0$ and $d(v) = \infty$ for $v \neq a$, and $S = \emptyset$;
2. Take $u \notin S$ with minimal $d(u)$ and add u to S; if $S = V(G)$, go to 4;
3. Recompute $d(v)$ for all $v \in N^{\text{out}}(u) \backslash S$:
   if $d(v) > d(u) + w(uv)$, set $d(v) := d(u) + w(uv)$ and $p(v) = u$;
   go back to 2;
4. Return d and p.

Shortest ab-path: The predecessor function p defines a path: Set $v_1 = b$ and iteratively set $v_k = p(v_{k-1})$, until some $v_m = a$. then $P = av_{m-1}v_{m-2} \cdots v_2 b$ is a shortest path from a to b.

Variations: Again, you could modify the algorithm if you're only interested in b: let it stop as soon as b goes into S.
If you wanted not just a but every shortest path, then you would have to let p take sets as values and set $p(v) = \{u\}$ whenever $d(v) > d(u) + w(uv)$, and add u to $p(v)$ whenever $d(v) = d(u) + w(uv)$ (i.e. whenever you find a path that is the same length as the current shortest path).

Negative weights: BFS and Dijkstra may not work when the graph has negative weights. For instance, for



the algorithm will end with $d(b) = 2$, when it should be 1. It will choose b before c, set $d(b) = 2$ and put b into S, after which it no longer updates $d(b)$.
In terms of BFS, the problem is that a negative edge with weight w goes back from D_k to D_{k-w}, so it's hard to be sure when you're done with D_k.
There are various things you could try to fix this, but it is probably more instructive to change perspectives, and look at this as a linear programming problem.

2.3 Linear Programming Formulation

Notation: Write $E = E(G)$ and $V = V(G)$. We will use a vector of variables $x = (x_e)_{e \in E}$, which, if all $x_e \in \{0, 1\}$, corresponds to the subgraph H of G with $V(H) = V(G)$ and $E(H) =$

$\{e \in E : x_e = 1\}$.
For convenience we will also write w_e for $w(e)$; we allow negative $w(e)$, but we will see below that the program is not always feasible.

IP for shortest ab-path

$$\text{minimize } \sum w_e x_e \quad \text{with } x \geq 0, \ \boxed{x \in \mathbb{Z}^{|E|}},$$
$$\sum_{e \in \delta^{\text{in}}(v)} x_e - \sum_{e \in \delta^{\text{out}}(v)} x_e = 0 \quad \text{for } v \in V \backslash \{a, b\},$$
$$\sum_{e \in \delta^{\text{in}}(b)} x_e = 1, \qquad - \sum_{e \in \delta^{\text{out}}(a)} x_e = -1.$$

Note that a feasible solution to this program is not necessarily a path, but can be a union of a path and several cycles. Still, a minimal solution to the integral program will be a path, unless there are cycles with negative weight (see below), in which case the program is unbounded (you could add that negative cycle arbitrarily many times).
For the LP relaxation of this program, the following theorem says that the same is true, with the added complication that if there are multiple shortest paths, some combinations of those would be optimal as well. In other words, the shortest paths are vertices of the corresponding polytope, and any point on the face spanned by them is also optimal.

Theorem 2.3.1. *x is an optimal solution to the relaxation of the IP above if and only if $x = \sum c_i p_i$, where each p_i is an integral vector and corresponds to a shortest ab-path P_i, and $\sum c_i = 1$ with $c_i \geq 0$.*

We won't prove it here, because later in the course we will see a more general theorem that we can deduce this statement from. But a straightforward proof would not be too difficult.

Dual Program
To dualize the LP above, it is convenient to consider the last two constraints as special cases of the general one, which we can do if we remove all edges coming into a and going out of b, and define the vector $f = (f_v)_{v \in V}$ by $f_b = 1$, $f_a = -1$, and all other $f_v = 0$.
Then we can take linear combinations of the constraints by a vector $(y_v)_{v \in V}$:

$$y_b - y_a = \sum_{v \in V} y_v \cdot f_v = \sum_{v \in V} y_v \cdot \left(\sum_{e \in \delta^{\text{in}}(v)} x_e - \sum_{e \in \delta^{\text{out}}(v)} x_e \right) = \sum_{uv \in E} x_{uv}(y_v - y_u),$$

and observe that if all $y_v - y_u \leq w_e$, then we have $y_b - y_a \leq \sum w_e x_e$. That gives us the dual program:

Dual of relaxation

$$\text{maximize } y_b - y_a \quad \text{with } y \in \mathbb{R}^{|V|},$$
$$y_v - y_u \leq w_{uv} \quad \text{for } uv \in E.$$

Potentials: A feasible dual solution is called a *potential* for the weight function w; in other words, a potential is a function $d : V(G) \to \mathbb{R}_{\geq 0}$ such that $d(v) - d(u) \leq w(uv)$ for every directed edge $uv \in E(G)$. That should look familiar: it is in fact the triangle inequality, and the distance functions d as found in the algorithms above satisfy it. But not every potential gives such a distance function, since the distance functions have equality, $d(v) - d(u) = w(uv)$, along every shortest path from a.
Note that feasibility does not depend on a or b, but optimality does.

Negative cycles: Seeing the dual program, we can deduce a condition for when shortest paths exist, or equivalently, when the dual has a feasible solution.
Given a weighted directed graph, define a *negative cycle* to be a directed cycle C such that $\sum_{e\in C} w_e < 0$.

Lemma 2.3.2. *Assume G is a weighted graph with the property that every vertex can be reached by a directed path from a.*
Then the dual program above has a feasible solution if and only if the weighted graph has no negative cycle.

Proof. If there is a negative cycle C, so $\sum_{e\in C} w_e < 0$, then for any potential y we would have

$$0 = \sum_{uv\in C} (y_v - y_u) \leq \sum_{e\in C} w_e < 0.$$

Suppose there is no feasible solution. Let y have as few edges as possible failing the condition $y_v - y_u \leq w_e$. Take any failing edge $e_1 = v_1v_2$, so $y_{v_2} - y_{v_1} > w_{e_1}$. Fix that edge by setting $y_{v_2} = y_{v_1} + w_{e_1}$. By minimality of y, there must be new failing edges, which must have tail v_2. Fix each of these: if for instance $y_{v_3} - y_{v_2} > w_{e_2}$, set $y_{v_3} = y_{v_2} + w_{e_2} = y_{v_1} + (w_{e_1} + w_{e_2})$. Again by minimality, at least one of the fixes must create a new failing edge. We can keep repeating this, so at some point we must encounter a vertex for the second time. Then we have a sequence $v_1, v_2, \ldots, v_k, \ldots, v_l$, such that $e_l = v_lv_k$ fails, i.e. $y_{v_k} - y_{v_l} > w_{e_l}$. But we have set $y_k = y_1 + \sum_{i=1}^{k-1} w_{e_i}$ and $y_l = y_1 + \sum_{i=1}^{l-1} w_{e_i}$, so we have

$$w_{e_l} < y_{v_k} - y_{v_l} = -\sum_{i=k}^{l-1} w_{e_i},$$

which means that $v_kv_{k+1}\cdots v_lv_k$ is a negative cycle.

□

Note that this means that for a graph without a negative cycle, and with all vertices reachable from a, the primal problem is feasible and bounded, and a shortest path exists.
If there is a negative cycle, one could still ask for the shortest path, but this turns out to be an NP-hard problem (see the Problem Set). The trick is in the fact that the LP above actually looks for minimum *walks* (like a path, but allowing vertices to be repeated, i.e. they could have cycles), and a when there is a negative cycle walks can have arbitrarily large negative weight.

2.4 Ford's Algorithm

Ford's Algorithm for shortest paths from??? a weights without negative cycles

1. Set $d(a) = 0$ and $d(v) = \infty$ for $v \neq a$;
2. Repeat the following $|V(G)|$ times:
   - For each edge $uv \in E(G)$, do:
     - If $d(v) > d(u) + w(uv)$, then set $d(v) = d(u) + w(uv)$ and $p(v) = u$;
3. Return d and p.

Shortest ab-path: The predecessor function p defines a path from a to b, as for Dijkstra's algorithm.

Finding negative cycles: Suppose the algorithm finishes, but we still have $d(v) > d(u) + w(uv)$ for some edge. Then set $v_n = v$, v_{n-1}

Theorem 2.4.1. *For a weighted directed graph G without negative cycles, and a vertex a, Ford's algorithm correctly determines the distance function d (i.e. $d(v) = dist(a, v)$ for all $v \in V(G)$), and p determines a shortest path from any b.*

Proof 1. We claim that at the end of the algorithm, we have $d(v) - d(u) \leq w(uv)$ for all edges uv, and we have $d(v) - d(u) = w(uv)$ whenever $p(v) = u$. Both claims follow from the fact that there is a shortest path from a to u and to v (since there is no negative cycle).
It follows that $y_v = d(v)$ defines a potential, i.e. a dual feasible solution to the program above. Given b, going back using p, we get a path from a to b, i.e. a primal feasible solution x, that has $d(v) - d(u) = w(uv)$ along all its edges, Furthermore, these two feasible solutions have complementary slackness, i.e. we have $x_{uv} = 0$ whenever the dual constraint corresponding to uv has slack ($d(v) - d(u) < w(uv)$). By the complementary slackness theorem (see the Introduction), both solutions are optimal, which means that $d(v)$ correctly gives the distance to b, and x is a shortest ab-path. □

Proof 2. Let d_i be the function d after the ith iteration of step 2 (of the $|V(G)|$ iterations). Then

$$d_i(v) = \text{ minimum length of any } av\text{-path with } \leq k \text{ edges.}$$

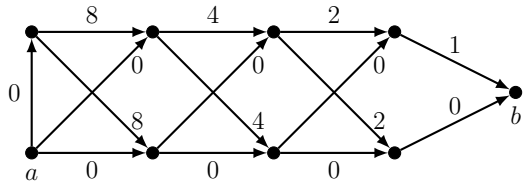
We can prove this by induction, using that there are no negative cycles.
It follows that $d(v) = d_{|V(G)|}(v) = \text{dist}(a, v)$, since a shortest path certainly has fewer than $|V(G)|$ edges. □

2.5 Problems

1. Given a weighted directed graph G that may have negative edges but no negative cycle, and given a potential on G, show that a shortest ab-path can be found by changing G in such a way that Dijkstra's algorithm works on it.

2. Given a sequence $a_1, \ldots, a_n$ of real numbers, we want to find a consecutive subsequence with minimal sum, i.e. $i \leq j$ such that $\sum_{k=i}^{j-1} a_k$ is minimal (the sum $= 0$ when $i = j$).
   Show that such a sequence can be found using a quick version of Ford's algorithm, where edges are corrected in an order such that each one only needs to be corrected once.
   Carry this out for the sequence $-1, 3, -2, 1, -1, -1, 5, -2, 3, -2$.

3. Consider the following version of Ford's algorithm, which is simpler than the one in the notes: Set $d(a) = 0$, $d(v) = \infty$ for $v \neq a$; then repeatedly pick any edge $uv \in E(G)$ such that $d(v) > d(u) + w(uv)$, set $d(v) = d(u) + w(uv)$; stop when there are no such edges left.
   Use the graph below, and generalizations of it, to deduce that its running time is not polynomial.



4. Show that if you had a polynomial algorithm that finds a shortest path in any weighted directed graph (allowing negative cycles), then it would give you a polynomial algorithm to determine if any unweighted directed graph has a Hamilton cycle.

(Solutions on page 75.)

Chapter 3

Flows

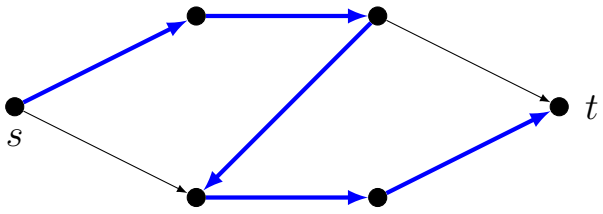
3.1 Disjoint Paths • 3.2 Flows

In this chapter, all graphs will be directed. The graphs will come with a source vertex s and a terminal vertex t. We will make the assumption that $\delta^{\text{in}}(s) = \emptyset$ and $\delta^{\text{out}}(t) = \emptyset$ (for the problems we are considering, these edges could be ignored anyway), and that the graphs have no parallel or reverse edges. All paths considered will be *directed* paths.

3.1 Disjoint Paths

EDGE-DISJOINT PATHS PROBLEM: Given a directed graph G and $s, t \in V(G)$, find the maximum number of edge-disjoint st-paths.

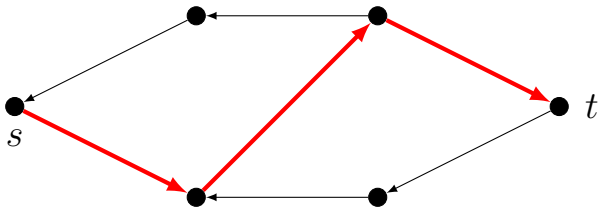
Greedy?
Let's try a greedy approach: find any st-path, remove it, find another one, etc. Here's a graph for which this will not give the maximum number of paths (2), if the first path we choose is the one in blue:



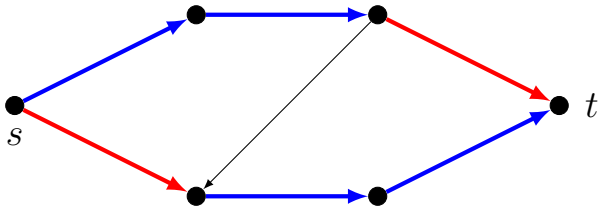
Reversal trick
We can fix the greedy approach with the following trick (although the result will no longer be greedy): Given a path, *reverse* its edges, and then look for the next path.
Write $(uv)^{-1} = vu$, for $S \subset E(G)$ write $S^{-1} = \{e^{-1} : e \in S\}$, and for any graph H write $H^{-1} = (V(H), E(H)^{-1})$.
After finding a path P, modify the graph to

$$G' = G - P + P^{-1}.$$

By this notation I just mean to remove the edges of P, then add their inverses; it does not mean that vertices of P, or adjacent edges, should be removed.
Below is the result for the graph from above, with a new path found in this graph, in red.



The good thing now is that the blue path P and the red path Q together give two edge-disjoint paths in the original graph, by removing the edge where they overlap (i.e. go against each other), and merging the remainders, like so:



This trick works more generally.

Lemma 3.1.1. *Given k edge-disjoint paths P_i in G, define $G' = G - \sum P_i + \sum P_i^{-1}$.*
If there is a path Q in G', then there are $k+1$ edge-disjoint paths R_i in G.

Proof. Let $S = (\bigcup E(P_i)) \cap E(Q)^{-1}$, i.e. the edges of the P_i that overlap (in reverse) with Q. Take $T = ((\bigcup E(P_i)) \setminus S) \cup (E(Q) \setminus S^{-1})$, i.e. the edges of Q and the P_i that remain after all the overlapping ones are removed, and $H = (V(H), T)$.
For all vertices $v \neq s, t$ we have $\deg_H^{\text{in}}(v) = \deg_H^{\text{out}}(v)$. This is because H is a union of paths, from which we only remove edges in reverse pairs; each time we remove a reverse pair of edges, their endpoints have their in-degree and out-degree lowered by 1.
Furthermore, $\deg_H^{\text{out}}(s) = k+1$, since we have k edges out of s for the paths R_i, and one more from Q. None of these is removed, since that would mean Q has an edge into s, which is impossible. It follows that from each of s's $k+1$ outgoing edges, we can trace a path that has to end in t, since at each vertex in-degree equals out-degree. These are $k+1$ edge-disjoint paths.
(Note that H may also contain cycles, but we can just ignore those.) □

Algorithm
This trick gives us the following algorithm.

Algorithm for edge-disjoint *st*-paths in an unweighted directed graph

1. Set $S = \emptyset$;
2. Let $G' = G - \sum_{P \in S} P + \sum_{P \in S} P^{-1}$.
3. Find an st-path Q in G'; if none exists, go to 5;
4. Set $S = \{R_i\}$ where R_i are as in the lemma; go back to 2;
5. Return S.

*Step 4:*Note that step 4 is stated indirectly, just because otherwise the notation would be a bit cumbersome. Basically, it is a subalgorithm that computes H and then finds the $k+1$ paths in H (for instance using BFS).

Polynomial running time: It should be clear that the algorithm is polynomial; we won't prove it here because it follows from the polynomiality of the flow algorithm below. As usual (in this course), one could implement it much more efficiently, by not completely recomputing the G' and H every time.

Linear programming formulation

We don't know yet if the algorithm above actually finds the maximum number of disjoint paths; we merely know that it finds a set of paths that cannot be improved on with the reversal trick. Let's see if linear programming can help.

We can write an LP in a similar way to the one we saw for shortest paths:

LP for Edge-Disjoint Paths

$$\text{maximize} \sum_{sw \in \delta^{\text{out}}(s)} x_{sw} \quad \text{with } 0 \le x \le 1, \ \boxed{x \in \mathbb{Z}^{|E|}},$$

$$\sum_{e \in \delta^{\text{in}}(v)} x_e - \sum_{e \in \delta^{\text{out}}(v)} x_e = 0 \quad \text{for } v \in V \backslash \{s,t\}.$$

But this isn't ideal, because unlike for shortest paths, we really need the $x \le 1$ condition, which makes the dual a bit annoying (we'd need a variable for every vertex constraint, and another variable for every $x_e \le 1$ constraint, see the dual for the maximum flow problem).

A more convenient LP is the one below, which uses the set $\mathcal{P} = \{st\text{-paths}\}$. Its constraints simply say that every edge can be contained in at most one chosen path.

LP for Edge-Disjoint Paths

$$\text{maximize} \sum_{P \in \mathcal{P}} x_P \quad \text{with } x \ge 0, \ \boxed{x \in \mathbb{Z}^{|\mathcal{P}|}},$$

$$\sum_{P \ni e} x_P \le 1 \quad \text{for } e \in E.$$

Not all feasible solutions of the relaxation are sets of disjoint paths, but an optimal solution x is a convex combination of maximum sets of disjoint paths, i.e. $x = \sum c_i s_i$ with $c_i \ge 0$, $\sum c_i = 1$, and each s_i corresponds to a set with the maximum number of disjoint paths. Again, we won't prove this here, because it will follow from a later theorem.

This form is much easier to dualize (the relaxation of):

Dual of relaxation

$$\text{minimize} \sum_{e \in E} z_e \quad \text{with } z \ge 0,$$

$$\sum_{e \in P} z_e \ge 1 \quad \text{for } P \in \mathcal{P}.$$

Separating sets: An integral feasible dual solution corresponds to a set of edges that we call an *st-separating set*: a set of edges such that every st-path contains at least 1 of these edges. In other words, removing these edges separates s from t.

Cuts: A *minimal* separating set is called an *st-cut* (or just *cut*): a set of edges of the form $\delta^{\text{out}}(S)$ for some $S \subset V(G)$ with $s \in S$, $t \notin S$. So a cut is a feasible dual solution, but an

integral feasible dual solution is only a cut if it contains no integral feasible solution with fewer edges.
(Note that not every text uses quite the same definition of cuts.)

Minimum cuts: A *minimum cut* is a cut with the minimum number of edges, i.e. an optimal dual solution. As for the primal problem, every optimal dual solution is a convex combination of integral optimal dual solutions, i.e. minimum cuts.

Theorem 3.1.2. *The algorithm finds a set with the maximum number of edge-disjoint paths.*

Proof. The proof is by duality: Given S that the algorithm has terminated with, we show that there is a cut with $|S|$ edges, which implies that both are optimal solutions.
Let $G' = G - \sum_{P\in S} P + \sum_{P\in S} P^{-1}$ as in the algorithm. Define

$$U = \{u \in V(G) : \exists su\text{-path in } G'\}.$$

Since the algorithm terminated with S, there is no st-path in G', so $t \notin U$, hence $\delta^{\text{out}}(U)$ is a cut (in G).
Suppose $uv \in \delta^{\text{out}}(U)$, with $u \in U$, $v \notin U$. Then $uv \notin G'$, otherwise we would have $v \in U$. Thus $uv \in P$ for some $P \in S$; we claim that this is a 1-to-1 correspondence between $\delta^{\text{out}}(U)$ and S. Clearly, every $P \in S$ has such an edge uv. Suppose P contains two edges leaving U; then it must also have an edge vu going *into* U, with $v \notin U$, $u \in U$. But this edge would have been reversed in G', which implies $v \in U$, contradiction.
So we have a primal feasible solution x corresponding to S and a dual feasible solution z corresponding to $\delta^{\text{out}}(U)$, with $\sum x_P = |S| = |\delta^{\text{out}}(U)| = \sum z_e$. By the duality theorem, both must be optimal solutions. □

3.2 Flows

The maximum flow problem is a generalization of the disjoint paths problem, and it is best introduced directly as a linear program. It is similar to the first LP for disjoint paths above, but instead of $x \le 1$ it has $x_e \le c_e$.

LP form of Flow Problem

$$\text{maximize} \sum_{e\in\delta^{\text{out}}(s)} x_e \quad \text{with}$$

$$\sum_{e\in\delta^{\text{in}}(v)} x_e - \sum_{e\in\delta^{\text{out}}(v)} x_e = 0 \quad \text{for } v \in V\backslash\{s,t\},$$

$$0 \le x_e \le c_e \quad \text{for } e \in E.$$

Definitions: A *network* is a directed graph G with a nonnegative function $c : E(G) \to \mathbb{R}$; $c_e = c(e)$ is called the *capacity* of e. A *flow* in a network is a function $f : E(G) \to \mathbb{R}$ such that $0 \le f(e) \le c_e$, and for every vertex, other than s and t, inflow equals outflow, i.e.

$$\sum_{e\in\delta^{\text{in}}(v)} x_e = \sum_{e\in\delta^{\text{out}}(v)} x_e.$$

The *value* of a flow f is $\sum_{e\in\delta^{\text{out}}(s)} x_e$; it is what we want to maximize.

Augmenting Path Algorithm
The algorithm for disjoint paths above also generalizes to an algorithm for flows. The way to

see it is that when we reverse the edges of the paths we have so far, we are saying that we cannot use these edges for the next path, but we could remove them again.
For flows, we do something similar with the flow that we have so far. For each edge, we see if we could still add to it (to increase the flow along that edge), and we see if we could remove from it (to be able to increase the flow along other edges). We store this information in an auxiliary graph, with a reverse edge wherever flow could be removed, find an st-path in that graph, and then increase the flow along that path as much as possible.

Algorithm for maximum flow in a network

1. Set all $f(e) = 0$, define G_f by $V(G_f) = V(G)$, $E(G_f) = \emptyset$;
2. Set $E_1 = \{e : f(e) < c_e\}$ and $E_2 = \{e : 0 < f(e)\}$;
   Take $E(G_f) = E_1 \cup E_2^{-1}$;
3. Find an st-path Q in G_f; if none exists, go to 6;
4. Compute $\alpha = \min\left(\min_{e\in Q\cap E_1}(c_e - f(e)), \min_{e\in Q\cap E_2^{-1}}(f(e))\right)$;
5. Augment along Q: $f(e) := \begin{cases} f(e)+\alpha & \text{for } e \in Q\cap E_1, \\ f(e)-\alpha & \text{for } e^{-1} \in Q \cap E_2^{-1}; \end{cases}$
   go back to 2;
6. Return f.

Theorem 3.2.1. *If all capacities c_e are rational, the algorithm terminates.*

Proof. Take M to be the least common multiple of the denominators of the c_e. Then in the algorithm α is always a positive integer multiple of $1/M$, so after each loop the value of f is increased by at least $1/M$. Since the value is bounded by $\sum_{e\in\delta^{\text{out}}(s))} c(e) = c(\delta^{\text{out}}(s))$ (or the capacity of any other cut, see below), the algorithm can take at most $M\cdot c(\delta^{\text{out}}(s))$ iterations. □

Non-termination: There are graphs with some irrational capacities such that the algorithm as given above does not terminate.

Non-polynomiality: There are also graphs with integer capacities for which the algorithm does not have polynomial running time.

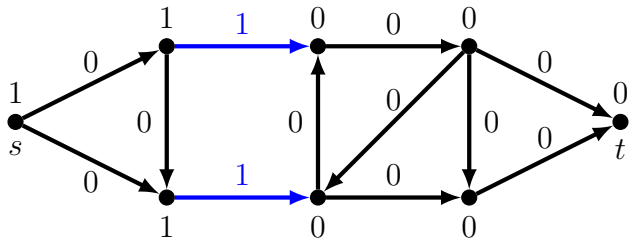
For both of these problems you will see an example in the problem set. Fortunately, both can be fixed by not picking the path Q randomly. But before we get to that, let's first check that when the algorithm terminates, it does actually give a maximum flow. For that we will need the dual. (recall that we assumed $\delta^{\text{in}}(s) = \emptyset$ and $\delta^{\text{out}}(t) = \emptyset$):

Dual of relaxation

$$\begin{aligned} \text{minimize } \sum_{e\in E} c_e z_e, \quad & z \geq 0,\ y \in \mathbb{R}^{|V|-2}, \\ y_v - y_u + z_{uv} \geq 0 \quad & \text{for } uv \in E, u \neq s, v \neq t, \\ y_w + z_{sw} \geq 1 \quad & \text{for } sw \in \delta^{\text{out}}(s), \\ -y_w + z_{wt} \geq 0 \quad & \text{for } wt \in \delta^{\text{in}}(t). \end{aligned}$$

Note that one could simplify the constraints by setting $y_s = 1$, $y_t = 0$, so that for really every $uv \in E$ the constraint has the form $y_v - y_u + z_{uv} \geq 0$. Indeed, for sw we then have $y_w - 1 + z_{sw} \geq 0$, and for wt we have $0 - y_w + z_{wt} \geq 0$, so in both cases we have the same constraint as above.

Cuts: A cut $\delta^{\text{out}}(U)$ with $s \in U, t \notin U$ gives a feasible dual solution: Set $y_u = 1$ for $u \in U$, $y_v = 0$ for $v \notin U$, and set $z_e = 1$ for $e \in \delta^{\text{out}}(U)$, $z_e = 0$ otherwise. In a picture it would look like this (the numbers at the vertices are the y_v, the ones at the edges are the z_e):



Theorem 3.2.2. *If the augmenting path algorithm terminates, then f is a maximum flow.*

Proof. We will show that there is a cut whose capacity equals the value of f, which proves by duality that f is maximum. Let

$$U = \{u : \exists \text{ an } su\text{-path in } G_f\}.$$

We have $s \in U$, and $t \notin U$, since otherwise the algorithm would not have terminated with f. So $\delta^{\text{out}}(U)$ is an st-cut.
For every $e = uv \in \delta^{\text{out}}(U)$ we must have $f(e) = c_e$, otherwise e would be an edge in G_f, and we would have $v \in U$, a contradiction. Furthermore, for every $e' = v'u' \in \delta^{\text{in}}(U)$ we must have $f(e') = 0$, since otherwise $e'^{-1} = u'v'$ would be an edge in G_f, which would mean $v' \in U$, again a contradiction.
It follows that

$$\sum_{e \in \delta^{\text{out}}(U)} c_e = \sum_{e \in \delta^{\text{out}}(U)} f(e) - \sum_{e \in \delta^{\text{in}}(U)} f(e) = \sum_{e \in \delta^{\text{out}}(s)} f(e).$$

This says exactly that the capacity of the cut equals the value of f. The last equality holds because in general, the value of f across any cut is the same. To prove it, sum up all the constraints of the linear program, then observe that for each edge, the terms will cancel, except for edges that leave U, enter U, or leave s:

$$0 = \sum_{v \in V(G)} \left(\sum_{e \in \delta^{\text{in}}(v)} f(e) - \sum_{e \in \delta^{\text{out}}(v)} f(e) \right) = \sum_{e \in \delta^{\text{out}}(U)} f(e) - \sum_{e \in \delta^{\text{in}}(U)} f(e) - \sum_{e \in \delta^{\text{out}}(s)} f(e).$$

□

Theorem 3.2.3 (Max-Flow Min-Cut)**.** *The maximum value of a flow equals the minimum capacity of an st-cut:*

$$\max_{\text{flows } f} \sum_{e \in \delta^{out}(s)} f(e) = \min_{\text{cuts } S} \sum_{e \in S} c_e.$$

Proof. If f is a maximum flow, then G_f has no st-path. As above, this gives a cut whose capacity equals the value of f. By duality, this cut must be minimum. □

Theorem 3.2.4. *If the algorithm always chooses a shortest path??? in G_f in step 3, then its running time is polynomial.*

Proof. Clearly, in every iteration some edge disappears from G_f, but others may appear when their reverse is on the augmenting path Q_f. We claim that thanks to choosing Q_f to be a shortest path, no edge can (re)appear more than $|V|$ times. This implies that there are at most $|E| \cdot |V|$ iterations, which proves the theorem.
To prove the claim, consider $d_f(v) = \text{dist}_{G_f}(s, v)$ (which we know is easily calculated by BFS). We first prove that d_f does not decrease, i.e. if the algorithm augments f to f', then $d_{f'}(v) \geq d_f(v)$ for all v. This follows by letting uv be the last edge on a shortest path from s to v in $G_{f'}$, so that

$$d_{f'}(v) = d_{f'}(u) + 1 \geq d_f(u) + 1 \geq d_f(v).$$

Here the equality follows because u comes before v on a shortest path, and the first inequality follows by induction. If $uv \in G_f$, then we clearly have $d_f(v) \leq d_f(u) + 1$, the third inequality. If $uv \notin G_f$, then it must have just appeared, which can only happen if $vu \in Q_f$, which implies $d_f(u) = d_f(v) + 1$, hence also $d_f(u) + 1 \geq d_f(v)$. This proves that $d_{f'}(v) \geq d_f(v)$ for all v.
To prove the claim, suppose $uv \in G_f$ but uv disappears in the augmentation from f to f'; and that later uv reappears in the augmentation from f'' to f'''. It follows that $uv \in Q_f$ before it disappears, and that $vu \in Q_{f''}$ before it reappears. These two facts imply

$$d_{f''}(u) = d_{f''}(v) + 1 \geq d_f(v) + 1 = d_f(u) + 2.$$

The two equalities follow because Q_f and $Q_{f''}$ are shortest paths, and the inequality follows from the observation that d_f only increases.
So whenever an edge uv reappears, $d_f(u)$ must have increased by 2. Since $d_f(u) \leq |V|$, the edge can certainly not reappear more than $|V|$ times. □

3.3 Problems

1. Show that, given an algorithm for finding a maximum set edge-disjoint paths in any directed *st*-graph, you can use it to find a maximum set of vertex-disjoint directed *st*-paths in any directed graph.
   Show that there is also a way to do this the other way around.

2. Give an integer program whose optimal solutions correspond to maximum sets of vertex-disjoint *st*-paths, in a given directed graph G with vertices s, t.
   Give the dual of the relaxation of this program. What objects in the graph do integral dual optimal solutions correspond to?

3. Use flows to give an algorithm for the *binary assignment problem*: Given a bipartite graph G with $c : E(G) \to \mathbb{Z}_{\geq 0}$ and $d : V(G) \to \mathbb{Z}_{\geq 0}$, find a maximum assignment, i.e. a $\varphi : E(G) \to \mathbb{Z}_{\geq 0}$ such that for all edges e we have $\varphi(e) \leq c(e)$ and for all vertices v we have $\sum_{e \in \delta(v)} \varphi(e) \leq d(v)$.

4. A *path flow* g is a flow such that $g(e) > 0$ only on the edges of one directed *st*-path.
   A *cycle flow* h is a flow such that $h(e) > 0$ only on the edges of one directed cycle.
   Prove that any flow f can be decomposed into path flows and cycle flows, i.e. there is a set $\mathcal{P}$ of path flows and a set $\mathcal{C}$ of cycle flows, with $|\mathcal{P}| + |\mathcal{C}| \leq |E(G)|$, such that

$$f(e) = \sum_{g \in \mathcal{P}} g(e) + \sum_{h \in \mathcal{C}} h(e) \quad \forall e \in E(G).$$

5. Use the first graph below to show that if the augmenting path algorithm chooses the path Q arbitrarily, then its running time is not polynomial.

Use the second graph below to show that if there are irrational capacities (here $\phi = (\sqrt{5}-1)/2$, so $\phi^2 = \phi - 1$), then the same algorithm may not terminate. Also show that it may not even converge to the right flow value.

(Solutions on page 77.)

Chapter 4

Bipartite Matchings

4.1 Introduction • 4.2 Maximum Weight Matchings • 4.3 Minimum Weight Perfect Matchings

4.1 Introduction

The graphs G in which we look for matchings will be undirected, sometimes weighted, sometimes not.
Recall that a *matching* in G is an $M \subset E(G)$ such that $m_1 \cap m_2 = \emptyset$ for all $m_1, m_2 \in M$. In other words, a set of edges that don't touch each other at any endpoint. A matching is *perfect* if $\cup_{m \in M} m = V(G)$, i.e. if every vertex is an endpoint of some edge of the matching (it is 'matched' or 'covered').

- Maximum Cardinality Matching Problem: In an unweighted graph, find a maximum with the maximum number of edges.
- Maximum/Minimum Weight Matching Problem: In a weighted graph, find a matching with maximum/minimum weight.
- Maximum/Minimum Weight Perfect Matching Problem: In a weighted graph, find a perfect matching, if it exists, such that its weight is maximum/minimum among all perfect matchings.

- Note that looking for a minimum cardinality matching would be silly. The maximum cardinality matching problem is a special case of the maximum weight matching problem, where all weights are 1.
- For the weighted problems, the maximum and minimum versions are equivalent: one can be turned into the other by multiplying all weights by -1.
- When looking for a maximum weight matching, one can ignore edges with negative weight, because they would never be included. For a minimum weight watching, one ignored positive-weight edges.
-The max/min weight matching and max/min weight perfect matching problems are equivalent; you will be asked to prove this in the problem set.

Let's right away consider the corresponding integer program and the dual of its relaxation:

IP for maximum weight matchings

$$\text{maximize} \sum_{e \in E(G)} w_e x_e \quad \text{with } x \geq 0, \ \boxed{x \in \mathbb{Z}^{|E|}},$$
$$\sum_{e \ni v} x_e \leq 1 \quad \text{for } v \in V.$$

Dual of relaxation

$$\text{minimize} \sum_{v \in V} y_v \quad \text{with } y \geq 0,$$
$$y_u + y_v \geq w_e \quad \text{for } uv \in E.$$

Theorem 4.1.1. *If the graph is bipartite, then optimal solutions of the integer program above are also optimal for the relaxation.*

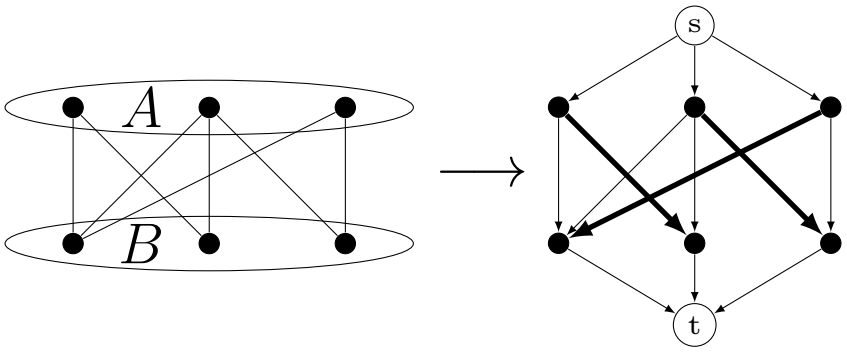
We will see a proof of this in a later lecture. We get the maximum cardinality matching problem if we choose all $w_e = 1$. An integral optimal dual solution, which must have all values 0 or 1, is then called a *vertex cover*: a set of vertices such that every edge has at least one of them as endpoint. The following standard theorem from graph theory now follows directly from the duality theorem:

Theorem 4.1.2 (König)**.** *In a bipartite graph the maximum cardinality of a matching equals the minimum cardinality of a vertex cover.*

4.2 Bipartite Maximum Cardinality Matchings

We will first consider bipartite graphs, for which it is considerably easier to find matchings. So assume in this section that G is bipartite, with parts A and B. We will focus on the maximum weight matching problem.

One can reduce the maximum cardinality matching problem to a maximum flow problem, as follows. Direct all edges of G from A to B, and add vertices s and t, with an edge from s to all vertices of A, and an edge from each vertex of B to t. Give all edges capacity 1.
Here is an example of what it looks like, with the capacities left out, and a matching with thick edges.



Now let f be an integral st-flow in this graph, and let $M = \{e \in E(A,B) : f(e) = 1\}$. Then M is a matching: if $f(ab) = 1$ and $f(ab') = 1$, then f would fail the flow constraint at a:

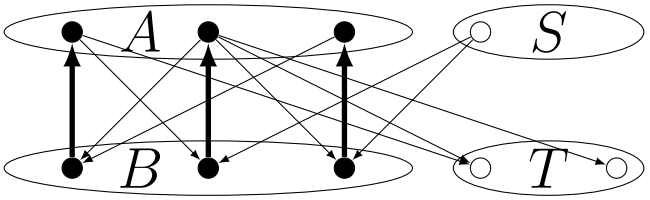
$$\sum_{e \in \delta^{\text{out}}(a)} f(e) \geq f(ab) + f(ab') > f(sa) = \sum_{e \in \delta^{\text{in}}(a)} f(e).$$

Conversely, given any matching M, we get a flow defined by $f(e) = 1$ for $e \in M$, $f(e) = 0$ for $e \notin M$; $f(sa) = 1$ if there is an $ab \in M$, $f(sa) = 0$ otherwise; and $f(bt) = 1$ if there is an $ab \in M$, $f(bt) = 0$ otherwise.

So integral flows in this graph correspond exactly to matchings, and the value of a flow equals the cardinality of the corresponding matching,

$$\sum_{e \in \delta^{\text{out}}(a)} f(e) = |\{a \in A : \exists ab \in M\}| = |M|.$$

Hence the augmenting path algorithm for maximum flows gives a polynomial algorithm for maximum cardinality matchings (we do not have to worry about irrational capacities, since all capacities are 0 or 1). Actually, we could even use the edge-disjoint path algorithm that we saw.
Here is an example of what the auxiliary graph G_M, corresponding to a matching M during the algorithm, will look like, ignoring s and t:



The algorithm will look for a path from S to T, because that is the only way from s to t, since the edges from s to matched vertices in A are at capacity, as are the edges from matched vertices in B to t. That path will be *alternating*, in the sense that it alternates matching-edges and non-matching-edges, with one more non-matching than matching. Given such an alternating ST-path, the algorithm will augment on it, which means it will swap matching and non-matching edges on the path, increasing their number.
For instance, there is a path from S to the second vertex of B, to the second vertex of A, to T. Augmenting along it will replace its one matching edge with the two other edges.
Let's state the resulting algorithm.

Augmenting path algorithm for bipartite maximum cardinality matchings

1. Set $M = \emptyset$ and $V(G_M) = V(G)$;
2. Set $E(G_M) = \{ba : \{a, b\} \in M\} \cup \{ab : \{a, b\} \notin M\}$;
   Let $S = A \backslash (\cup M)$, $T = B \backslash (\cup M)$;
3. Find any ST-path Q in G_M; if none exists, go to 5;
4. Augment along Q: $M := M \Delta E(Q)$; go back to 2;
5. Return M.

Just for practice, and because we'll need the ideas below, let's prove correctness:

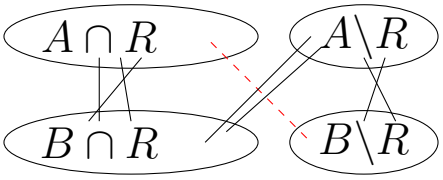
Theorem 4.2.1. *The augmenting path algorithm returns a maximum cardinality matching M in the bipartite graph G.*

Proof. We will show that there is a vertex cover C with $|C| = |M|$, which proves by duality that M is maximum.
Given S and T, the sets of unmatched vertices, define

$$R = \{v \in V(G) : \exists \text{ a path in } G_M \text{ from } S \text{ to } v\},$$

and consider

$$C = (A \backslash R) \cup (B \cap R).$$



We claim that C is a vertex cover in G. If not, then there is an edge not touching C, which means it is an edge ab between $A \cap R$ and $B \backslash R$. It cannot be in the matching, for then it would be oriented ba in G_M and the only edge of G_M that enters a, so a would not be in R since b is not. But since it is not in the matching, it is oriented ab, and it would provide an augmenting path by extending the path from S to a.
Now for $|C| = |M|$. Duality (or König) right away gives $|M| \leq |C|$. To prove that $|C| \leq M$, we show that every $v \in C$ is matched by a different $m \in M$. We have $S \subseteq A \cap R$ and $T \subseteq B \backslash R$, so every $v \in C$ is matched by some $m \in M$. And no $a, b \in C$, so $a \in A \backslash R$ and $b \in B \cap R$, can be matched by the same $m \in M$, since that would imply $a \in R$, a contradiction. □

4.3 Bipartite Maximum Weight Perfect Matching

In the weighted cases, it turns out to be a bit more convenient to formulate the algorithm for perfect matchings. So we need the corresponding linear programs, which only differ in that there is equality in the primal constraints (and hence y need not be nonnegative).

LP for maximum weight perfect matchings

$$\text{maximize} \sum_{e \in E(G)} w_e x_e \quad \text{with } x \geq 0,$$
$$\sum_{e \ni v} x_e = 1 \quad \text{for } v \in V.$$

Dual

$$\text{minimize} \sum_{v \in V} y_v \quad \text{with } y \in \mathbb{R}^{|V|},$$
$$y_u + y_v \geq w_{uv} \quad \text{for } uv \in E.$$

We will make use of the Complementary Slackness Theorem, which we state here specifically for matchings (it's not hard to prove directly, and you will be asked to do so in the problem set). Note that if we didn't have equality in the primal constraint, then there would be more conditions (that if $y_v = 0$ then the corresponding primal constraint is tight); this is why perfect matchings are more convenient here.

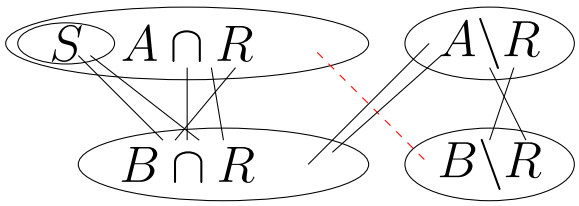
Lemma 4.3.1. *Let x be a feasible primal solution and y a feasible dual solution to the linear programs for maximum weight perfect matchings.*
If $x_{ab} = 0$ whenever $y_a + y_b > w_{ab}$, then both are optimal solutions.

We first describe the algorithm informally.
Given a feasible dual solution y, we will look at the graph G_y of tight edges, defined by $V(G_y) = V(G)$ and

$$E(G_y) = \{ab \in E(G) : y_a + y_b = w_{ab}\}.$$

We find a maximum cardinality matching M_y using the algorithm above. If it is perfect in G_y, then it will also be a perfect matching in G. Since it is complementary to y, both must be optimal by the lemma, so M_y is a maximum weight perfect matching.
If M_y is not perfect, we will try to improve y by lowering its value, in such a way that at the

same time we create new tight edges. I.e. we make at least one new edge satisfy $y_a + y_b = w_{ab}$, which will hopefully make it possible to find a larger matching among the tight edges.
To make this work we use ideas from the augmenting path algorithm and its correctness proof above. We will use the directed graph G_y, with matching edges directed from B to A, and non-matching edges directed from A to B. And we again consider the set R of vertices reachable in G_y from its uncovered vertices in A. This gives a configuration like this in G_y:



As we saw before, $C_R = (A\backslash R) \cup (B \cap R)$ is then a vertex cover in G_y, so there are no tight edges between $A \cap R$ and $B\backslash R$. That allows us to lower all the y_a's in $A \cap R$, while simultaneously increasing the y_b's in $B \cap R$, in such a way that at least one of the previously-slack edges between $A \cap R$ and $B\backslash R$ will become tight.
We will state the algorithm, and then prove in more detail that it works. For convenience, we assume that G has a perfect matching, which also implies that $|A| = |B|$. We could easily check this before starting the algorithm.

Primal-dual algorithm for bipartite maximum weight perfect matchings

1. Set $M_y = \emptyset$ and $V(G_y) = V(G)$;

   Define y by $y_a = 0$ for $a \in A$, $y_b = \max_{e \in \delta(b)} w(e)$ for $b \in B$;

2. Set $E(G_y) = \{ab \in E(G) : y_a + y_b = w_{ab}\}$;

   Find a maximum cardinality matching N in G_y;

   if $|N| > |M_y|$, set $M_y := N$ (else leave M_y unchanged);

   if M_y is perfect in G, go to 5; if not, set $S = A\backslash(\cup M)$;

3. Orient each $ab \in E(G_y)$ from B to A if it is in M_y, from A to B otherwise;

   Set $R = \{v \in V(G) : \exists$ a dipath in G_y from S to $v\}$;

   Compute $\delta = \min_{ab \in E_G(A \cap R, B \backslash R)} (y_a + y_b - w_{ab})$;

4. Augment by δ: $\begin{cases} y_a := y_a - \delta \text{ for } a \in A \cap R, \\ y_b := y_b + \delta \text{ for } b \in B \cap R; \end{cases}$

   go back to 2;

5. Return y, M_y.

Here is a simpler version of the algorithm that we could have tried: If there is no perfect matching in G_y, then find a set $X \subset A$ such that $N(X) < X$ (like our $A \cap R$); decrease all y_a in X by an appropriate $\delta > 0$ and increase all y_b in $N(X)$ by δ; then $\sum y_v$ will strictly decrease

by a multiple of δ as above. If the weights were integral, then we would have $\delta \geq 1$ throughout, so the algorithm would terminate. But it would not be polynomial, at least not if the weights can be arbitrarily large.
This is why we need to use the non-perfect matching like we do above. To deal with non-integral weights and to prove polynomiality, we need the more detailed observation about $|M_y|$ and $|B \cap R|$ in the lemma below.
Also note that it is not true that the number of tight edges always increases (which would have made things simpler); as we'll see in the proof below, edges between $B \cap R$ and $A \backslash R$ may go from tight to slack.

Lemma 4.3.2. *The algorithm is well-defined. After an augmentation (step 4), y is still feasible, its value $\sum_{v \in V} y_v$ has decreased, and either $|M_y|$ or $|B \cap R|$ has increased.*

Proof. Note that S is nonempty, because we assumed $|A| = |B|$, so when a matching is not perfect, it must leave vertices unmatched on both sides. Also $A \cap R \neq \emptyset$, since it contains S, and $B \backslash R \neq \emptyset$, since it contains T (the unmatched vertices in B, which satisfy $|T| = |S|$). The minimum exists because $E_G(A \cap R, B \backslash R) \neq \emptyset$: otherwise $A \cap R$ could never be perfectly matched, because $N(A \cap R) = B \cap R$, and below we will show $|B \cap R| < |A \cap R|$. And finally $\delta > 0$, since the edges between $A \cap R$ and $B \backslash R$ cannot be in G_y, otherwise they would create an augmenting path. So the steps are indeed well-defined.
Let y' be the new dual solution. For an edge ab between $A \cap R$ and $B \cap R$, the constraint is unchanged:

$$y'_a + y'_b - w_{ab} = (y_a - \delta) + (y_b + \delta) = y_a + y_b \geq w_{ab}.$$

Edges between $A \backslash R$ and $B \backslash R$ are clearly unaffected. For edges ab between $B \cap R$ and $A \backslash R$, only δ is added to y_b, which cannot break the constraint. It could lose tightness, but that will not be a problem. Finally, edges ab between $A \cap R$ and $B \backslash R$ will not get their constraints broken because of the way we choose δ, and at least one of these must become tight. This proves that the new y' is feasible.
To show that $\sum y_v$ strictly decreases, we show that $|A \cap R| > |B \cap R|$. This follows from

$$|A \cap R| = |A| - |A \backslash R| = \frac{|V|}{2} - |A \backslash R| > |C_R| - |A \backslash R| = |B \cap R|,$$

where we used that $|C_R| = |M_y| < \frac{|V|}{2}$, which follows from the fact that M_y is not perfect.
Suppose $|M_y|$ does not increase. Then by design of step 2, M_y remains the same. Since we have created at least one new tight edge in $ab \in E(A \cap R, B \backslash R)$, b will be in $B \cap R$ in the next step. No vertex can leave $B \cap R$, since the only edges that can lose tightness are in $E(B \cap R, A \backslash R)$, and removing these from G_y cannot break any path from S, so this does not affect R. Hence $|B \cap R|$ will have increased by at least 1. □

Theorem 4.3.3. *The primal-dual algorithm terminates, is polynomial, and returns a maximum weight perfect matching.*

Proof. The returned matching is clearly perfect, and it is maximum since it is a matching in the last G_y, which implies that it satisfies complementary slackness with the feasible dual solution y.
The algorithm terminates in polynomial time by the last statement of the lemma: There can be at most $\frac{|V|}{2}$ steps in which $|M_y|$ increases, and in between those steps there can be at most $|B| = \frac{|V|}{2}$ steps in which M_y remains the same but $|B \cap R|$ increases. So there are fewer than $|V|^2$ iterations, and each iteration is also polynomial: the only expensive things it does is the maximum cardinality matching algorithm to find N, and it does a breadth-first search to determine R. □

A few words on primal-dual algorithms in general. The algorithm above is a special case of a linear programming algorithm called *primal-dual*, which similarly works with a feasible dual, tries to find a feasible primal that satisfies complementary slackness, and if it fails it tries to improve the feasible dual. For linear programming in general it is not polynomial, but it turns out to work very well for linear programs that correspond to combinatorial optimization problems.
Many of the algorithms that we have seen (including Dijkstra's algorithm and the maximum flow algorithm) can be derived from this general method, without needing much insight. But this derivation is somewhat tedious, which is why I have only shown the result for the example of weighted perfect matchings. In case you go on to study more advanced combinatorial optimization, you will definitely see more of the primal-dual method.

4.4 Problems

1. Show that the maximum weight matching problem and the maximum weight perfect matching problem are equivalent, in the sense that if you have a polynomial algorithm for one, then you also have a polynomial algorithm for the other.

2. Show in two ways that if a bipartite graph is k-regular (every vertex has degree $k \geq 1$), then it has a perfect matching: once using linear programming, and once using König's Theorem.

3. Prove complementary slackness for bipartite maximum weight perfect matchings (Lemma 4.4 in the notes): If we have a primal feasible x and a dual feasible y such that for all $e \in E(G)$ either $x_e = 0$ or $y_a + y_b = w_{ab}$, then both are optimal.

4. Let G be a graph, not necessarily bipartite. Prove that a matching M in G is maximum if and only if there is no augmenting path for M. (a path between two unmatched vertices that alternates M-edges and non-M-edges).
   So why can't we use the augmenting path algorithm for nonbipartite graphs?

(Solutions on page 80.)

Chapter 5

General Matchings

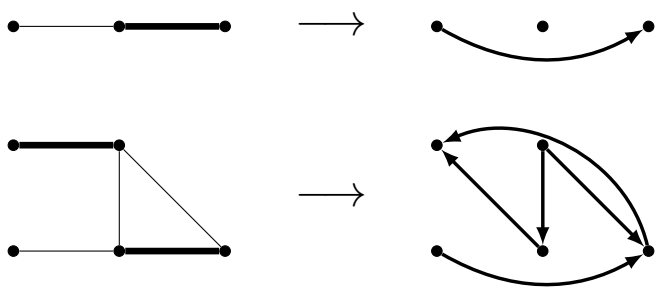
5.1 Flowers and blossoms • 5.2 Blossom Algorithm • 5.3 Postman Problem

Throughout this chapter G will be an undirected graph. A few definitions that I haven't mentioned yet: A *walk* in G is a sequence $v_0v_1\cdots v_m$ of vertices such that each $v_iv_{i+1} \in E(G)$ (so a walk is like a path, but vertices may occur more than once). A walk is *closed* if $v_m = v_0$. A *tour* is a closed walk that passes through all the vertices. There are corresponding definitions for directed graphs, but we will not need those here.

5.1 Flowers and blossoms

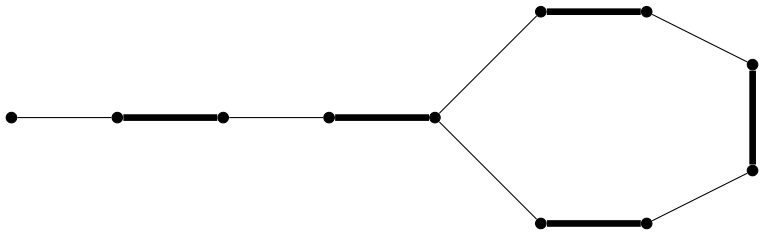
As we saw, given a matching in any graph, bipartite or not, the matching is not maximum if and only if there exists an augmenting path. For bipartite graphs, we could find such an augmenting path using a simple trick, directing the matching edges from B to A and non-matching edges in the other direction. For nonbipartite graphs, this trick is not available, and this makes finding augmenting paths harder.
We can try to do it as follows. Given a graph G with matching M, define a directed graph D_M by $V(D_M) = V(G)$ and

$$E(D_M) = \{ab : \ \exists v \in V(G) \text{ with } av \in E(G), vb \in M\}.$$

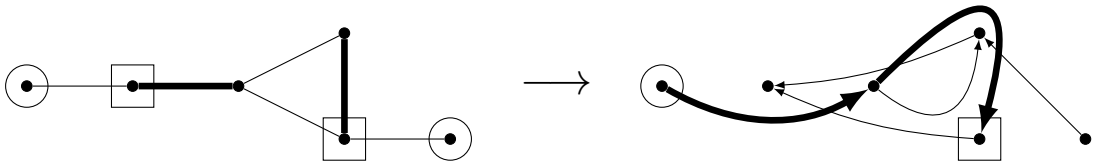


Simply put, an edge in D_M corresponds to a non-matching edge followed by a matching edge in G. Hence a (directed) path P in D_M gives an M-alternating *walk* in G; we will denote this (undirected) walk by $\pi(P)$.
Define, as before, $S = V(G)\backslash(\cup M)$, the set of vertices unmatched by M. Given an unmatched vertex $a \in S$, an M-augmenting path corresponds (if we omit the last edge) to a path in D_M from a to a neighbor of another unmatched b, i.e. a path from a to $N(S\backslash a)$. Finding such a path P in D_M is easy, but the problem is that $\pi(P)$ might not be a path, but a walk.
Fortunately, when $\pi(P)$ is not a path, it will have a specific form. We define an *M-flower* in G

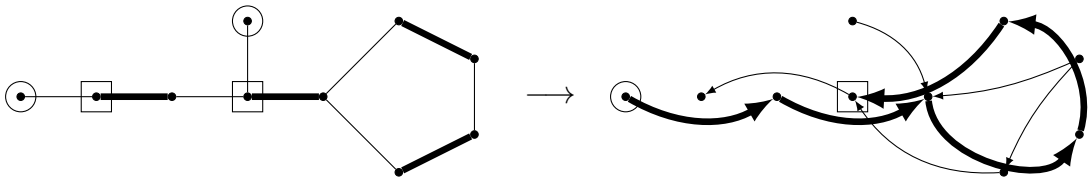
to be an even M-alternating path $v_0v_1\cdots v_t$ with another edge v_tv_i for some even $i < t$. So an M-flower consists of an even M-alternating path $v_0\cdots v_i$, attached to an odd cycle $v_i\cdots v_tv_i$, which is almost alternating, except that v_iv_{i+1} and v_tv_i are both not from M. We call that odd almost-alternating cycle an *M-blossom*, and the even path the *stem*.



The following pictures illustrate the two things that can basically happen when we find a path from S to $N(S)$ in D_M. Vertices in S are labelled with a circle, vertices in $N(S)$ with a square. The thick edges in D_M form a directed path from S to $N(S)$.
In the first picture, the path in D_M corresponds to an augmenting path in G (minus its last edge).



In the second picture, we have a path from S to $N(S)$ in D_M, but it does not correspond to an alternating path in G, but to an alternating *walk*. That walk in fact forms an M-flower.



If we took any path, more complicated things could happen, but if we take *shortest* paths, then the two situations above are basically the only possibilities. The only exception (which complicates the statement of the lemma below a bit) is when $\pi(P)$ is a path that ends up at a neighbor of its first vertex, which is not a neighbor of any other vertex from S. But in that case, adding one edge turns $\pi(P)$ into an M-flower whose stem is just one vertex.

Lemma 5.1.1. *Let P be a shortest path form S to $N(S)$ in D_M. Then either $\pi(P)$ can be extended by one edge to an M-augmenting path, or it is an M-flower, or it can be extended by one edge to an M-flower.*

Proof. If $\pi(P)$ is a path but cannot be extended to an augmenting path, that must be because its last vertex in $N(S)$ is a neighbor of its first vertex v_0, but not of any other vertex in S. Then adding the edge to v_0 gives an M-flower (whose stem is just v_0). This is exactly the third option in the statement.
Assume $\pi(P)$ is not a path and write it as $v_0v_1\cdots v_k$. Choose $i < j$ with $v_i = v_j$ and j as small as possible, so $v_0v_1\cdots v_{j-1}$ is a path.
The three edges $v_{i-1}v_i$, v_iv_{i+1} and $v_{j-1}v_j = v_{j-1}v_i$ all touch v_i, so only one of them can be a matching edge, and either $v_{i-1}v_i$ or v_iv_{i+1} must be one, because the path is alternating. So $v_{j-1}v_i$ is not a matching edge, and $v_jv_{j+1} = v_iv_{j+1}$ is a matching edge (possibly equal to v_iv_{i-1} or v_iv_{i+1}), and j is odd.
If v_iv_{i+1} were the matching edge touching v_i, then we must have $v_{j+1} = v_{i+1}$, which means

$v_0v_1\cdots v_iv_{j+1}\cdots v_k$ is a shorter $SN(S)$-path $v_0v_2\cdots v_{i-1}v_{j+1}\cdots v_k$ in D_M, contradiction.
So $v_{i-1}v_i$ is the matching edge touching v_i, which implies that i is even, which means that $v_0v_1\cdots v_{j-1}v_i$ is an M-flower. □

Note that in a bipartite graph, there are no odd cycles, so no blossoms, hence a shortest $SN(S)$-path automatically gives an M-augmenting path. This is what we used in the augmenting path algorithm, except that there was a simpler version of D_M, thanks to the bipartiteness.
To get an algorithm for general graphs, we would have to find a shortest $SN(S)$-path in D_M that is not a flower. Unfortunately, we don't know an algorithm for finding a path under such a restriction (just as we don't have an algorithm for finding a path between two sets with the restriction that it is alternating). Of course, we could just look for *all* $SN(S)$-paths, until we find one that is not a flower, but this will take exponential time.
The idea that solves this problem was introduced by Jack Edmonds, in a paper from 1965 (which, by the way, introduced the notion of a polynomial-time algorithm). We can *shrink* the blossom to a point, and then look for an augmenting path in the resulting graph, which has fewer vertices. As we prove below, an augmenting path in the graph with shrunken blossom gives an augmenting path in the original graph.
So we can take a shortest $SN(S)$-path in D_M; if it is an alternating path in G, then we can augment M right away; if not, we have a flower, so we shrink the blossom and recursively find an augmenting path in the smaller graph (which might involve more shrinking), which gives us an augmenting path in G on which we augment M. We also prove that if the shrunken graph has no augmenting path, then neither does G, so M is maximum.

First we introduce notation for shrinking a subgraph H of G. We formally define G/H by

$$V(G/H) = (V(G)\backslash V(H)) \cup \{h\}, \quad E(G/H) = (E(G)\backslash(E(H) \cup \delta(H))) \cup \{ah : a \in N(H)\}.$$

So we remove the vertices of H, replace them by a single vertex h, then remove all edges that touch a vertex of H, and connect a vertex to h if it was previously connected to some vertex of H. For a matching M in G, viewed as a set of edges, we define M/H by

$$M/H = (M\backslash(E(H) \cup \delta(H))) \cup \{ah : \exists b \in H \text{ with } ab \in M\}.$$

So we remove from M any edges inside H, and replace edges to H by edges to h. Note that in general, M/H need not be a matching, because it might have more than one edge touching h, if before there was more than one matching edge in $\delta(H)$. Fortunately, if H is a blossom, then M/H will be a matching.

Lemma 5.1.2. *Let B be an M-blossom in G. Then M is a maximum-size matching in G if and only if M/B is a maximum-size matching in G/B.*
Also, from an M/B-augmenting path in G/B one can construct an M-augmenting path in G in polynomial time.

Proof. Suppose M is not maximum, so there is an M-augmenting path P in G. If P is vertex-disjoint from B, then P is also an M/B-augmenting path in G/B, so assume P passes through one or more vertices of B. For convenience, we can assume that B has an unmatched vertex r, since we can swap matching and non-matching edges along the stem to make this the case. This implies that the shrunken vertex b in G/B is unmatched. One of the endpoints of P must be distinct from r, call it s. Then starting from s, P must enter B by a non-matching edge uv. In G/B, the part of P starting from s and ending with ub is an M/B-augmenting path.
Suppose M/B is not maximum, so there is an M/B-augmenting path P in G/B. If P does not pass through b, then P is also an M-augmenting path in G, so assume it does. If b is one of the unmatched endpoints of P, then B has an unmatched vertex r, and we can replace b in

P with the path from the first vertex of B that it meets, along one of the sides of the blossom (whichever one gives an alternating path), to b. This is an M-augmenting path in G. If b is not an endpoint of P, then P must have a matching edge and a non-matching edge touching b. Replacing b with one of the two paths along the blossom then makes P into an M-augmenting path in G. The second statement follows. □

The second statement of the lemma needs to be included, because it is crucial to the algorithm, but it does not follow directly from the first statement of the lemma.

5.2 The Blossom Algorithm

Now we can give the algorithm. Because it is recursive (rather than just iterative like our previous algorithms), we'll have to separate the finding of an augmenting path into a subroutine.

`blossom`(G, M) (returns an M-augmenting path in G, or "none" if none exists)

1. Set $S = V(G) \backslash (\cup M)$;
2. Set $D_M = \{ab : \exists v \in V(G) \text{ with } av \in E(G), vb \in M\}$ (a directed graph);
3. Find a shortest path P in D_M from S to $N_G(S)$; if none exists, stop, return "none";
4. If $\pi(P)$ can be extended (by one edge) to an augmenting path P', then return P';

   Otherwise it gives an M-flower, which has a blossom B;

   Compute $Q =$`blossom`$(G/B, M/B)$;

   Return the M-augmenting path in G constructed from Q as in Lemma 5.2.

Blossom algorithm for a maximum cardinality matching in G

1. Set $M = \emptyset$;
2. Greedily add as many edges to M as possible;
3. Compute $P =$`blossom`(G, M); if $P =$"none", go to 5;
4. Augment M on P; go to 3;
5. Return M.

Note that step 2 won't effect the worst-case running time much, but it makes a huge difference when actually doing an example.

Theorem 5.2.1. *The blossom algorithm terminates in polynomial time, and correctly returns a maximum matching.*

Proof. The number of augmentations is bounded by $|V|$, and in each augmentation step, the number of recursive calls to `blossom` is also bounded by $|V|$, since $|G/B| < |G|$ each time. And in each recursive call, finding the path P in G/B takes $O(|E|)$ steps, and extending it to a path in G is polynomial by Lemma 5.2 (in fact it is $O(|V|)$). So the algorithm terminates in

polynomial time (it's $O(|V|^4)$).
When it terminates with M, the last recursive call to `blossom` must have returned "none", so after some sequence of shrinkings, D_M has no $SN(S)$-path at all, which means that there is no augmenting path in that shrunken graph, which implies by (repeated applications of) Lemma 5.1.2 that there is no M-augmenting path in G. Therefore M is maximum. □

5.3 Postman Problem

Just like for bipartite graphs, there is also an algorithm for maximum weight matchings or minimum weight perfect matchings for general graphs. However, we will not cover it in this course, not because it is too difficult, but just to save some time. Another reason is that it doesn't really require new ideas, but it combines things that we've already seen, namely shrinking blossoms and the primal-dual approach.
We will state the existence of this algorithm here as a theorem (without proof), because we will use it several times, including in this section on the postman problem. So we will use this algorithm as a "black box".

Theorem 5.3.1. *There is a polynomial-time algorithm that finds a minimum weight perfect matching in a graph G with weights $w : E(G) \to \mathbb{R}$, if one exists.*

The postman problem is related to the Euler tour problem, which you have seen in the first problem set. We recall the facts quickly.

- An *Euler tour* in an undirected graph is a tour that uses every edge exactly once.
- A graph G has an Euler tour if and only if it is connected and each vertex has even degree.
- There is a polynomial algorithm that finds an Euler tour in any connected graph with even degrees. We describe it informally.
  From a given vertex, trace a closed walk using that every vertex it enters must have an unused edge, until we can't go on, which must be because we are back at the starting vertex. Then remove the edges of this closed walk, and from any vertex of the walk that still has edges, recursively find a closed walk using this vertex, and insert this into the main closed walk.
- This algorithm also works for *multigraphs*, where there are allowed to be multiple edges between two vertices.

The Euler tour algorithm, Dijkstra's algorithm, and the minimum weight perfect matching algorithm (as a black box) together give an algorithm for the following problem. (It is sometimes oddly called the *Chinese* postman problem, because it was first studied by a Chinese mathematician. A more politically correct name is *postperson* problem, and a more boring one is *route inspection* problem).

POSTMAN PROBLEM: Given a connected graph G, with weight function $w : E \to \mathbb{R}_{\geq 0}$,
find a minimum-weight tour that uses each edge *at least* once (a minimum postman tour).

Postman algorithm for a connected graph G

1. Let $U = \{v \in V(G) : \deg(v) \text{ odd}\}$;
2. For each pair $u, v \in U$, find a shortest uv-path P_{uv} in G with weight w;

3. Let H be the complete graph on U, with weights $d(uv) = w(P_{uv})$;
4. Find a minimum-weight perfect matching M in H with weights d;
5. Construct a multigraph G' by adding to G a second copy of each edge in each P_{uv};
6. Find an Euler tour T in G', and return T (as a walk in G).

Theorem 5.3.2. *The postman algorithm terminates in polynomial time and correctly returns a minimum postman tour.*

Proof. To check well-definedness, observe that shortest paths exist because the weights are nonnegative, that perfect matchings exist in H since it is complete, and that an Euler tour exists in G' since we have added an odd number of edges at each odd-degree vertex, and an even number of edges at each other vertex (actually, exactly 1 is added at odd-degree vertices, and 0 or 2 otherwise, but we won't need that).
The algorithm is clearly polynomial, since we find a shortest path $\leq |V|^2$ times (probably with Dijkstra, since the weights are nonnegative), and then use minimum-weight perfect matching and Euler tour algorithms once each.
The found tour T is clearly a tour and visits each edges at least once. We just have to prove that it is minimum. The edge set of any postman tour gives a function $n : E(G) \to \mathbb{Z}_{\geq 1}$ (the number of times it uses that edge). Then $N = \{e : n(e) > 1\}$ consists of paths P_i between odd-degree vertices, and cycles (by a constructive argument that we've seen several times now). If $2m$ is the number of odd-degree vertices, than the number of paths is m. So for any postman tour we have

$$\sum_{e \in E} n(e) = |E| + \sum_{e \in N} (n(e) - 1) \geq |E| + \sum_{i=1}^{m} w(P_i).$$

But the sum on the right is exactly what is minimized for T, so T is minimum. □

5.4 Problems

1. Consider the following greedy algorithm for matchings:

   Repeatedly add uv such that $\deg(u) + \deg(v)$ *is minimal, then remove* $\delta(u) \cup \delta(v)$.

   Give an example for which this algorithm does cannot return a maximum cardinality matching.

2. *Execute the blossom algorithm to find a maximum cardinality matching for the following example, starting with the given matching.*

3. Execute the postman algorithm to find a minimum postman tour in the following example (find the minimum weight perfect matching by inspection).

4. Show that Dijkstra's algorithm can be used to find a shortest st-path in an undirected weighted graph, if all weights are nonnegative. Why does this fail when there are negative weights?
   Give an algorithm for shortest st-paths in an undirected graph G with arbitrary weights, but without negative cycles.

(Solutions on page 81.)

Chapter 6

Integer Programs

6.1 Introduction • 6.2 Total Unimodularity • 6.3 Integrality Theorems • 6.4 Matching Polytopes

6.1 Introduction

We will give a simplified and not-too-rigorous introduction to polytopes. Most of the statements about polytopes are also true for polyhedra, but often there are some subtleties or weird exceptions. Since most of the optimization problems that we'll see correspond to polytopes, we'll restrict ourselves to those.

Definitions:
• A *polyhedron* is a set $P \subset \mathbb{R}^n$ of the form $P = \{x : Ax \leq b\}$, for an $m \times n$ matrix A and $b \in \mathbb{R}^m$.
• A *polytope* is a bounded polyhedron. It can be shown that this is equivalent to being the convex hull of a finite set of points.
• A *vertex* of a polytope P is a point $z \in P$ for which there is a hyperplane H such that $P \cap H = z$. Here a hyperplane is a set $\{x : \sum a_i x_i = c\} \subset \mathbb{R}^n$, where not all a_i are 0.
One can prove that a polytope is the convex hull of its vertices.

Lemma 6.1.1. *For any vertex z of a polytope, there are a nonsingular $n \times n$ submatrix A_z of A and a subvector b_z of b such that z is the unique point satisfying $A_z z = b_z$.*

We won't prove this here. To explain it a bit, observe that rows of A correspond (typically) to hyperplanes that make up the boundary of of the polytope, and a vertex is an intersection of n (or more) such hyperplanes; these correspond to n independent rows of A, which together form A_z. But there are some exceptions that one would have to deal with.
• A polytope is *integral* if all its vertices are integral.
• The *integer set* P_I of a polytope $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ is

$$P_I = \{x \in \mathbb{Z}^n : Ax \leq b\},$$

i.e. the points contained in P that have integer coordinates. Clearly we have $P_I \subseteq P$. A polytope P is integral if and only if the convex hull of P_I equals P.

Integer programs: Consider an integer program and its relaxation; their constraints define a polyhedron P; let's assume it is a polytope. This polytope equals the set of solutions of the relaxation, and its integer set P_I is the set of solutions to the integer program. Among the optimal solutions of the linear program there must be vertices.

If the polytope is integral, then the vertices among its optimal solutions are integral, hence they are also optimal solutions of the integer program. So not every optimal solution of the integer program is integral, but if there is an optimal solution there must be an integral one.

Integrality theorems: Below we will prove the integrality theorems that we used for our algorithms in the first few sections, which roughly stated that optimal solutions of a particular integer program were the same as optimal solutions of the relaxation, so we could use the relaxation and what we know about linear programs to solve the combinatorial optimization problem. Now we can say this more precisely: the polytope defined by the relaxation was integral.
The fact that we've been able to find exact polynomial algorithms for most of the problems we've seen so far has a lot to do with their polytopes being integral. After all, we know that there are polynomial algorithms to solve linear programs, so in a roundabout way this gives a polynomial algorithm for any integer program whose polytope is integral. Unfortunately, for many more complicated optimization problems this is not the case, and we will soon see examples of that.

6.2 Total Unimodularity

For some (but not all) optimization problems, we can tell directly from the matrix of constraints in its integer program whether its polytope is integral. In the next section we will use this to prove our integrality theorems.

A matrix is called *totally unimodular* if each square submatrix has determinant $0, 1$ or -1. (a square matrix A is *unimodular* if $|\det(A)| = 1$, but we won't use that here).

Theorem 6.2.1. *Let $P = \{x : Ax \leq b\}$ be a polytope. If A is totally unimodular and b is integral, then P is integral.*
The same is true for the polytopes $\{x : Ax \leq b, x \geq 0\}$, $\{x : Ax = b, x \geq 0\}$, and any variation of these with $Ax \geq b$ instead of $Ax \leq b$.

Proof. We'll only prove the first case here. The other cases can be shown to follow from the first. For instance, $\{x : Ax \leq b, x \geq 0\}$ is of the first form if one writes it as

$$\begin{bmatrix} A \\ -I \end{bmatrix} x \leq \begin{bmatrix} b \\ 0 \end{bmatrix}.$$

Then one just has to show that this larger matrix is also totally unimodular.
So let z be a vertex of $\{x : Ax \leq b\}$, and let A be of size $m \times n$. By the Lemma in the introduction, there is a nonsingular submatrix A_z of A and a subvector b_z of b such that z is the unique point satisfying $A_z z = b_z$. Hence $z = A_z^{-1} b_z$.
As A is totally unimodular, the same is true for A_z, and since it is square and nonsingular we have $\det(A_z) = \pm 1$. By Cramer's rule, each entry of A_z^{-1} is the determinant of a submatrix of A_z divided by $\det(A_z)$, so A_z^{-1} is an integer matrix. Since b_z is assumed integral, it follows that the vertex z is integral. □

Given an undirected graph $G = (V, E)$, we define its *incidence matrix* to be the $|V| \times |E|$ matrix A with rows corresponding to vertices, and columns corresponding to edges, and entries

$$A_{ve} = \begin{cases} 1 & \text{if } v \in e, \\ 0 & \text{if } v \notin e. \end{cases}$$

Theorem 6.2.2. *If A is the incidence matrix of a graph G, then A is totally unimodular if and only if G is bipartite.*

Proof. Suppose G is not bipartite. This implies that it has an odd cycle, and one can check that the determinant of the incidence matrix of an odd cycle is 2. This is a submatrix of A, so A is not totally unimodular.
Suppose G is bipartite. Let B be a square submatrix of A. We show by induction on the size of B that its determinant equals $0, 1$ or -1. The base case, when B has size 1×1, is obvious. We distinguish three cases (which cover all possibilities, since an incidence matrix has at most two 1's in a column).
- B has a column with only 0's: Then $\det(B) = 0$.
- B has a column with exactly one 1: Then we can write, after permuting rows and columns:

$$B = \begin{pmatrix} 1 & \cdots \\ \overline{0} & B' \end{pmatrix},$$

where $\overline{0}$ denotes a column of 0's, and B' is a smaller matrix. Then $\det(B) = \det(B')$, and by induction $\det(B') \in \{0, \pm 1\}$.
- Every column of B has exactly two 1's: Since G is bipartite, we can write (after permuting rows)

$$B = \begin{pmatrix} B' \\ B'' \end{pmatrix},$$

in such a way that each column of B' contains exactly one 1 and each column of B' contains exactly one 1. If the partite sets of G are U and V, this is done by letting B' correspond to the vertices of U and B'' to the vertices of V. Then adding up the rows in B' gives the vector with all 1's, and so does adding up the rows in B''. But that means the rows are linearly dependent, so $\det(B) = 0$. □

Given a directed graph $G = (V, E)$, we define its *incidence matrix* to be the $|V| \times |E|$ matrix A with rows corresponding to vertices, and columns corresponding to edges, and entries

$$A_{ve} = \begin{cases} 1 & \text{if } v = t(e), \\ -1 & \text{if } v = h(e), \\ 0 & \text{else.} \end{cases}$$

(Recall that if $e = uv$, then $u = t(e)$ is the tail and $v = h(e)$ is the head.)

Theorem 6.2.3. *If A is the incidence matrix of a directed graph, then A is totally unimodular.*

Proof. Almost identical to that for bipartite graphs, with some minuses thrown in. □

6.3 Integrality Theorems

Bipartite matchings
First we'll look at the linear program that we used for bipartite weighted matchings. Note that it is the relaxation of the integer program which exactly described matchings.

LP for bipartite max weight matchings

$$\text{maximize} \sum_{e \in E(G)} w_e x_e \quad \text{with } x \geq 0,$$

$$\sum_{e \ni v} x_e \leq 1 \quad \text{for } v \in V.$$

The polyhedron for this linear program is really a polytope, because $0 \leq x_e \leq 1$ for all e.

Theorem 6.3.1. *If G is bipartite, then the polytope defined by the linear program above is integral.*

Proof. The constraints are exactly $Ax \leq \overline{1}$, where A is the incidence matrix of G. Then A is totally unimodular because G is bipartite, and the vector $\overline{1}$ is obviously integral, so the polytope is integral. □

Shortest Paths

Next we'll look at the relaxation of the integer program for shortest paths. Note that the integer program did not exactly describe paths, because non-optimal solutions could involve cycles, but its minima were exactly the shortest paths.

LP for shortest ab-path

$$\text{minimize } \sum w_e x_e \quad \text{with } 0 \leq x \leq 1,$$

$$\sum_{e \in \delta^{\text{in}}(v)} x_e - \sum_{e \in \delta^{\text{out}}(v)} x_e = 0 \quad \text{for } v \in V \backslash \{a, b\},$$

$$\sum_{e \in \delta^{\text{in}}(b)} x_e = 1, \qquad - \sum_{e \in \delta^{\text{out}}(a)} x_e = -1.$$

We're cheating a little bit, because in the lecture on shortest path, we did not include the constraint $x \leq 1$. This made it easier to dualize, but without it the associated polyhedron is not a polytope (on those extraneous cycles the x_e could be arbitrarily large). It's not hard to prove that this doesn't make a difference, as minimal solutions will satisfy $x \leq 1$ anyway.

Theorem 6.3.2. *The polytope defined by the linear program for shortest paths is integral.*

Proof. The matrix is almost, but not quite, the incidence matrix of the directed graph (actually, with all entries multiplied by -1, but that's no problem). Indeed, most edges occur twice in the constraints, once with a $+$ in the constraint for its head, and once with a $-$ in the constraint for its tail. These correspond to the $+1$ and -1 in the column of this edge.
The only exceptions are edges that have b as a tail or a as a head, which occur only once, and have only one $+1$ or -1 in their column. If we remove the corresponding columns from the matrix, we have a submatrix of the incidence matrix of the graph. So this submatrix is also totally unimodular. Adding the removed columns back in one by one, the total unimodularity is preserved, because these columns have one 1 and the rest 0's.
So the matrix in the LP for shortest paths is totally unimodular, and the vector b is clearly integral, therefore the corresponding polytope is integral. □

Flows

Note that in the lecture on flows, we didn't actually need integrality, because flows were defined as real functions on the edges, so a maximum flow was by definition a solution of a linear program, not of an integer program.
It is still true that when the capacities are integral, then the vertices of the corresponding polytope are integral, and there is a maximum flow that is integral. But this doesn't directly follow from total unimodularity, because the capacities are in the constraint matrix, and they need not be in $\{0, \pm 1\}$.
One can get around this, but it's actually easier to just prove integrality using the augmenting path algorithm. So we won't do it here.

Trees
We only briefly mentioned the integer program for trees, and we didn't need it at all. To prove integrality for it, total unimodularity would not be not much help. But trees are a special case of matroids, which we will see soon, and probably we will then see, and prove in a different way, the integrality theorem for matroid polytopes.

6.4 Matching Polytopes

Matching polytopes: Given a graph G, define its *matching polytope* $\mathcal{M}(G)$ to be the convex hull of the incidence vectors of matchings in G (vectors $(x_e)_{e\in E}$ such that $x_e = 1$ for matching-edges and $x_e = 0$ for non-matching-edges).
Similarly, define its *perfect matching polytope* $\mathcal{PM}(G)$ to be the convex hull of the incidence vectors of perfect matchings in G.
So unlike the previous polytopes, these are defined as convex hulls of finite sets of points, and the goal is to find a set of inequalities that exactly describe them. For bipartite graphs, we have already found those, but for non-bipartite graphs we haven't.

Bipartite graphs: As we just saw, when G is bipartite, then the polytope

$$P = \{x : Ax \le 1, x \ge 0\}$$

is integral, where A is the incidence matrix of G. This means that P equals the convex hull of P_I, which equals $\mathcal{M}(G)$, so $P = \mathcal{M}(G)$.
Similarly, we could show that if G is bipartite then $\mathcal{PM}(G) = \{x : Ax = 1, x \ge 0\}$.

Non-bipartite graphs: For non-bipartite graphs, $P \ne \mathcal{M}(G)$ and $P' \ne \mathcal{PM}(G)$. Indeed, if a graph is not bipartite, then it contains an odd cycle. On an odd cycle there cannot be a perfect matching, but there can be a "fractional" perfect matching: If $x_e = 1/2$ for each e, then $\sum_{e\ni v} x_e = 1$ for each v. Also, the value $\sum x_e$ of this solution will be $|C|/2$, which is larger than the value of the maximum cardinality matching, which is $(|C| - 1)/2$.
So although the obvious integer program still describes all matchings for nonbipartite graphs, its relaxation is inadequate. However, there is a *different* integer program for matchings, which has more constraints, whose relaxation does do the job. We will show this for perfect matchings, and we'll state the relaxation right away.

LP for general max weight perfect matchings

$$\begin{aligned} \text{maximize} \quad & \sum_{e\in E(G)} w_e x_e \quad \text{with } x \ge 0, \\ & \sum_{e\in\delta(v)} x_e = 1 \quad \text{for } v \in V, \\ & \sum_{e\in\delta(S)} x_e \ge 1 \quad \text{for } S \subset V(G),\ |S| \text{ odd}. \end{aligned}$$

Note that any perfect matching does indeed satisfy the added constraints: For any odd set S of vertices, at least one vertex from S must be matched to a vertex outside S, which means there is a matching edge in $\delta(S)$.
Also note that these constraints are broken by the fractional example given above: If $x_e = 1/2$ for all e in an odd cycle C, then $\sum_{e\in\delta(C)} x_e = 0$.
Geometrically, here's what's happening: The first polytope may have non-integral vertices which have entries $1/2$ on odd cycles (we can prove that those are the only kind), and the extra

constraints exactly cut those off, without cutting off any integral vertices, and without creating any new fractional vertices. That's what the following theorem states.
Note that although this theorem shows that one can use linear programming to find perfect matchings, this actually does not directly give a polynomial algorithm, because the number of constraints in the program is exponential.
We will only sketch proof of this theorem, because it would be quite long. But this outline does show that the proof pulls together the observation that we've made so far, and also that there is a vague connection with shrinking blossoms.

Theorem 6.4.1. *For any graph G, the polytope defined by the linear program above is integral, and equals the perfect matching polytope $\mathcal{PM}(G)$.*

Proof. (sketch) Let $\widetilde{x}$ be a vertex. Then by Lemma 6.1, there is a subset of the constraints for which $\widetilde{x}$ is the unique solution with equality in each of these constraints. More precisely, there is $\widetilde{E} \subset E$, and a set $\mathcal{W}$ of odd $S \subset V$ with $|S| \geq 3$, such that $\widetilde{x}$ is the unique solution to

$$x_e = 0 \ \forall e \in \widetilde{E}, \quad \sum_{e \in \delta(S)} x_e = 1 \ \forall S \in \mathcal{W}, \quad \sum_{e \in \delta(v)} = 1 \ \forall v \in V.$$

First suppose that $\mathcal{W} \neq \emptyset$, and pick any $S \in \mathcal{W}$. Then we can shrink S to a vertex to get a graph G_1, and we can shrink $V \backslash S$ to a vertex to get a graph G_2. One can check (but we won't) that $\widetilde{x}$ gives two vertices $\widetilde{x}_1$ and $\widetilde{x}_2$ of the corresponding polytopes for G_1 and G_2. Since these are smaller, by induction $\widetilde{x}_1$ and $\widetilde{x}_2$ are perfect matchings (actually, convex combinations), and one can check that these can be combined into a maximum weight perfect matching in G which corresponds to $\widetilde{x}$. In particular, $\widetilde{x}$ is integral.
We still have to consider the case $\mathcal{W} = \emptyset$, which also functions as a base case for the induction. In this case, $\widetilde{x}$ is also a vertex of the polytope $\{x : Ax \leq 1, x \geq 0\}$. We claim, but will not prove, that such a vertex x is almost integral, in the sense that $x_e \in \{0, 1, \frac{1}{2}\}$, and the edges e for which $x_e = \frac{1}{2}$ form odd cycles. In other words, the counterexample that we saw above, an odd cycle, is basically the only kind of counterexample.
But $\widetilde{x}$ cannot have such an odd cycle C with weights $1/2$, since it would violate the constraint for $S = C$. Hence all $\widetilde{x}_e \in \{0, 1\}$ and $\widetilde{x}$ corresponds to a perfect matching. □

Chapter 7

Matroids

7.1 Definition • 7.2 Examples • 7.3 Greedy algorithm

7.1 Definitions

• **Matroids:** A *matroid* is a pair $(X, \mathcal{I})$, with X a finite set and $\mathcal{I}$ a nonempty collection of subsets of X, that satisfies the axioms

(M1): If $Y \in \mathcal{I}$ and $Z \subset Y$ then $Z \in \mathcal{I}$,

(M2): If $Y, Z \in \mathcal{I}$ and $|Y| < |Z|$ then $Y \cup \{x\} \in \mathcal{I}$ for some $x \in Z \backslash Y$.

Without these axioms such a pair is called a *set system*, and a set system that satisfies (M1) is called an *independence system*. In an independence system or matroid, the sets $Y \in \mathcal{I}$ are called *independent*, the sets $Y \notin \mathcal{I}$ *dependent*. The axiom (M1) is called the *hereditary axiom*, (M2) is called the *exchange axiom*.

• **Bases:** Given an independence system or matroid, a maximal independent set B is called a *basis* of X. For any $Y \subset X$, a subset $B \subset Y$ that is maximal independent is a *basis* of Y.

Lemma 7.1.1. *An independence system $(X, \mathcal{I})$ satisfies* M2 *if and only if it satisfies*

(M2′)*: For any $Y \subset X$, all bases of Y have the same cardinality.*

Proof. Suppose it satisfies M2. If Z_1 and Z_2 are bases of some $Y \subset X$ with different cardinality, say $|Z_1| < |Z_2|$, then by M2 there is an $x \in X$ such that $Z_1 \cup \{x\} \in \mathcal{I}$, contradicting the fact that Z_1 is a basis. This proves M2′.
Suppose the system satisfies M2′, and $Y, Z \in \mathcal{I}$ with $|Y| < |Z|$. Consider $Y \cup Z \subset X$. Then Z is contained in some basis B of $Y \cup Z$, and $|B| \geq |Z| > |Y|$. But Y must also be contained in some basis B' of $Y \cup Z$, and by M2′ we must have $|B'| = |B| > |Y|$. So there must be some $x \in B' \backslash Y$, and then $Y \cup \{x\} \in \mathcal{I}$ by M1, since it is a subset of $B' \in \mathcal{I}$. That proves M2. □

• **Rank:** The common cardinality of the bases of X in a matroid $M = (X, \mathcal{I})$ is called the *rank* of the matroid. Similarly, for any $Y \subset X$ the common cardinality of the bases of Y is called the *rank* of Y; we will write $r_M(Y)$ for it, and call r_M the *rank function*.

7.2 Examples

• **Explicit example:** Let $X = \{1, 2, 3, 4\}$ and

$$\mathcal{I} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}\}.$$

This is clearly an independence system, and also a matroid, since one can check that M2 holds for each pair $Y, Z \in \mathcal{I}$. For instance, if $Y = \{3\}$ and $Z = \{1, 2\}$, then $|Y| < |Z|$, so there should be $x \in Z \backslash Y = \{1, 2\}$ such that $Y \cup \{x\} \in \mathcal{I}$, and indeed $x = 2$ will do this.
The bases of X are $\{1, 2\}$ and $\{2, 3\}$, and they do indeed have the same cardinality, as they should by the lemma. The rank of X is 2. The rank of the subset $Y = \{1, 3, 4\}$ is $r_M(Y) = 1$.
Suppose we add $\{4\}$ to $\mathcal{I}$. Then $(X, \mathcal{I})$ is still an independence system, but no longer a matroid. Indeed, $\{4\}$ will become a basis, hence M2′ is not satisfied. Alternatively, taking $Y = \{4\}$ and $Z = \{1, 2\}$ shows that M2 is violated.

• **Uniform matroids:** Let X be any set, and for an integer k take

$$\mathcal{I} = \{Y \subset X : |Y| \leq k\}.$$

Then $M = (X, \mathcal{I})$ is a matroid, called a *uniform matroid*. If $|X| = n$, it is denoted by U_{kn}. Note that in U_{nn} all subsets of X are independent.
The bases of X are the subsets Y with $|Y| = k$. The rank of $Y \subset X$ is $r_M(Y) = \min(|Y|, k)$.

• **Linear matroids:** Let $A \in \mathbb{R}^{m \times n}$ be a matrix, and write A_i for the ith column of A. Let $X = \{1, 2, \ldots, n\}$, and

$$\mathcal{I} = \{Y \subset X : \text{ the } A_i \text{ with } i \in Y \text{ are linearly independent}\}.$$

This is a matroid: Clearly a subset of a linearly independent set is linearly independent, and it is a basic fact from linear algebra that given two linearly independent sets Y, Z of vectors, with $|Y| < |Z|$, then there is an $x \in Z$ that is linearly independent from Y. Or to use M2′: Given any $Y \subset X$, denote the corresponding submatrix by A_Y; then the cardinality of a basis of Y equals the rank (in the linear algebra sense) of A_Y, so does not depend on the basis.
The (matroid) bases are the (linear algebra) bases of the column space of A (the linear space spanned by its column vectors). The (matroid) rank of a subset $Y \subset X$ is the (linear algebra) rank of the corresponding submatrix A_Y. The word "matroid" was in fact meant to mean "matrix-like", and matroids were introduced as a generalization of linear independence.
Note that the definition depends on the field of the linear space (here $\mathbb{R}$), since linear dependence does. For instance, the vectors $(2, 1)^T$ and $(1, 2)^T$ are independent over $\mathbb{R}$, but they are dependent over $\mathbb{F}_3$, the field with 3 elements, since their sum is 0 modulo 3.

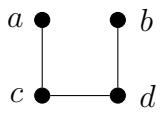
• **Forest matroids:** Let $G = (V, E)$ be a graph. Let $X = E$ and

$$\mathcal{I} = \{F \subset E : F \text{ is a forest}\}.$$

Then $M = (X, \mathcal{I})$ is a matroid. A subgraph of a forest is of course a forest, proving M1. To prove M2, let F_1 and F_2 be forests with $|F_1| < |F_2|$. Then F_2 has fewer components than F_1, and within any component of F_1, F_2 can have at most as many edges as F_1 (namely the number of vertices in the component minus 1). Hence there must be an edge x in F_2 that connects two components of F_1, which means that $F_1 \cup \{x\}$ is also a forest.
Alternatively, we can use M2′. Given a set of edges $Y \subset X$, it defines a subgraph G_Y. If c is

the number of components of G_Y, then the maximum cardinality of a forest in G_Y is $|V| - c$ (this is a basic exercise in graph theory). Hence all bases (maximal forests) of Y have the same cardinality.
So the bases of $Y \subset X$ are the maximal forests in G_Y, and the rank of $Y \subset X$ is $r_M(Y) = |V| - c(Y)$, where $c(Y)$ is the number of components of G_Y.

• **Matchings?** The matchings in a graph do *not* form a matroid, if considered as subsets of the edge set. Indeed, take the graph



The set of matchings would be

$$\{\emptyset, \{ac\}, \{cd\}, \{bd\}, \{ac, bd\}\},$$

which would have $\{ac, bd\}$ and $\{cd\}$ as bases of different cardinalities.

7.3 Greedy Algorithm

We can generalize the algorithm for finding maximum weight forests to one for matroids. To be precise, the problem to solve is:

Maximum weight independent set problem: Given a matroid $M = (X, \mathcal{I})$, with weight function $w : X \to \mathbb{R}$, find an independent set $Y \in \mathcal{I}$ with maximum weight $w(Y) = \sum_{y \in Y} w(y)$.

Note that we allow negative weights, but they will never be included in a maximum weight independent set, because of M1. The algorithm will stop when it reaches elements with weight ≤ 0; it could also just remove them from X before starting.

Greedy algorithm for maximum weight independent sets

1. Set $Y = \emptyset$ and $S = X$;
2. Find $x \in S$ with maximum $w(x)$; if $S = \emptyset$ or $w(x) \leq 0$, go to 4;
3. If $Y \cup \{x\} \in \mathcal{I}$, set $Y := Y \cup \{x\}$;

   remove x from S; go to 2;
4. Return Y.

This algorithm is basically polynomial, but there is a subtlety: One has to assume that checking if $Z \in \mathcal{I}$ can be done in polynomial time. This is not always true, because $\mathcal{I}$ can be exponentially large, so if it is just given as a list of subsets, that means it will take exponential time to check independence.
Fortunately, for the examples that we'll see, $\mathcal{I}$ comes from some natural structure, like a graph or a vector space, for which independence can be checked in polynomial time.
That the greedy algorithm returns a maximum weight independent set follows in a similar way as for forests. What is more interesting is that matroids are exactly those set systems for which the greedy algorithm works for all weights.

Theorem 7.3.1. *A set system $(X, \mathcal{I})$ is a matroid if and only if the greedy algorithm returns a $Y \in \mathcal{I}$ of maximum weight $w(Y)$, for each weight function $w : X \to \mathbb{R}$.*

Proof. First we show that the algorithm works for matroids. Call a $Y \in \mathcal{I}$ *good* if it is contained in a maximum-weight independent set. We will show that if Y is good, and the algorithm adds x to Y (so $x \in S$ with maximum $w(x)$ and $Y \cup \{x\} \in \mathcal{I}$), then $Y \cup \{x\}$ is also good. It then follows that the algorithm terminates with a good set Y that is itself a maximum-weight independent set.
Note that $w(x) > 0$, since otherwise x would never be added. Let Z be a maximum-weight independent set with $Y \subset Z$. If $x \in Z$ then $Y \cup \{x\}$ is clearly good. If $x \notin Z$, consider $Z \cup \{x\}$, which must be dependent, since its weight is greater than that of Z. Because $Y \cup \{x\} \subset Z \cup \{x\}$ is independent, we can repeatedly apply M2 to add elements of Z until we obtain an independent set Z' that contains $Y \cup \{x\}$ and that has the same size as Z. This means that there exists $x' \in Z \backslash Y$ such that

$$Z' = (Z \cup \{x\}) \backslash \{x'\} \in \mathcal{I}.$$

At the step in the algorithm where it adds x to Y, x' must still be in S, because if the algorithm had considered x' then it would have added it (since for any $Y' \subset Y$, $Y' \cup \{x'\} \subset Z$ is independent), whereas $x' \notin Y$. Thus the algorithm must be choosing x over x', so $w(x) \geq w(x')$. This implies that $w(Z') \geq w(Z)$ (and in fact $w(Z') = w(Z)$), so Z' is a maximum-weight independent set containing $Y \cup \{x\}$, so $Y \cup \{x\}$ is good.

Next we show that if a set system $(X, \mathcal{I})$ does not satisfy M1, then there will be weights w for which the greedy algorithm does not return a maximum weight element of $\mathcal{I}$.
There must be an $A \notin \mathcal{I}$ and $b \notin A$ with $A \cup \{b\} \in \mathcal{I}$. Define w by

$$w(x) = \begin{cases} 2 & \text{if } x \in A, \\ 1 & \text{if } x = b, \\ -1 & \text{else.} \end{cases}$$

Then $A \cup \{x\}$ is the unique maximum weight element of $\mathcal{I}$. But the greedy algorithm can never get to $A \cup \{x\}$, since before it gets to x, it must have considered all $a \in A$, and it must have rejected at least one of them because $A \notin \mathcal{I}$.

Finally we show that if a set system $(X, \mathcal{I})$ satisfies M1 but not M2, then there will be weights w for which the greedy algorithm does not return a maximum weight element of $\mathcal{I}$.
There must be $Y, Z \in \mathcal{I}$ with $|Y| < |Z|$ but $Y \cup \{x\} \notin \mathcal{I}$ for any $x \in Z \backslash Y$. Define w by

$$w(x) = \begin{cases} 1 + \epsilon & \text{if } x \in Y \ \ (\text{with } \epsilon > 0), \\ 1 & \text{if } x \in Z \backslash Y, \\ 0 & \text{else.} \end{cases}$$

After the first $|Y|$ iterations, the greedy algorithm will reach Y, since all its subsets are in $\mathcal{I}$ by M1. But then no more nonzero elements can be picked, since $Y \cup \{x\} \notin \mathcal{I}$ for $x \in Z \backslash Y$. So the maximum weight found by the greedy algorithm is $|Y| \cdot (1 + \epsilon)$.
But $Z \in \mathcal{I}$ and

$$w(Z) \geq |Z \cap Y| \cdot (1 + \epsilon) + |Z \backslash Y| \cdot 1 \geq |Z| \geq |Y| + 1.$$

So if we take $\epsilon > 0$ small enough so that $|Y| \cdot (1 + \epsilon) < |Y| + 1$, then the greedy algorithm does not find the maximum weight set in $\mathcal{I}$. □

7.4 Problems

1. Show that any forest matroid is also a linear matroid. Also show that the uniform matroid $U_{2,4}$ is a linear matroid, but not a forest matroid.

2. Let A be the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

   Let M be the corresponding linear matroid over $\mathbb{F}_2$, the field with 2 elements. Show that M is not a forest matroid, and not a linear matroid over $\mathbb{R}$. (M is called the *Fano matroid*.)

3. In a matroid $M = (X, \mathcal{I})$, a *circuit* is a minimal dependent set, i.e. a C such that $C \notin \mathcal{I}$, but if $D \subsetneq C$ then $D \in \mathcal{I}$.
   Circuits are the generalization of cycles in graphs. The following two facts are the generalizations of standard facts about cycles.

   a) Prove that if $C_1 \neq C_2$ are two circuits, and $x \in C_1 \cap C_2$, then there is a circuit $C_3 \subset (C_1 \cup C_2) \backslash \{x\}$.

   b) Prove that if $Y \in \mathcal{I}$, but $Y \cup \{x\} \notin \mathcal{I}$, then $Y \cup \{x\}$ contains exactly one circuit.

4. Describe a *greedy removal algorithm* for maximum weight indepent sets in a matroid, which starts with X and greedily removes elements. Prove that it works.

5. Let $X_1, \ldots, X_m$ be a partition of X. Define

$$\mathcal{I} = \{Y \subset X : \forall i \;\; |Y \cap X_i| \leq 1\}.$$

   Prove that $(X, \mathcal{I})$ is a matroid. Such a matroid is called a *partition matroid*.
   Let $G = (A \cup B, E)$ be a bipartite graph. Show that the set of matchings of G is an intersection of two partition matroids with $X = E$, i.e. $\{\text{matchings}\} = \mathcal{I}_1 \cap \mathcal{I}_2$.

(Solutions on page 84.)

Chapter 8

Matroid Intersection

8.1 Definition and examples • 8.2 Matroid Intersection Algorithm

8.1 Definitions

• Given two matroids $M_1 = (X, \mathcal{I}_1)$ and $M_2 = (X, \mathcal{I}_2)$ on the same set X, their *intersection* is

$$M_1 \cap M_2 = (X, \mathcal{I}_1 \cap \mathcal{I}_2).$$

Note that the intersection of two matroids is an independence system, but not typically a matroid. We can similarly define the intersection of any finite number of matroids.

Below are some examples of graph-theoretical objects that can be viewed as elements of a matroid intersection. Keep in mind that since these will (typically) not be matroids, we know that the greedy algorithm cannot work for them (in general). In the next section, we'll see an algorithm that works for any intersection of two matroids.
We'll use some shorter notation. The forest matroid of an undirected graph G will be written as $F(G)$; the partition matroid of a set X, corresponding to a partition $\{X_1, \ldots, X_m\}$ of X, will be written as $P(\{X_i\})$ (see Problem Set 7), so

$$P(\{X_i\}) = \{Y \subset X : |Y \cap X_i| \leq 1 \ \forall i\}.$$

Simultaneous forests
Given two graphs G_1 and G_2, and a bijection $\varphi : E(G_1) \to E(G_2)$ between their edges, we could look for a (maximum) forest F_1 in G_1 such that $\varphi(F_1)$ is a forest in G_2. This is the same as looking for a maximum forest in the matroid intersection $M_1 \cap M_2$, where

$$M_1 = F(G_1), \quad M_2 = F(G_2).$$

Here we should identify E_1 and E_2 via φ, and call both X.

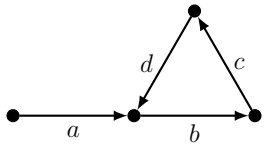
Bipartite matchings
We saw in the last lecture that matchings (bipartite or not) do not form a matroid, at least not as sets of edges. But in the problem set we saw that the edge sets of bipartite matchings do form an intersection of the following two partition matroids (let A and B be the partite sets):

$$M_1 = P(\{\delta(a)\}_{a \in A}), \quad M_2 = P(\{\delta(b)\}_{b \in B})$$

Directed forests

Given a directed graph D, let G_D be the undirected graph with the same edges, so $G_D = \{\{a, b\} : ab \in E(D)\}$. Recall that a directed forest in D is an $F \subset E(D)$ that forms a forest in G_D and satisfies $|F \cap \delta^{\text{in}}(v)| \leq 1$ for all $v \in V(D)$.

In the lecture on trees we saw that maximum directed forests cannot always be found by a greedy approach, which indirectly implies that they do not form a matroid. Actually, we saw this for spanning directed trees, but we also showed that these problems are equivalent. More directly, in the directed graph



the directed forests are not a matroid, since $\{a, b, c\}$ and $\{c, d\}$ are bases of different size.

But directed forests are the intersection of the two matroids

$$M_1 = F(G_D), \quad M_2 = P(\{\delta^{\text{in}}(v)\}).$$

This should be clear from the definition, since the two conditions in the definition of directed forest (being a forest, single-entry) correspond exactly to these two matroids.

Colorful forests

The two previous examples were objects that we have already seen before, and in fact we already have polynomial algorithms for them. But many more less standard combinations are possible.

For instance, given a graph G and a partition $E = \cup E_i$ of its edges, take

$$M_1 = F(G), \quad M_2 = P(\{E_i\}).$$

Then $M_1 \cap M_2$ can be interpreted as follows. The partition can be seen as a *coloring* of the edges, giving all the edges in E_i the same color. Then a maximum edge set in $M_1 \cap M_2$ is a *colorful* forest: it uses every color at most once.

Intersections of three or more matroids

Intersecting more than two matroids can also give meaningful objects, like colorful directed forests or colorful matchings. More significantly, you'll see in the problem set that subsets of Hamilton paths form an intersection of 3 matroids. Since finding Hamilton paths is NP-hard (as we'll see next time), this right away shows that finding a maximum independent set in an intersection of 3 (or more) matroids is NP-hard, so there is no polynomial algorithm for it unless $P = NP$.

So to summarize: matroids can be solved easily by the greedy algorithm, intersections of 2 matroids are not as easy but can be solved by the polynomial algorithm in the next section, and for intersections of 3 or more matroids there is no general polynomial algorithm. Below we will show that every independence system, which includes many combinatorial optimization problems, is an intersection of matroids.

However, for some intersections of 3 or more matroids there may be a polynomial algorithm. For instance, one can intersect one of the previous intersection of 2 matroids with another matroid in a "trivial" way, like with the matroid of all subsets. Or less trivially, one can intersect with a uniform matroid of sets of size $\leq k$, which gives a problem that can be solved just as easily.

More interestingly, we have a polynomial algorithm for shortest paths, and the set of subsets of shortest paths forms an independence system. By the theorem below, this is an intersection of matroids, but it does not seem to be an intersection of 2 matroids.

Theorem 8.1.1. *Any independence system $(X, \mathcal{I})$ with finite X is an intersection of a finite number of matroids, i.e. there are matroids $M_i = (X, \mathcal{I}_i)$ with*

$$\mathcal{I} = \bigcap \mathcal{I}_i.$$

Proof. Recall that a *circuit* in an independence system is a minimal dependent set (in a forest matroid, a circuit is the same as a cycle in the graph). If C is a circuit of $(X, \mathcal{I})$, then $(X, \mathcal{I}_C)$ with

$$\mathcal{I}_C = \{Y \subset X : C \not\subset Y\}$$

is a matroid. Indeed, it clearly satisfies M1, and it satisfies M2′ since the bases are exactly the sets $X \backslash \{c\}$ for some $c \in C$, so they all have the same size.
There are finitely many circuits, and $\mathcal{I}$ is the intersection of the $\mathcal{I}_C$ for all circuits, because a set is independent if and only if it does not contain a dependent set, which is equivalent to not containing a circuit. □

8.2 Matroid Intersection Algorithm

Here we will use the notation $S + T = S \cup T$, $S - T = S \backslash T$, and also $S + x = S \cup \{x\}$ and $S - x = S \backslash \{x\}$.

The algorithm for intersections of two matroids will again use augmenting paths, though not in a way that directly derives from the augmenting paths for matchings.
Let $M_1 = (X, \mathcal{I}_1)$ and $M_2 = (X, \mathcal{I}_2)$. Given an independent set $I \in \mathcal{I}_1 \cap \mathcal{I}_2$, we define a directed graph $D_I = (X, E(D_I))$ which is bipartite with partite sets I and $X - I$. The edges are defined by, for each $y \in I$ and $z \in X - I$, putting

$$yz \in E(D_I) \text{ if } I - y + z \in \mathcal{I}_1,$$
$$zy \in E(D_I) \text{ if } I - y + z \in \mathcal{I}_2.$$

In words, we put an edge from y to z if exchanging y in I for z gives an independent set in $\mathcal{I}_1$, and an edge from z to y if it gives an independent set in $\mathcal{I}_2$. Note that it is possible that there is no edge between y and z, or an edge in both directions.
What we hope for is a set $Y \subset I$ and $Z \subset X - I$ such that $|Z| = |Y| + 1$ and $I - Y + Z \in \mathcal{I}_1 \cap \mathcal{I}_2$, so we can augment I to $I - Y + Z$. This suggests taking a directed path in D_I from $X - I$ to $X - I$, but not any such path would work. It turns out that one should take a shortest path from $X_1 \subset X - I$ to $X_2 \subset X - I$, where

$$X_1 = \{z \in X - I : I + z \in \mathcal{I}_1\},$$

$$X_2 = \{z \in X - I : I + z \in \mathcal{I}_2\}.$$

If $Q = z_0 y_1 z_1 y_2 z_2 \cdots y_m z_m$ is such a path, then with $Y = \{y_i\}$ and $Z = \{z_i\}$ it is true that $I \Delta V(Q) = I - Y + Z \in \mathcal{I}_1 \cap \mathcal{I}_2$. This is the augmentation step in the algorithm.

Algorithm for max independent set in matroid intersection $M_1 \cap M_2$ on X

1. Set $I = \emptyset$ and $V(D_I) = X$;
2. Greedily add to I any $z \in X - I$ such that $I + z \in \mathcal{I}_1 \cap \mathcal{I}_2$;

3. Set $E(D_I) = \{yz : y \in X, z \in X - I, I - y + z \in \mathcal{I}_1\}$
$\cup \{zy : y \in X, z \in X - I, I - y + z \in \mathcal{I}_2\}$;

   Let $X_1 = \{z \in X - I : I + z \in \mathcal{I}_1\}$, $X_2 = \{z \in X - I : I + z \in \mathcal{I}_2\}$;

4. Find a shortest X_1X_2-path Q in D_I; if none exists, go to 6;

5. Augment using Q: $I := I\Delta V(Q)$; go back to 2;

6. Return I.

The second step is not necessary, since such a z will be found as the shortest path $Q = z$, but when doing the algorithm by hand it is highly recommended, since it saves a lot of work in setting up D_I.

Theorem 8.2.1. *Given two matroids M_1 and M_2 on the same X, the algorithm above returns a maximum common independent set $I \in M_1 \cap M_2$ in polynomial time.*

Proof. (partial) That it terminates and is polynomial should be clear. We will omit the proof that each $I \in \mathcal{I}_1 \cap \mathcal{I}_2$, since this is somewhat long and technical; we will merely justify it at the end of the proof in the two simplest cases.
The only other thing to prove is that the final I is indeed maximum. Given any $J \in \mathcal{I}_1 \cap \mathcal{I}_2$ and $U \subset X$, observe that

$$|J| = |J \cap U| + |J \cap (X - U)| \leq r_{M_1}(U) + r_{M_2}(X - U). \tag{8.1}$$

The inequality holds because $J \cap U$ is independent and contained in U, hence for any basis B_U of U we have $|J \cap U| \leq |B_U| = r_{M_1}(U)$; similarly $|J \cap (X - U)| \leq |B_{X-U}| = r_{M_2}(X - U)$.
For the final I in the algorithm, we show that there is a $U \subset X$ for which we have equality, i.e. $|I| = r_{M_1}(U) + r_{M_2}(X - U)$. This proves that I is maximum.
Consider

$$U = \{z \in X : \exists \text{ a path in } D_I \text{ from } z \text{ to } X_2\}.$$

Then $X_2 \subset U$ and $X_1 \cap U = \emptyset$. We claim that simply $r_{M_1}(U) = |I \cap U|$ and $r_{M_2}(X - U) = |I \cap (X - U)|$, which proves that we have equality in (8.1) for I.
Suppose $|I \cap U| < r_{M_1}(U)$. Then there is an M_1-basis B_U of U with $|I \cap U| < |B_U|$, so by M1 there is an $x \in B_U - I$ such that $(I \cap U) + x \in \mathcal{I}_1$. Since $X_1 \cap U = \emptyset$, we know that $I + x \notin \mathcal{I}_1$, so $|(I \cap U) + x| \leq |I|$.
If $|(I \cap U) + x| = |I|$, then $(I \cap U) + x = I - y + x$ for some $y \in I - U$. If $|(I \cap U) + x| < |I|$, by M1 there must be an $x' \in I - U$ such that $(I \cap U) + x + x' \in \mathcal{I}_1$, and we can repeat this until we also have a set of the form $I - y + x \in \mathcal{I}_1$, for some $y \in I - U$. Either way we have a $y \in I - U$ such that $I - y + x \in \mathcal{I}_1$, which implies $yx \in E(D_I)$ by definition of D_I. But this would imply that there is a path from y via x to X_2, so $y \in U$, a contradiction.
The proof that $r_{M_2}(X - U) = |I \cap (X - U)|$ is very similar.

As we said, we omit the proof that each I during the algorithm is in $\mathcal{I}_1 \cap \mathcal{I}_2$, but we will show it for the two simplest cases.
If $z \in X_1 \cap X_2$, then $Q = z$ is the shortest path, and indeed $I\Delta V(Q) = I + z \in \mathcal{I}_1 \cap \mathcal{I}_2$. This is basically a greedy step.
If $Q = z_0y_1z_1$, then we know that $I + z_0 \in \mathcal{I}_1$ because $z_0 \in X_1$, $I + z_1 \in \mathcal{I}_2$ because $z_1 \in X_2$, $I - y_1 + z_0 \in \mathcal{I}_2$ because $z_0y_1 \in E(D_I)$, and finally $I - y_1 + z_1 \in \mathcal{I}_1$ because $y_1z_1 \in E(D_I)$. Since $|I + z_0| > |I - y_1 + z_1|$, we have by M1 that there is a $z \in I + z_0$ that is not in $I - y_1 + z_1$

such that $(I - y_1 + z_1) + z \in \mathcal{I}_1$. This z would have to be either y_1 or z_0, but if it was y_1, then we'd have $I + z_1 = (I - y_1 + z_1) + y_1 \in \mathcal{I}_1$, which means $z_1 \in X_1 \cap X_2$ and $Q = z_1$ would have been a shorter path. So $z = z_0$, and we have $I\Delta V(Q) = (I - y_1 + z_1) + z_0 \in \mathcal{I}_1$. An identical argument gives $I\Delta V(Q) \in \mathcal{I}_2$. □

The inequality that we used in the proof in fact gives a min-max formula that is interesting in its own right, since one can use it to prove that a particular common independent set is maximal. Had we worked out the integer program for matroid intersection, and proven integrality, then we would have seen that this formula follows from our good old friend the LP duality theorem.

Theorem 8.2.2 (Matroid Intersection Theorem)**.**

$$\max_{J \in \mathcal{I}_1 \cap \mathcal{I}_2} |J| = \min_{U \subset X} \left(r_{M_1}(U) + r_{M_2}(X - U) \right).$$

Proof. We saw $\leq$ in the proof above, and the algorithm gives us an I and a U such that equality holds. □

8.3 Problems

1. Show that the problem of finding a Hamilton path from s to t in a given directed graph D can be solved using an intersection of 3 matroids.

2. Given an undirected graph $G = (V, E)$, an *orientation* is a directed graph $D = (V, E')$ with a bijection $\varphi : E' \to E$ such that $\varphi(ab) = \{a, b\}$. In other words, each edge $\{a, b\} \in E$ is given a direction, either ab or ba.
   Given $k : V \to \mathbb{N}$, show that the problem of finding an orientation such that

   $$\delta^{\text{in}}(v) = k(v)$$

   for each $v \in V$, or showing that none exists, can be solved using the matroid intersection algorithm.

3. Use the matroid intersection algorithm to show that there is no simultaneous spanning tree in the following two graphs (i.e., there is no $T \subset \{a, b, \ldots, j\}$ that is a spanning tree in both).

4. Make up 2 matroids such that the matroid intersection algorithm needs at least 2 non-greedy steps (i.e. with $|Q| > 1$) to get a maximum common independent set.

(Solutions on page 87.)

Chapter 9

NP-Hardness

9.1 Definitions • 9.2 $\mathcal{NP}$-hard problems

9.1 Definitions

We will give a pretty informal definition of when an optimization problem is NP-hard. To give a more formal version would mean considerably more work, with little extra benefit for our purpose here.
$\mathcal{NP}$-hardness helps us classify which optimization problems are *hard* in the sense that there is no polynomial algorithm for them (unless $\mathcal{P} = \mathcal{NP}$), and which ones are the hardest among these. To make sense of this, we will first have to go through several other definitions.

- *Decision problems:* We have to distinguish between *decision problems* (yes/no questions, like "is there a perfect matching in a graph") and general computational problems, which include *optimization problems* ("find the maximum weight perfect matching"). The notions of $\mathcal{NP}$ and $\mathcal{NP}$-completeness are only defined for decision problems, while $\mathcal{NP}$-hardness will be defined for optimization problems.

- *Definitions:* A decision problem is

  - in $\mathcal{P}$ if there is an algorithm that solves any instance in polynomial time;
  - in $\mathcal{NP}$ if, when the answer for an instance is YES, there is a certificate proving it that can be checked in polynomial time;
  - $\mathcal{NP}$*-complete* if any problem in $\mathcal{NP}$ can be polynomially reduced to it.

- $\mathcal{NP}$*:* $\mathcal{NP}$ does not stand for "non-polynomial" (and does not exactly mean that), but for "non-deterministically polynomial"; very roughly, this means that you (actually, a Turing machine) could "non-deterministically" guess an answer, and then check it in polynomial time.
Most important for us is that if a decision problem is not in $\mathcal{NP}$, then there cannot be a polynomial algorithm for it, since that would provide a way of polynomially checking an answer. In other words, $\mathcal{P} \subset \mathcal{NP}$, and the million-dollar question is whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$.

- *Reductions:* We say that problem A reduces polynomially to problem B if any instance of A can be transformed to an instance of B, by a process that takes polynomial time. What this comes down to is that if we have a polynomial algorithm for problem B, then this transformation also gives a polynomial algorithm for A. So if one had a polynomial algorithm for one $\mathcal{NP}$-complete

decision problem, then one would have polynomial algorithms for all problems in $\mathcal{NP}$.
Below we will leave out the word "polynomially", since the reductions that we will consider are clearly polynomial.

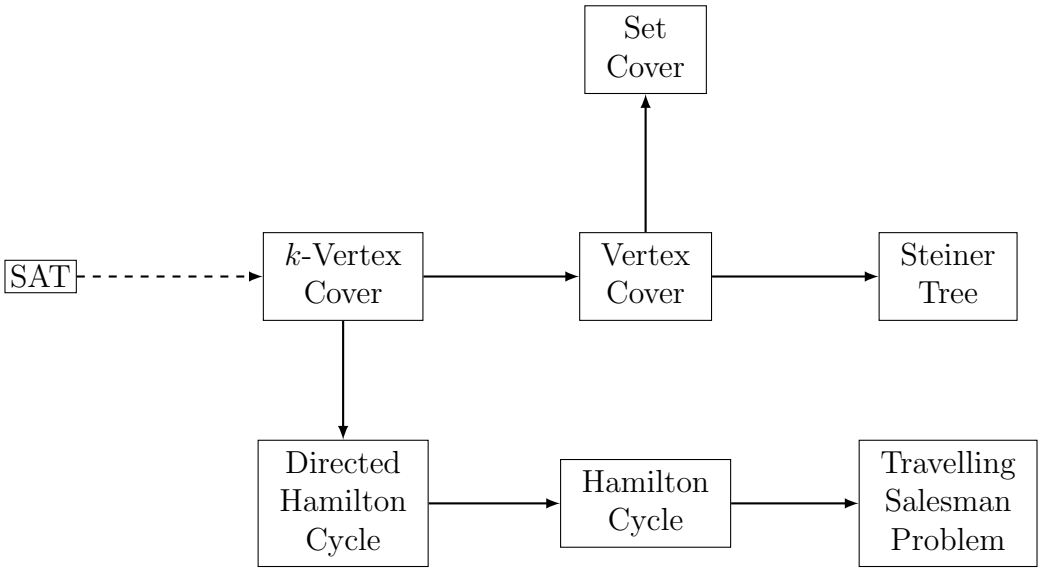
Optimization problems: An optimization problem (or a computational problem) is

- *$\mathcal{NP}$-hard* if some $\mathcal{NP}$-complete problem can be polynomially reduced to it.

This implies that if a problem is $\mathcal{NP}$-hard, then all $\mathcal{NP}$ problems can be reduced to it. Note that a decision problem is also a computational problem, so a decision problem is $\mathcal{NP}$-hard if and only if it is $\mathcal{NP}$-complete.

9.2 $\mathcal{NP}$-hard problems

We will show for a number of standard optimization problems that they are $\mathcal{NP}$-hard. This is usually done by proving that all $\mathcal{NP}$ problems reduce to the decision problem SAT, and then, for a given optimization problem X, reducing SAT to X, perhaps indirectly. This implies that X is $\mathcal{NP}$-hard, since if there was a polynomial algorithm for X, then that would give one for SAT, which would give a polynomial algorithm for every other $\mathcal{NP}$ problem.
SAT asks if a logical formula is satisfiable, i.e. if there is an assignment of true/false to the variables such that the formula comes out true. But introducing SAT in full detail would be too much of a digression for us, since it has more to do with logic than with optimization, and proving that it is $\mathcal{NP}$-complete would also require more formal definitions. For a proof, see for instance Schrijver's notes, 6.8.
So instead we will start by assuming that one decision problem, k-Vertex Cover, is $\mathcal{NP}$-hard, and then use that to prove it for other problems. If one knows SAT, then it is not too hard to prove that SAT reduces to k-Vertex Cover; see for instance Erickson's notes, 29.9.
Here is a diagram of the reductions that we will prove in this lecture. An arrow from A to B means that we will reduce A to B, which means that if there is a polynomial algorithm for B, then there is also one for A.



Recall that a *vertex cover* is a $C \subset V(G)$ such that $e \cap C \neq \emptyset$ for all $e \in E(G)$.

k-Vertex Cover: Given an undirected graph G, decide whether there is a vertex cover with $\leq k$ vertices.

VERTEX COVER: Given an undirected graph G, find a minimum cardinality vertex cover.

Theorem 9.2.1. *k-VERTEX COVER is $\mathcal{NP}$-complete and* VERTEX COVER *is $\mathcal{NP}$-hard.*

Proof. The first can be deduced from the $\mathcal{NP}$-completeness of SAT as mentioned above.
The second follows from the first by reducing k-VERTEX COVER to VERTEX COVER. This is easy, since given G and k, we can use VERTEX COVER to get a minimum vertex cover C, and we just have to check if $|C| \leq k$. □

SET COVER: Given a finite set X, a set of subsets $\mathcal{S} = \{S_i\}_{i \in I}$ of X, and $c : \mathcal{S} \to \mathbb{R}_{>0}$, find $J \subset I$ such that

$$\bigcup_{j \in J} S_j = X$$

with minimum $c(J) = \sum_{j \in J} c(S_j)$.

Theorem 9.2.2. SET COVER *is $\mathcal{NP}$-hard.*

Proof. VERTEX COVER reduces to SET COVER as follows. Given G, let $X = E(G)$ and $\mathcal{S} = \{S_v\}_{v \in V(G)}$ with $S_v = \delta(v)$. Then for c defined by $c(S_v) = 1$, a set cover for X is a vertex cover for G, with the same weight. Indeed, it is a $J \subset I = V(G)$ such that $\cup_{v \in J} \delta(v) = E(G)$, which implies that $e \cap J \neq \emptyset$ for all $e \in E(G)$, and we have $c(J) = |J|$. Hence a minimum weight set cover for X corresponds exactly to a minimum cardinality vertex cover in G. □

STEINER TREE: Given an undirected graph G, weights $w : E(G) \to \mathbb{R}_{>0}$, and $S \subset V(G)$, find a minimum-weight tree T in G such that $S \subset V(T)$.

Theorem 9.2.3. STEINER TREE *is $\mathcal{NP}$-hard.*

Proof. We reduce VERTEX COVER to STEINER TREE as follows. Given a graph G in which to find a vertex cover, we define a new graph H in which to find a Steiner tree. Let $V(H) = V(G) \cup E(G)$ and

$$E(H) = \{uv : u, v \in V(G)\} \cup \{ve : v \in e \in E(G)\}.$$

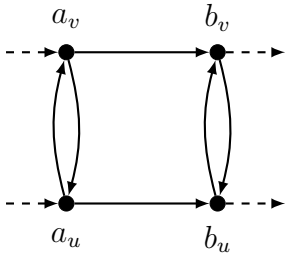
In other words, we take the complete graph on $V(G)$, add a vertex for every $e \in E(G)$, and connect a v in the complete graph to the vertex e if $v \in e$ in G.
Let $w(e) = 1$ for all $e \in E(H)$ and $S = E(G) \subset V(H)$. We show that a Steiner tree T for S in H corresponds to a vertex cover C in G with $w(T) = |C| + |S| - 1$, hence a minimum Steiner tree corresponds to a minimum vertex cover.
Let T be a Steiner tree for S in H. If we take $C = V(T) \backslash S$, then $C \subset V(G)$ is a vertex cover in G. Indeed, any $e = uv \in E(G)$ forms a vertex in S, which is only connected to the vertices u and v, so one of these must be in $V(T) \backslash S = C$, and it covers e. The vertex cover C has size $|C| = |V(T) \backslash S| = w(T) + 1 - |S|$, since $w(T) = |E(T)| = |V(T)| - 1$ because T is a tree on $V(T)$.

Let C be a vertex cover in G. Then as a subset of the complete graph part of H, there is a tree on C with $|C|-1$ edges. Add to this tree an edge to each $e \in S = E(G)$, using as few edges as possible. This is possible because C is a cover, and it gives a Steiner tree for S. It has weight $w(T) = |C| - 1 + |S|$. □

DIRECTED HAMILTON CYCLE: Given a directed graph G, decide whether it has a directed Hamilton cycle, i.e. a directed cycle C such that $V(C) = V(G)$.

Theorem 9.2.4. DIRECTED HAMILTON CYCLE *is $\mathcal{NP}$-hard (so also $\mathcal{NP}$-complete).*

Proof. We reduce k-VERTEX COVER to DIRECTED HAMILTON CYCLE. Let G be a graph for which we want to know if there is a Vertex Cover with $\leq k$ vertices. Given $uv \in E(G)$, we'll use the following building block H_{uv}:



For each vertex $u \in V(G)$, we link the H_{uv} into a chain as follows. Let $\delta(u) = \{uv_1, \ldots, uv_m\}$, and for each i, connect vertex b_u of H_{uv_i} to vertex a_u of $H_{uv_{i+1}}$. So for each vertex we have a chain of H_{uv}, and each H_{uv} is in two different chains, one for u and one for v.
Next we add a set K of k extra vertices, with for each vertex u an edge from each $x \in K$ to each first vertex a_u of the chain for u, and an edge from each last vertex b_u of the chain for u. We call the resulting graph H.
Now suppose there is a Hamilton cycle in H. It must pass through each H_{uv}, and it can do that in 3 ways: $a_u a_v b_v b_u$, $a_v a_u b_u b_v$, or separately through $a_u b_u$ and $a_v b_v$ (if it went any other way, there would be vertices left that it can never visit). Either way, if it enters at a_u then it leaves at b_u, so it must stay within the chain for u. It must enter the chain from some $x \in K$, and leave to some other $y \in K$.
Let C be the set of k vertices whose chain the Hamilton cycle goes through entirely. Then C is a vertex cover: Given an edge $uv \in E(G)$, the Hamilton cycle must pass through H_{uv} somehow, so it must use the chain for u or v (or both), so at least one of those is in C.
Conversely, if C is a vertex cover in G, then we can construct a Hamilton cycle in H from it. Write $C = \{u_1, \ldots, u_k\}$ and $K = \{x_1, \ldots, x_k\}$. From x_1 we go through the chain for u_1, out of the chain we go to x_2, then through the chain for u_2, etc, until we go through the chain for u_k, out of which we go back to x_1. For each H_{uv}, we go $a_u b_u$ if v is also in the cover, and $a_u a_v b_v b_u$ if not. □

HAMILTON CYCLE: Given an undirected graph G, decide whether it has a Hamilton cycle, i.e. a cycle C such that $V(C) = V(G)$.

Theorem 9.2.5. HAMILTON CYCLE *is $\mathcal{NP}$-hard (so also $\mathcal{NP}$-complete).*

Proof. We reduce Directed Hamilton Cycle to Hamilton Cycle. Given D in which to decide on the existence of a directed Hamilton cycle, for each $v \in V(D)$ take 3 vertices v_1, v_2, v_3, with 2 edges v_1v_2 and v_2v_3. Then for each directed edge $uv \in E(D)$, take an undirected edge u_3v_1. Call this undirected graph G. Now a directed Hamilton cycle in D corresponds exactly to a Hamilton cycle in G (possibly in reverse). □

Travelling Salesman Problem: Given an undirected complete graph G, with weights $w : E(G) \to \mathbb{R}_{\geq 0}$, find a Hamilton cycle of minimum weight.

Theorem 9.2.6. *The* Travelling Salesman Problem *is $\mathcal{NP}$-hard.*

Proof. We reduce Hamilton Cycle to Travelling Salesman Problem. Given a graph G in which to decide if there is a Hamilton cycle, give all edges weight 1, then add edges of weight 10 between any unconnected vertices. Then there is a Hamilton cycle in G if and only if the minimum weight Hamilton cycle in the new weighted graph has weight $|V(G)|$, since then it used only edge from G. □

9.3 Problems

1. Show that the following two optimization problems are $\mathcal{NP}$-hard:

   Independent Set: Given an undirected graph G, find a maximum cardinality independent set, i.e. a set $I \subset V(G)$ such that $E(I) = \{uv \in E(G) : u, v \in I\} = \emptyset$.

   Clique: Given an undirected graph G, find a maximum cardinality clique, i.e. a $K \subset V(G)$ such that $uv \in E(G)$ for all $u, v \in K$.

2. Show that the following optimization problem is $\mathcal{NP}$-hard:

   Longest Path: Given a directed graph G with weights $w : E(G) \to \mathbb{R}$, and $s, t \in V(G)$, find a directed path from s to t of maximum weight.

3. Show that the following optimization problem is $\mathcal{NP}$-hard:

   Integer Programming: Given a matrix $A \in \mathbb{Z}^{m \times n}$ and vectors $b \in \mathbb{Z}^m, c \in \mathbb{Z}^m$, find a vector $x \in \mathbb{Z}^n$ such that $Ax \leq b$ and cx is maximum, if possible.

4. Show that the following optimization problem is $\mathcal{NP}$-hard:

   Metric TSP: Let G be a complete undirected graph G with a weight function $d : E(G) \to \mathbb{R}_{>0}$ that satisfies the triangle inequality

   $$d(uw) \leq d(uv) + d(vw)$$

   for all $u, v, w \in V(G)$.
   Find a minimum weight Hamilton cycle in G.

(Solutions on page 88.)

Chapter 10

Approximation Algorithms

10.1 Definition

Consider a general minimization problem, asking for a feasible object S such that its value $w(S)$ is minimum. Write S^* for an optimum solution.
A *k-approximation algorithm* (with $1 \leq k \in \mathbb{R}$) for such a problem is a polynomial algorithm that returns a feasible object S such that

$$w(S) \leq k \cdot w(S^*).$$

The number k is called the approximation factor. One could make a similar definition for maximization problems, with $0 < k \leq 1$, but actually all the problems that we will see approximation algorithms for happen to be minimization problems.

Remarks:
- This definition of an approximation algorithm depends on a worst-case bound, which means that an algorithm might have a worse approximation factor than another one, but do much better in practice. For instance, the "average" quality of approximation might be better (although it would be tricky to define "average"). The situation is a bit similar to using worst-case bounds for the running time of an algorithm.
- An optimization problem need not have any approximation algorithm at all. Set Cover and general TSP, for instance, do not have a k-approximation algorithm for any k (we will prove this for TSP in the next lecture). In this case, it might still be worth it to consider algorithms where the approximation factor depends on the size of the input. For instance, we will see a $\log(n)$-approximation algorithm for Set Cover, where $n = |X|$ is the size of the ground set.
- When there is a polynomial reduction from problem A to problem B, we know that a polynomial algorithm for B gives a polynomial algorithm for A. But this does not imply that a k-approximation algorithm for B also gives a k-approximation algorithm for A. So even for problems that are equivalent in terms of reductions, there may be big differences in how well they can be approximated. For instance, Vertex Cover and Independent Set are equivalent, but Vertex Cover has a 2-approximation algorithm, as we'll see below, while Independent Set does not have a k-approximation algorithm for any k (which we won't prove).

10.2 Vertex Covers

VERTEX COVER: Given an undirected graph G, find a minimum cardinality vertex cover (a $C \subset V(G)$ such that $C \cap e \neq \emptyset$ for all $e \in E(G)$).

Cover-from-Matching Algorithm

1. Greedily find a maxim*al* matching M in G;
2. Return $C = \bigcup_{e \in M} e$.

Theorem 10.2.1. *The algorithm above is a 2-approximation algorithm.*

Proof. The C returned by the algorithm is a vertex cover, because if there was an edge that it did not cover, then that edge could be added to the matching, contradicting its maximality. We have to show that if C^* is a minimum vertex cover, then $|C| \leq 2|C^*|$. Now C^* must cover every edge in the maximal matching M, and no vertex can cover more than one edge of a matching, so $|C^*| \geq |M|$. On the other hand, $|C| = 2|M|$ by definition. Together this gives $|C| \leq 2|C^*|$. □

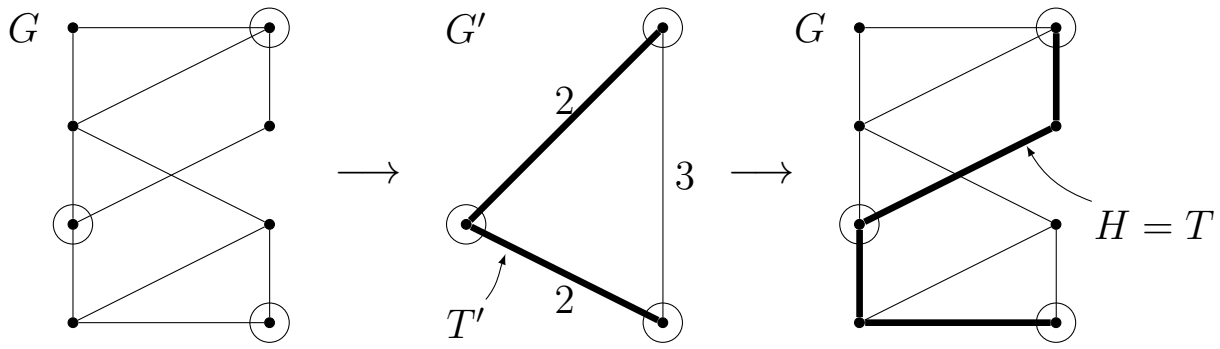
10.3 Steiner Trees

STEINER TREE: Given an undirected graph G, weights $w : E(G) \to \mathbb{R}_{>0}$, and $S \subset V(G)$, find a minimum-weight tree T in G such that $S \subset V(T)$.

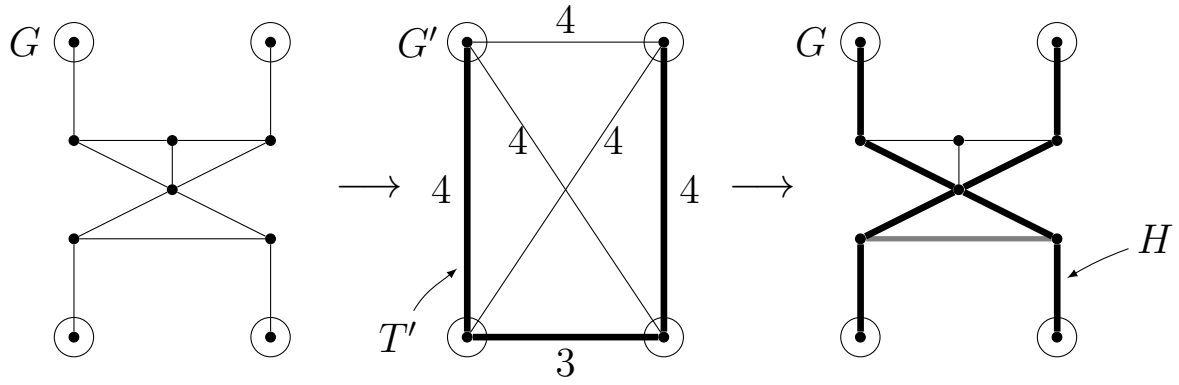
Steiner Tree Algorithm

1. Compute the graph of distances on S and call it G';
   ($V(G') = S$, $E(G') = \{s_1s_2 : s_1, s_2 \in S\}$, $d(s_1s_2) = d_G(s_1, s_2)$;
   $d(s_1s_2)$ is the length of the shortest s_1s_2-path in G with weights w; to compute this we replace every undirected edge by two directed edges, then use Dijkstra since the weights are positive)
2. Find an MST T' on G' with weights the distances $d(uv)$;
3. Let H be the subgraph of G corresponding to T';
   (i.e. H is constructed by taking for every $uv \in E(T)$ a uv-path of length $d(uv)$)
4. Find an MST T of H; return T.

Here is an example of the algorithm in action, where the weights in G are all 1; in other words, the graph is unweighted.



In this case the last step was not really necessary, since H was already a tree itself. In fact, when $|S| = 3$, this will always be the case. But here is an example with $|S| = 4$ where it is not.



Here H has a 3-cycle, so to get a tree T we remove one edge from that cycle.

Theorem 10.3.1. *The algorithm above is a 2-approximation algorithm.*

Proof. The returned T is clearly a tree, and $S \subset V(T)$, so T is a Steiner tree. We need to show that if T^* is a minimum weight Steiner tree, then $w(T) \leq 2w(T^*)$. This will take several steps. Let K be the graph obtained by doubling every edge in T^*. Then the degree of every vertex of K is even, so K has an Euler tour E (see section 5.3 of these lecture notes). Clearly $w(E) = w(K) = 2w(T^*)$.
From E we construct a cycle E' in the distance graph G' as follows. Let $S = \{s_1, s_2, \ldots, s_m\}$, where the s_i are numbered in the order in which they appear in E, starting from some arbitrarily chosen vertex. Also write $s_{m+1} = s_1$. Then we take E' to be the cycle $s_1 s_2 \cdots s_m s_{m+1}$ in G.
We observe that $d(E') \leq w(E)$. This follows from the fact that $d(s_i s_{i+1})$ is the length of the shortest path between s_i and s_{i+1}, which must be shorter than the path from s_i to s_{i+1} in E. So if we let P_i be the path from s_i to s_{i+1} in E, then

$$w(E) = \sum w(P_i) \geq \sum d(s_i s_{i+1}) = d(E').$$

Next we note that the cycle E' contains a spanning tree of G' (remove any edge), which means that $d(T') < d(E')$. We also have $w(H) \leq d(T')$, since H is built out of paths Q_e for all $e \in E(T')$, with $w(Q_e) = d(e)$, so

$$w(H) \leq \sum w(Q_e) = \sum d(e) = d(T').$$

Finally we can put everything together:

$$w(T) \leq w(H) \leq d(T') < d(E') \leq w(E) = w(K) = 2w(T^*).$$

So we actually proved that $w(T) < 2w(T^*)$, but in the problem set we will see that the number 2 cannot be improved on. □

There are approximation algorithms for Steiner Tree with approximation factor < 2, although they are not as simple as this one. The current best-known one was found in 2010 and has $k = 1.39$. I'm especially mentioning this because the authors, Byrka, Grandoni, Rothvoß, and Sanità, were working at EPFL at the time!
On the other hand, it was proven that no k-approximation algorithm exists for $k \leq 1.0006$ (unless $\mathcal{P} = \mathcal{NP}$).
Compare this to the situation for Vertex Cover: There is no algorithm known with $k < 2$, so surprisingly the one we saw is the best that is known. It was also proven that $k \leq 1.36$ is not possible (unless $\mathcal{P} = \mathcal{NP}$).

10.4 Set Cover

We will consider the cardinality version of Set Cover here. In the last lecture we proved that the weighted version is $\mathcal{NP}$-hard, but this does not imply directly that the cardinality version is too. It is, though, since in the proof we reduced Vertex Cover to an instance of Set Cover which was actually an instance of the cardinality version. The algorithm given here could be easily extended to the weighted version.

> Set Cover: Given a finite set X, a set of subsets $\mathcal{S} = \{S_i\}_{i \in I}$ of X, find a minimum cardinality $J \subset I$ such that $\bigcup_{j \in J} S_j = X$.

There is no approximation algorithm like for the previous examples, as the following theorem states. It is called an *inapproximability theorem*. The proof is a bit too far afield, but in the next lecture we will see the proof of an inapproximability theorem for TSP.

Theorem 10.4.1. *There is no k-approximation algorithm for* Set Cover *for any $k \in \mathbb{R}$.*

Because of this fact, it would still be worth it to have an algorithm that approximates by a factor $f(n)$ ($n = |X|$) that grows slowly with n. Below we will see a greedy algorithm that does this with $f(n) = \log(n)$. A more advanced approximability theorem states that this is best possible, in the sense that there is no $f(n)$-approximation algorithm that grows more slowly than $c \log(n)$ for some constant c.

> **Greedy Set Cover Algorithm**
>
> 1. Let $Y = X$ and $J = \emptyset$;
> 2. Pick S_j with maximum $|S_j \cap Y|$;
> 3. Set $J := J \cup \{j\}$ and $Y := Y \backslash S_j$;
> 4. If $Y \neq \emptyset$ go back to 2; else return J.

Theorem 10.4.2. *The algorithm above is an $H(n)$-approximation algorithm, where $n = |X|$ and $H(n)$ is the harmonic series $H(n) = \sum_{i=1}^{n} \frac{1}{i}$, which has $\log(n) < H(n) < \log(n) + 1$.*

Proof. We have to show that if J^* is a minimum set cover, then $|J| \leq H(n) \cdot |J^*|$.
Let $X = \{x_1, \ldots, x_n\}$, with the x_i in the order in which the algorithm picked them. For $j \in J$ let Y_j be the Y right before j was added, and for each $x_i \in S_j \cap Y_j$, define $c_i = |S_j \cap Y_j|$. In other words, c_i is the number of x_k that became covered in the same step as x_i. For instance,

if in a step x_7, x_8, x_9 are all the elements that become covered, then $c_7 = c_8 = c_9 = 3$. With this definition we have

$$\sum_{i=1}^{n} \frac{1}{c_i} = \sum_{j \in J} \sum_{x_i \in S_j \cap Y_j} \frac{1}{c_i} = \sum_{j \in J} 1 = |J|.$$

Consider the elements $x_i, \ldots, x_n$. By the greediness, no S_j covers more than c_i of these $n-i+1$ elements. Hence any set cover needs at least $(n-i+1)/c_i$ sets to cover these elements, so in particular

$$|J^*| \geq \frac{n-i+1}{c_i}.$$

Therefore

$$|J| = \sum_{i=1}^{n} \frac{1}{c_i} = |J^*| \cdot \sum_{i=1}^{n} \frac{1}{n-i+1} = |J^*| \cdot \sum_{k=1}^{n} \frac{1}{k} = H(n) \cdot |J^*|.$$

□

10.5 Problems

1. Show that for Vertex Cover the greedy approach (repeatedly add a vertex if the result is still a vertex cover) does not give a k-approximation for any k.
   Show by an example that the Cover-from-Matching algorithm for Vertex Cover is not a k-approximation algorithm for any $k < 2$.

2. Give an example where the Steiner Tree approximation algorithm does not give a minimum Steiner Tree.
   Show that it is not a k-approximation algorithm for any $k < 2$.

3. The Metric Steiner Tree problem is the special case of Steiner Tree where the graph is complete and the weights d are ≥ 0 and satisfy the triangle inequality

   $$d(u, w) \leq d(u, v) + d(v, w).$$

   Show that any algorithm that simply returns a minimum spanning tree of the special set S is a 2-approximation algorithm for Metric Steiner Tree.

4. Show that Set Cover reduces to Directed Steiner Tree (which is like Steiner Tree, but the graph is directed, and the tree should be directed).

(Solutions on page 90.)

Chapter 11

Metric TSP

11.1 Inapproximability of TSP • 11.2 Double-Tree Algorithm • 11.3 Christofides' Algorithm

Travelling Salesman Problem (TSP): Given an undirected complete graph G, with weights $w : E(G) \to \mathbb{R}_{\geq 0}$, find a Hamilton cycle of minimum weight.

11.1 Inapproximability of TSP

Theorem 11.1.1. *There is no k-approximation algorithm for TSP for any $k \in \mathbb{R}$, unless $\mathcal{P} = \mathcal{NP}$. The same is true if we replace k by any function $f(n)$ of $n = |V(G)|$ that is computable in polynomial time.*

Proof. We show that, given a k-approximation algorithm for TSP, we get a polynomial algorithm for the decision problem Hamilton Cycle. Since we proved that problem to be $\mathcal{NP}$-complete, this would mean $\mathcal{P} = \mathcal{NP}$.
Let G be an instance of Hamilton Cycle, i.e. a graph for which we want to determine if it contains a Hamilton cycle. We define a new complete graph G' with $V(G') = V(G)$, and weights w that are 1 for edges that are also in G, and kn otherwise.
We claim that G has a Hamilton cycle if and only if the k-approximation algorithm applied to G' finds a Hamilton cycle of weight $\leq kn$. If G has no Hamilton cycle, then every Hamilton cycle in G' has weight $> kn$, since it must use one of the edges of weight kn. If G does have a Hamilton cycle, then it gives a Hamilton cycle T^* of weight n in G', which must be minimum. Then the k-approximation algorithm will return a Hamilton cycle of weight $\leq k \cdot w(T^*) = kn$.
The second statement of the theorem follows by just replacing k by $f(n)$ in the proof above. The only complication is that the function should be computable in polynomial time, since otherwise the resulting algorithm for Hamilton Cycle would not be polynomial. □

Note that a function that is computable in polynomial time need not be a polynomial; for instance, $f(n) = 2^n$ can be computed in n multiplication steps. We won't go into what functions have this property or not, the point here is that an approximation factor like the $\log(n)$ that we saw for Set Cover is not possible for TSP (unless, sigh, $\mathcal{P} = \mathcal{NP}$). This follows since $\log(n) \leq n$ and n is computible in polynomial time (by the ingenious algorithm that, given n, returns n).

11.2 Double-Tree Algorithm

Since approximating the general TSP is hopeless, it makes sense to consider restricted versions, like the metric one that we saw in Problem Set 9. Most applications satisfy this condition. Note that this is not the same as the *Euclidean* TSP, where the vertices are points in $\mathbb{R}^2$ and the weights their Euclidean distances.

METRIC TSP: Let G be a complete undirected graph G with a weights $d : E(G) \to \mathbb{R}_{\geq 0}$ that satisfy the triangle inequality

$$d(uw) \leq d(uv) + d(vw)$$

for all $u, v, w \in V(G)$.
Find a minimum-weight Hamilton cycle in G.

Note that the inapproximability proof above does not apply here, because the graph G' constructed there does not satisfy the triangle inequality: For most H, there will be a triangle in G' with one edge of weight kn and two edges of weight 1, and $kn > 1 + 1$.

The following lemma will be used in the proofs for our next two algorithms. Recall that a *tour* is a closed walk, i.e. a sequence of touching edges that starts and ends at the same vertex, and is allowed to repeat vertices and edges.

Lemma 11.2.1 (Shortcutting)**.** *Let G be a complete graph with weights d satisfying the triangle inequality.*
Given a tour E that visits all vertices in G, one can construct a Hamilton cycle C with $d(C) \leq d(E)$.

Proof. Write $V(G) = \{v_1, v_2, \ldots, v_n\}$, with the vertices in the order in which they appear in in E, starting from an arbitrary vertex and in an arbitrary direction. Also write $v_{n+1} = v_1$. Then we take the Hamilton cycle $C = v_1 v_2 \cdots v_n v_{n+1}$.
For each i, let P_i be the walk along E from v_i to v_{i+1}. Then by repeated applications of the triangle inequality and completeness of the graph we have

$$d(v_i v_{i+1}) \leq d(P_i).$$

Therefore

$$d(C) = \sum_{i=1}^{n} d(v_i v_{i+1}) \leq \sum_{i=1}^{n} d(P_i) = d(E).$$

□

Double-Tree Algorithm

1. Find an MST T in G with respect to d;
2. Double the edges of T to obtain a graph D;
3. Find an Euler tour E in D;
4. Shortcut E to get a Hamilton cycle C and return C.

Theorem 11.2.2. *The Double-Tree algorithm is a 2-approximation algorithm for Metric TSP.*

Proof. If C^* is a minimum Hamilton cycle we have

$$d(C) \leq d(E) = 2d(T) \leq 2d(C^*).$$

The first inequality comes from Lemma 11.2.1, and the equality is clear from the construction. The last inequality follows from the fact that a Hamilton cycle minus any edge e is a spanning tree, so $d(T) \leq d(C^*) - d(e) \leq d(C^*)$. □

11.3 Christofides' Algorithm

Christofides' Algorithm for Metric TSP

1. Find an MST T in G with respect to d;
2. Let H be the complete graph on the vertices that have odd degree in T;
3. Find a minimum weight perfect matching M in H, with weights d;
4. Find an Euler tour E in the graph $(V(G), E(T) \cup M)$;
5. Shortcut E to get a Hamilton cycle C and return C.

To see that step 3 is well-defined, recall from basic graph theory the fact that the number of odd-degree vertices is always even (proof: the sum of the degrees is twice the number of edges). So $|H|$ is even and has a perfect matching.
Step 4 is well-defined because adding the matching edges exactly turns the odd-degree vertices into even-degree vertices, so that an Euler tour exists.

Theorem 11.3.1. *Christofides' algorithm is a 3/2-approximation algorithm for* Metric TSP.

Proof. If C^* is a minimum Hamilton cycle we have

$$d(C) \leq d(E) = d(T) + d(M) \leq d(C^*) + \frac{1}{2}d(C^*) = \frac{3}{2}d(C^*).$$

Everything is similar to in the previous proof, except for the inequality $d(M) \leq d(C^*)/2$.
We'll prove that $d(C^*) \geq 2d(M)$. Write $H = \{h_1, \ldots, h_m\}$ in the order of appearance in C^*, and define the Hamilton cycle on H

$$C_H = h_1 h_2 \cdots h_m h_1.$$

Then $d(C^*) \geq d(C_H)$ by the triangle inequality.
Since $|H|$ is even, $|E(C_H)|$ is even, so we have $E(C_H) = M_1 \cup M_2$, a disjoint union of two perfect matchings of H (simply put edges of C_H alternately in M_1 and M_2). Since M is a minimum weight perfect matching on H, we have $d(M) \leq d(M_i)$. Therefore

$$d(C^*) \geq d(C_H) = d(M_1) + d(M_2) \geq 2d(M).$$

□

A few research results
Christofides' algorithm has the best-known worst-case approximation factor for METRIC TSP. The best lower bound is $220/219 = 1.004\cdots$, i.e. no k-approximation is possible with $k \leq 220/219$ (yes, unless $\mathcal{P} = \mathcal{NP}$). On the other hand, for the EUCLIDEAN TSP (which is also $\mathcal{NP}$-hard), there is an $(1+\varepsilon)$-approximation algorithm for any $\varepsilon > 0$ (by Arora in 1998).
Instead of looking at worst-case bounds, one can also empirically test the typical behavior. On a set of random examples (see the book by Cook, Cunningham, Pulleyblank, Schrijver, Ch.7), Christofides has an average approximation factor $l = 1.09$. For comparison, the greedy approach (repeatedly picking the closest unvisited vertex) has $l = 1.26$ on the same examples, and the Double-Tree algorithm does worse (it wasn't even included in this test). So Christofides' algorithm does pretty well in practice, but it has the drawback of being pretty slow, since it has to find a minimum weight perfect matching, which we know is a pretty complicated algorithm. It is also terrible for non-metric graphs.
In the last lecture, we'll see some of the approaches that do not have worst-case bounds, but are faster, have l as low as 1.01, and do all right on non-metric graphs.

11.4 Problems

1. Give tight examples for the Double-Tree algorithm and Christofides' algorithm (i.e. give examples which show that these are not k-approximation algorithms for any $k < 2$ and $k < 3/2$, respectively).

2. Show that a k-approximation algorithm for Metric TSP gives a k-approximation algorithm for TSPR.

> TRAVELLING SALESMAN PROBLEM WITH REPETITIONS (TSPR): Given an undirected complete graph G, with weights $w : E(G) \to \mathbb{R}_{\geq 0}$, find a tour (closed walk) that visits every vertex *at least* once.

3. Find a 4/3-approximation algorithm for 1-2-TSP.
(Hint: Cover the vertices by cycles of minimum total weight, then patch them together.)

> TRAVELLING SALESMAN PROBLEM (1-2-TSP): Given an undirected complete graph G, with weights $w : E(G) \to \mathbb{R}_{\geq 0}$ such that all $w(e) \in \{1, 2\}$, find a Hamilton cycle of minimum weight.

(Solutions on page 92.)

Chapter 12

Bin Packing

12.1: $\mathcal{NP}$-hardness • 12.2: Simple Algorithms • 12.3: 3/2-Approximation • 12.4: Asymptotic $(1+\varepsilon)$-Approximation Scheme

BIN PACKING: Given a set $I = \{1, \ldots, n\}$ of *items* with *sizes* $s_i \in (0, 1]$, and a set $B = \{1, \ldots, n\}$ of *bins* with capacity 1, put all the items in as few bins as possible.
More formally, find an assignment $a : I \to B$ such that for each $b \in B$ we have $\sum_{i:a(i)=b} s_i \leq 1$ and $a(I)$ is minimal.

Note that an instance of BIN PACKING is a list of numbers $s_1, \ldots, s_n$, the definition just uses the sets I and B to make things more conceptual. In particular, that $|B| = n$ does not really matter, one could just as well consider an unlimited number of bins.

12.1 $\mathcal{NP}$-hardness

We first have to show that BIN PACKING is $\mathcal{NP}$-hard. We'll do this by reducing VERTEX COVER to SUBSET SUM, then reducing SUBSET SUM to PARTITION, and finally PARTITION to BIN PACKING. (I don't know of a shorter way.)

SUBSET SUM: Given a sequence $a_1, \ldots, a_n, t$ of nonnegative integers, decide whether there is a subset $S \subset \{1, \ldots, n\}$ such that

$$\sum_{i \in S} a_i = t.$$

PARTITION: Given a sequence $a_1, \ldots, a_n$ of nonnegative integers, decide whether there is a subset $S \subset \{1, \ldots, n\}$ such that

$$\sum_{i \in S} a_i = \sum_{j \notin S} a_j.$$

Theorem 12.1.1. SUBSET SUM *and* PARTITION *are $\mathcal{NP}$-complete.*

Proof sketch. One can prove this by reducing k-VERTEX COVER (see Lecture 9) to SUBSET SUM, and SUBSET SUM to PARTITION.
Given a graph G and a number k, we encode it as a sequence of integers as follows. We number

the edges of G from 0 to $|E(G)|-1$, and take $a_i = 10^i$ for each edge i. Then we also include for each vertex v of G the integer

$$b_v = 10^{|E(G)|} + \sum_{i \in \delta(v)} 10^i.$$

Finally we take

$$t = k \cdot 10^{|E(G)|} + \sum_{i=0}^{|E(G)|-1} 2 \cdot 10^i.$$

The reader can now check that G has a vertex cover with k vertices if and only if there is a subsequence of the a_i, b_v that sums to t. The idea is that the first term in t forces exactly k vertices to be used, and the second term forces each edge to be covered by a vertex.
Next we reduce Subset Sum to Partition. Given an instance $a_1, \ldots, a_n, t$ of Subset Sum, with $A = \sum_{i=1}^n a_i$, we construct an instance $a_1, \ldots, a_n, a_{n+1}, a_{n+2}$ of Partition by setting $a_{n+1} = 2A - t$ and $a_{n+2} = A + t$. Then the reader can check that the answer for the Subset Sum is YES if and only if the answer for the Partition instance is YES. □

Theorem 12.1.2. Bin Packing *is $\mathcal{NP}$-hard. It has no k-approximation algorithm with $k < 3/2$ (unless $\mathcal{P} = \mathcal{NP}$).*

Proof. We reduce Partition to Bin Packing. Given an instance $a_1, \ldots, a_n$ of Partition, we let $A = \sum a_i$ and consider the instance of Bin Packing with $s_i = 2a_i/A$. It is easy to see that the answer to the Partition instance is YES if and only if the minimum number of bins for the Bin Packing instance is 2.
If there was a k-approximation algorithm for Bin Packing with $k < 3/2$, then when the exact minimum is 2 bins, this algorithm would always find it, since 3 bins would be an approximation with factor $\geq 3/2$. So by the reduction above, this algorithm would exactly solve Partition in polynomial time, which would imply $\mathcal{P} = \mathcal{NP}$. □

12.2 Simple algorithms

We will first informally consider the simplest greedy-like algorithms that one would think of, then in the next section we will analyze the best of these more formally.
We will illustrate these simple algorithms for one example, given by the s_i (for convenience we will take the bin capacities to be 10)

$$3, 6, 2, 1, 5, 7, 2, 4, 1, 9.$$

- **Next Fit**: *Put the current item in the current bin if it fits, otherwise in the next bin.* On the example this would give the following packing:

  $$[3, 6] - [2, 1, 5] - [7, 2] - [4, 1] - [9]$$

  On the problem set you are asked to prove that this is a 2-approximation algorithm. The approximation factor is less good than for the next two algorithms, but a practical advantage is that Next Fit is what is called *online*: It only considers the current item, without looking ahead or changing previous items, and it only considers the current bin, without ever changing previous bins. The next two algorithms do not have both these properties.

- **First Fit**: *Put the current item into the first bin it fits into.*
  On the example this would give:

$$[3, 6, 1] - [2, 5, 2, 1] - [7] - [4] - [9]$$

  It has been proved (with some effort) that this is roughly a $\frac{17}{10}$-approximation algorithm.

- **First Fit Decreasing**: *Sort the items in decreasing order, then do First Fit.*
  On the example this would give:

$$[9, 1] - [7, 3] - [6, 4] - [5, 2, 2, 1]$$

  We will see below that this is a 3/2-approximation algorithm, and above we saw that this is best possible in terms of the approximation factor.
  A tight example for the factor 3/2 for First Fit Decreasing is

$$2, 3, 2, 5, 6, 2 \quad \to \quad [6, 3] - [5, 2, 2] - [2],$$

  where there is clearly an example using only 2 bins.

12.3 3/2-Approximation Algorithm

As we saw above, an approximation algorithm for Bin Packing with approximation factor $< 3/2$ is not possible. Here we'll see something that we haven't seen before in this course, namely an approximation algorithm that exactly matches a lower bound.
However, this lower bound is a little misleading, because it really only holds for instances that come down to deciding between 2 or 3 bins, as can be seen from the proof of Theorem 12.1.2. We'll see in the next section that if you know that more bins will be needed, better approximations are possible.

First Fit Decreasing Algorithm (FFD)

1. Sort the items by decreasing size and relabel them so that $s_1 \geq s_2 \geq \cdots \geq s_n$;
2. For $i = 1$ to n, put i in the first bin that it fits into, i.e.

$$a(i) = \min \left\{ b \in B : \left(\sum_{j:a(j)=b} s_j \right) + s_i \leq 1 \right\};$$

3. Return a.

Theorem 12.3.1. *First Fit Decreasing is a 3/2-approximation algorithm for* Bin Packing.

Proof. Let a be the assignment found by the algorithm, with $k = |a(I)|$, and let a^* be the minimal assignment, with $k^* = |a^*(I)|$. We want to show that $k \leq \frac{3}{2}k^*$. We will assume that the items have been sorted. We'll write $S = \sum_{i \in I} s_i$, so we have the trivial bound

$$k^* \geq S.$$

Let $b \leq k$ be an arbitrary bin used by a. We will analyze the following two cases: b contains an item of size $> 1/2$, or it does not.

Suppose b contains an item i of size $s_i > 1/2$. Then the previously considered items $i' < i$ all have $s_{i'} > 1/2$, and each bin $b' < b$ must contain one of these, so we have $\geq b$ items of size $> 1/2$. No two of these can be in the same bin in any packing, so a^* uses at least b bins, i.e. $k^* \geq b$.
Suppose b does not contain an item of size $> 1/2$. Then no used bin $b'' > b$ contains an item of size $> 1/2$, which implies that each of these bins contains ≥ 2 items, except maybe for the last used one (bin k). So the $k - b$ bins $b, b+1, \ldots, k-1$ together contain $\geq 2(k-b)$ items. We know that none of these items would have fit in any bin $b' < b$.
We consider two subcases. If $b \leq 2(k-b)$, then we can imagine adding to every bin $b' < b$ one of these $2(k-b)$ items, which would give us $b-1$ overfilled bins. This implies that $S > b-1$. On the other hand, if $b > 2(k-b)$, then we can imagine adding each of the $2(k-b)$ elements to a different bin $b' < b$, giving us $2(k-b)$ overfilled bins. Then $S > 2(k-b)$.
So for any b we have in all cases that either $k^* \geq b$ or $k^* \geq 2(k-b)$. Now we choose b so that it will more or less maximize the minimum of b and $2(k-b)$: Equating $b = 2(k-b)$ gives $b = \frac{2}{3}k$, and we take $b = \lceil \frac{2}{3}k \rceil$ to get an integer. Then we have that $k^* \geq \lceil \frac{2}{3}k \rceil \geq \frac{2}{3}k$, or $k^* \geq 2(k - \lceil \frac{2}{3}k \rceil) \geq \frac{2}{3}k$. Hence we have $k \leq \frac{3}{2}k^*$. □

12.4 Asymptotic Approximation Scheme

We won't define in general what an approximation scheme is, but just state what it means in this case. The word "asymptotic" refers to the fact that the approximation factor only holds for large instances above some threshold. Non-asymptotic approximation schemes do exist, for instance for Knapsack or Euclidean TSP.
These schemes are less useful in practice than they may sound, because the running times tend to be huge (although polynomial). Another drawback in this case is that the running time is polynomial in n, but not in $1/\varepsilon$. So this result is more theoretical, and in practice fast approximation algorithms like the one above might be more useful.
As above we write k for the number of bins used by the algorithm and k^* for the minimum number of bins.

Theorem 12.4.1. *For any $0 < \varepsilon \leq 1/2$ there is an algorithm that is polynomial in n and finds an assignment having at most $k \leq (1+\varepsilon)k^*$ bins, whenever $k^* \geq 2/\varepsilon$.*

To give some feeling for this statement, suppose you want a 5/4-approximation algorithm. Then $\varepsilon = 1/4$, so we need $k^* \geq 2/\varepsilon = 8$. Of course we don't know k^* beforehand, but using the trivial bound $k^* \geq \sum s_i$ we see that the theorem gives a 5/4-approximation algorithm for any instance with $\sum s_i \geq 8$.
Or the other way around, suppose we know that we're dealing with instances that have $\sum s_i \geq 100$. Then also $k^* \geq 100$, so $\varepsilon = 1/50$ will work, giving us a 1.02-approximation algorithm.
The proof requires the following two lemmas.

Lemma 12.4.2. *Let $\varepsilon > 0$ and $d \in \mathbb{N}$ be constants. There is a polynomial algorithm that exactly solves any instance of* Bin Packing *with all $s_i \geq \varepsilon$ and $|\{s_1, \ldots, s_n\}| \leq d$.*

Proof. This can be done simply by enumerating all possibilities and checking each one.
The number of items in a bin is at most $L = \lfloor 1/\varepsilon \rfloor$. Therefore the number of ways of putting items in a bin (identifying items of the same size and disregarding the ordering) is less than $M = \binom{L+d-1}{L}$ (using a standard formula from combinatorics/probability theory). The number of feasible assignments to n bins is then $N = \binom{n+M-1}{M}$ (by the same formula).
Now $N \leq c \cdot n^M$, and M is independent of n, so the number of cases to check is polynomial in n. Checking an assignment takes constant time, so this gives an exact polynomial algorithm. (But note that the running time is badly exponential in $1/\varepsilon$ and d.) □

Lemma 12.4.3. *Let $\varepsilon > 0$ be a constant. There is a polynomial algorithm that gives a $(1+\varepsilon)$-approximation for any instance of* Bin Packing *that has all $s_i \geq \varepsilon$.*

Proof. Let I be the list of items. Sort them by increasing size, and partition them into $P+1$ groups of at most $Q = \lfloor n/P \rfloor$ items (so the smallest Q items in a group, then the next Q items, etc., and the last group may have fewer than Q items). We will choose the integer P later.
We construct two new lists H and J. For H, round down the size of the items in each group to the size of the smallest item of that group. For J, round up to the largest size in each group. Let $k^*(H)$, $k^*(I)$, and $k^*(J)$ be the minimal numbers of bins for each instance.
Then we clearly have

$$k^*(H) \leq k^*(I) \leq k^*(J),$$

since a minimal assignment for one list will also work for a list that is smaller entry-by-entry. On the other hand, we have

$$k^*(J) \leq k^*(H) + Q \leq k^*(I) + Q,$$

because, given an assignment a_H for H, we get an assigment a_J for J as follows. The items of group $i+1$ in H will be larger than (or equal to) the items of group i in J, so we can let a_J assign the items of group i in J to the bins that the items in group $i+1$ in H were assigned to by a_H. This leaves the $\leq Q$ items of the largest group in J, which we assign to the Q extra bins. Note that we've ignored the smallest group of H.
If we choose P so that $Q \leq \varepsilon \cdot k^*(I)$, then we have $k^*(J) \leq (1+\varepsilon) \cdot k^*(I)$. We can choose $P = \lceil 1/\varepsilon^2 \rceil$, since then $Q = \lfloor \varepsilon^2 n \rfloor \leq \varepsilon \cdot k^*(I)$, using that $k^*(I) \geq \sum s_i \geq \varepsilon \cdot n$.
Now we have what we want, since we can apply Lemma 12.4.2 to J, with ε and $d = P+1$. The resulting assignment for J will also work for I. We do not need to assume $\varepsilon \leq 1/2$, because we already have a 3/2-approximation algorithm. □

Proof of Theorem 12.4.1. Given a list of items I, remove all items of size $< \delta$ to get a list I' (with $\delta > 0$ to be chosen later). We have $k^*(I') \leq k^*(I)$. Lemma 12.4.3 gives an assignment a' of I' using $k(I') \leq (1+\delta) \cdot k^*(I')$ bins.
We now assign the removed small items to the bins used by a' as far as possible, using First Fit; we may have to use some new bins. If we do not need new bins, we are done since then we have an assignment of I with $k(I) = k(I') \leq (1+\delta) \cdot k^*(I') \leq (1+\varepsilon) \cdot k^*(I)$ bins, for any choice of δ with $\delta \leq \varepsilon$.
If we do need new bins, then we know that all $k(I)$ bins (except possibly the last one) have remaining capacity less than δ. This means that

$$k^*(I) \geq \sum_{i \in I} s_i \geq (k(I)-1)(1-\delta),$$

so

$$k(I) \leq \frac{k^*(I)}{1-\delta} + 1 \leq (1+2\delta) \cdot k^*(I) + 1,$$

using the inequality $\frac{1}{1-x} \leq 1+2x$, which holds for $0 < x \leq 1/2$.
Now choose $\delta = \varepsilon/4$. Then when $k^*(I) \geq 2/\varepsilon$ we have $1 \leq \varepsilon \cdot k^*(I)/2$, so

$$k(I) \leq (1+\frac{\varepsilon}{2}) \cdot k^*(I) + 1 \leq (1+\frac{\varepsilon}{2}) \cdot k^*(I) + \frac{\varepsilon}{2} \cdot k^*(I) \leq (1+\varepsilon) \cdot k^*(I).$$

□

The algorithm would now look as follows. As mentioned above, it is not an algorithm that would really be implemented, so this is meant more to as an overview of the steps in the proof.

Asymptotic $(1+\varepsilon)$-Approximation Algorithm for Bin Packing

1. Order I by increasing size;
2. Remove items of size $< \varepsilon$ to get new list I';
3. Partition I' into $P = \lceil 1/\varepsilon^2 \rceil$ groups;
4. Round the sizes up to the largest size in each group, call this list J';
5. Find the minimal assignment for J' using enumeration;
6. Make this into an assignment for I by assigning the removed items using First Fit;
7. Return this assignment.

12.5 Problems

1. Show that Next Fit is not a k-approximation algorithm for any $k < 2$.
   Find an example for which First Fit (without sorting) has an approximation factor $5/3$.
2. Show that Next Fit is a 2-approximation algorithm for Bin Packing.
3. We call an algorithm for Bin Packing *monotonic* if the number of bins it uses for packing a list of items is always larger than for any sublist.
   Show that Next Fit is monotonic, but First Fit is not.
4. Show that instances of Bin Packing for which $s_i > 1/3$ for all item sizes s_i can be solved exactly using a polynomial algorithm that we saw earlier in the course.
   Show that this is not that useful, because FFD solves these instances exactly as well.

(Solutions on page 94.)

Chapter 13

Solutions

Section 1.4:

1. *Prove carefully that the Forest-Growing Algorithm given in class returns an MST, if it exists.*
   Call a forest *good* if it is contained in an MST. The empty graph that the algorithm starts with is clearly good. We will show inductively that all the forests occuring in the algorithm are good, hence also the last forest. Because the last forest satisfies $|E(F)| = |V(G)| - 1$, it must be a spanning tree, and then it is minimum because it is good.
   Let F be a forest in step 2, and suppose it is good, so contained in an MST T^*. We want to show that $F + e$ is good, if $w(e)$ is minimum in S and $F + e$ does not have a cycle. If $e \in T^*$, then clearly $F + e$ is good, so assume $e \notin T^*$. Then adding e to T^* must create a cycle, which must contain an edge f not in F, such that its endpoints are not in the same component of F (since e does not create a cycle in F, its endpoints are in different components, so the vertices of the cycle cannot all be in the same component).
   This implies that f is still in S: if not, then the algorithm must have previously considered f, but rejected it because it would have created a cycle in a predecessor of F. But this contradicts the fact that its endpoints are in different components of F. Since $f \in S$ the algorithm must have chosen e over f, so $w(f) \geq w(e)$. Then $T^{**} = T^* - f + e$ is a minimum spanning tree containing F and e, which means $F + e$ is good.
   This proves that the final forest is good, hence an MST. To be careful, we should also check that the algorithm is correct when it returns "disconnected". When it does, we must have $|E(F)| < |V(G)| - 1$, so it cannot be a tree, hence is disconnected.

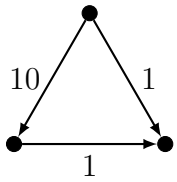
2. *Give a greedy algorithm that finds a minimum spanning tree by removing edges while preserving connectedness. Prove that it works. Give an example to show that the same approach does not work for minimum spanning directed trees.*

> **Pruning Algorithm**
>
> 1. Start with $H = G$ and $S = \emptyset$;
> 2. Pick $e \in E(H) \backslash S$ with maximum $w(e)$; if $E(H) \backslash S = \emptyset$, go to 4;
> 3. If $H - e$ is connected, then remove e from H; else add e to S; go to 2;
> 4. If H is connected, return H; else return "disconnected".

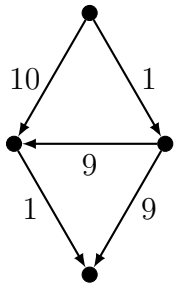
Proof of correctness: If G is not connected, then the algorithm will just put every edge in S and correctly return disconnected. If G is connected, then H will be connected

throughout.
Call a connected subgraph of G *good* if it contains an MST of G. We'll show that the final H is good. Since no edge could be removed without disconnecting it, the final H is minimally connected, so a tree, and spanning because $V(H) = V(G)$ all along. It follows that the final H is an MST.
Suppose we have a good H in step 2, containing an MST T^* of G. If we remove an $e \notin T^*$, then clearly $H - e$ is good, so assume $e \in T^*$. Write $e = uv$. Removing e from T^* must disconnect it into two components, say U containing u and V containing v. Since $H - e$ is connected, there must be a path in it from u to v, which must contain an edge f with one endpoint in U and one in V (and $f \notin T^*$). We cannot have $f \in S$, since then $H - f$ should be disconnected, which it isn't because it contains T^*. So $f \in E(H) \backslash S$, which implies $w(e) \geq w(f)$. Therefore $T^{**} = T^* - e + f$ is an MST contained in $H - e$, so $H - e$ is good.

Directed trees: Take for example



Then removing the 10 would leave the graph connected (in the undirected sense), but there would no longer be a spanning directed tree.
You could avoid this by only removing an edge if afterwards there is still a spanning directed tree, which you can check using for instance breadth-first search. But this also fails:



If you remove the 10, there is still a spanning directed tree, but its weight is 19, whereas there is a spanning directed tree using 10 with weight 12.

3. *Prove that if its weight function w is injective, then a graph has at most one minimum spanning tree.*
Suppose w is injective and there are two distinct MSTs T and T^*. Take $e \in T \backslash T^*$ and consider $T^* + e$. It must contain a cycle, which must contain an edge $f \in T^* \backslash T$. Now $w(e) \neq w(f)$, and we can assume $w(e) < w(f)$ without loss of generality. Then $T^* + e - f$ is a spanning tree with $w(T^* + e - f) < w(T^*)$, contradiction.

4. *Give an algorithm that finds the second-smallest spanning tree in a weighted undirected graph (in other words, given a graph and an MST T, find the smallest among all spanning trees distinct from T). Prove that it works.*
The question should maybe say "*a* second-smallest", since there could be several MSTs, or the MST could be unique, with several second-smallest ones.

Algorithm: For each $e \in T$, find an MST in $G - e$. Compare the weights of the trees that you get, and take the smallest.

It's polynomial, though of course you could implement it more efficiently, not running a complete MST algorithm every time.
That it works follows directly from the following fact:

Fact: *Given an MST T, a second-smallest spanning tree can always be found among the trees of the form $T - e + f$, $e \in T$, $f \notin T$.*
Proof: First assume that T is unique.
Let T be an MST, and T' a second-smallest spanning tree. Take $e \in T \backslash T'$ with minimum $w(e)$. Then adding e to T' creates a cycle, which contains an edge $f \in T' \backslash T$. We claim that $w(e) < w(f)$.
Suppose that $w(f) \leq w(e)$. Add f to T, creating a cycle, which contains an edge $g \in T \backslash T'$. Now we have $w(g) < w(f)$, otherwise $w(T - g + f) \leq w(T)$, contradicting minimality or uniqueness of T. But then $w(g) < w(f) \leq w(e)$, contradicting our choice of e. This proves that $w(e) < w(f)$.
It follows that $w(T' - f + e) < w(T')$, so since T' is second-smallest, we must have $T' - f + e = T$.
Suppose that T is not unique.
Take another MST T' with as many edges in common with T as possible. Take $e \in T \backslash T'$, create a cycle in T', containing $f \in T' \backslash T$. If $w(e) \neq w(f)$, then we'd get a smaller MST, so $w(e) = w(f)$. Then $T' - f + e$ is an MST having fewer edges in common with T, which implies $T' - f + e = T$. Done.

Section 2.5:

1. *Given a weighted directed graph G that may have negative edges but no negative cycle, and given a potential on G, show that a shortest ab-path can be found by changing G in such a way that Dijkstra's algorithm works on it.*

   Let w be the weight function and y a potential, so $y_v - y_u \leq w(uv)$ for any edge uv. Then define a new weight function

   $$w'(uv) = w(uv) - y_v + y_u,$$

   so that all $w'(uv) \geq 0$. Hence we can apply Dijkstra's algorithm to G with w', but we need to prove that the shortest paths are the same as for the original graph. For a path P from a to b we have

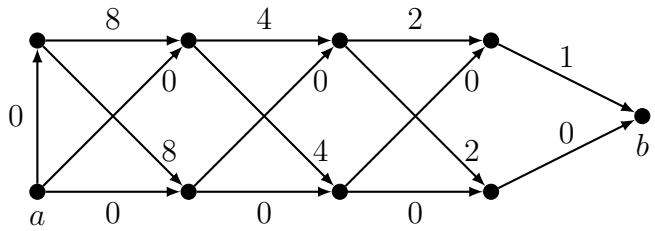
   $$w'(P) = \sum_{uv \in P} w'(uv) = \sum_{uv \in P} w(uv) - y_v + y_u = w(P) + y_b - y_a.$$

   This means that the weight of any ab-path is changed by the same amount, so the shortest ab-paths are the same for w' as for w (note that for different a', b', weights of $a'b'$-paths are changed by different amounts than weights of ab-paths, but that doesn't matter).

2. *Given a sequence $a_1, \ldots, a_n$ of real numbers, we want to find a consecutive subsequence with minimal sum, i.e. $i \leq j$ such that $\sum_{k=i}^{j-1} a_k$ is minimal (the sum $= 0$ when $i = j$). Show that such a sequence can be found using a quick version of Ford's algorithm, where edges are corrected in an order such that each one only needs to be corrected once. Carry this out for the sequence $-1, 3, -2, 1, -1, -1, 5, -2, 3, -2$.*

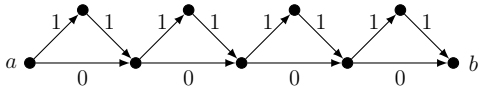
Define a weighted directed graph with vertices $a_1, \ldots, a_n$, with an edge from a_i to a_{i+1} of weight a_i, and with two more vertices: r, with edges from r to every a_i of weight 0; and s, with an edge from every a_i to s of weight 0.
A shortest rs-path in this graph is the same as a minimal subsequence: The path will go from r to some a_i, to some a_j, to s. Then $\sum_{k=i}^{j-1} a_k$ is minimal. So Ford's algorithm can solve the problem.
A single "pass" of Ford (correcting each edge only once) will suffice if it corrects all the ra_i-edges first, then the a_ia_{i+1}-edges in order, then all the a_is-edges.
We won't write it out here, but the minimal subsequence is $-2, 1, -1, -1$ (obviously, you could see this right away, the idea was just to see how this works on a small example).

3. *Consider the following version of Ford's algorithm, which is simpler than the one in the notes: Set $d(a) = 0$, $d(v) = \infty$ for $v \neq a$; then repeatedly pick any edge $uv \in E(G)$ such that $d(v) > d(u) + w(uv)$, set $d(v) = d(u) + w(uv)$; stop when there are no such edges left. Use the graph below, and generalizations of it, to deduce that its running time is not polynomial.*



Let G_k be the graph similar to that above, but with k horizontal edge above and below, with the first upper horizontal edge having weight 2^k; so the one above is G_3. Then "simple Ford" might need 2^k steps on G_k, if it corrects the edges in an especially bad order. Basically, the 0-edges clearly give the shortest paths, but if the algorithm persistently avoids these then it will take long to find the shortest path.
More precisely, first correct the vertical edge from a, giving potential 0 at its endpoint. Then correct along the nonzero edges, giving potentials 8, 12, 14 above and below, and finally potential 15 at b.
Now I claim that we can make b go through every potential value from 15 down to 0. In general, this would mean 2^k values, each after at least one correction step, so $\geq 2^k$ steps. Let u be the upper neighbor of b, and l the lower one. Correcting lb gives b potential 14. Then correcting u and l along their incoming 0-edges, they get potential 12. Correcting ub gives b potential 13, then correcting lb gives b potential 12. Go on like this.
To look at it a very different way, we are writing any number ≤ 15 in binary. For instance, $6 = 0110$, so take the unique path with edges 0, 4, 2, 0 from s to b: correcting along this path ends up with potential 6 for b. And we can do this for 15 down to 0.

This problem set originally had a suggestion that does not work, with an incorrect proof, which we reproduce below for instructive purposes.



\begin{incorrect} Let G_k be the graph similar to that above, but with k triangles; so the one above is G_4. Then "simple Ford" might need 2^k steps on G_k, if it corrects the edges in an especially bad order. Basically, the 0-edges clearly give the shortest paths,

but if the algorithm persistently does the 1-edges first, then it will take long to find the shortest path.

More precisely, we claim that G_k can take twice as many steps as G_{k-1}. Suppose it corrects the two 1-edges from a first, giving a's right-hand neighbor a potential value 2. Suppose that next it does not yet do the 0-edge, but goes on with the rest of the graph, which is basically a G_{k-1}. Then it ends up with potential value of 2 on b. If only then it corrects the 0-edge from a, it changes the potential value of a's right-hand neighbor to 0. Then it has to go through the entire G_{k-1} again, to finally end up with 0 at b.

So G_k can take twice as many steps as G_{k-1}, and since G_1 can take 2 steps, G_k can take 2^k steps in total.

Since G_k has $3k$ edges and $2k+1$ vertices, this means that the running time is not polynomial. `\end{incorrect}`

It goes wrong where it says "go through G_{k-1} again"; this is sort of true, but it doesn't have to redo everything from scratch for G_{k-1}, it does already have estimates of the distances there. So redoing this G_{k-1} does not take as long as the first time.
To see that simple Ford is polynomial on these G_k, look at the sum of all the finite potential values. At most, this is $1 + 2 + \cdots + 2k \leq 4k^2$. At each correction step this is lowered by 1, so there can be at most $4k^2$ correction steps.
This actually shows that to make a family of examples on which simple Ford is not polynomial, you'll have to take some exponentially large weights, like the 2^k in the examples above.

4. *Show that if you had a polynomial algorithm that finds a shortest path in any weighted directed graph (allowing negative cycles), then it would give you a polynomial algorithm to determine if any unweighted directed graph has a Hamilton cycle.*

   Let G be an unweighted directed graph with n vertices, for which we want to determine if it has a Hamilton *path* or not (a path using all vertices; below we will relate this to finding a Hamilton cycle).

   Define a weight function by $w(e) = -1$ for all e. Now run the hypothetical algorithm (unbothered by all the negative cycles) with any two vertices a, b, which returns a shortest ab-path. If the length of this shortest path is $-n+1$, then it must be a Hamilton path; otherwise, it will have length $> -n+1$, and there is no Hamilton path. Hence we would have a polynomial algorithm for Hamilton paths.

   Now suppose we have a polynomial algorithm for Hamilton paths, and we want one for Hamilton cycles. Given G, pick any vertex u, and for any outgoing edge uv, try the following.
   Remove uv and replace it by new vertices u', v', with only edges uu' and $v'v$. Determine if this graph has a Hamilton path; if it does, it must start in v' and end in u', so putting uv back in gives a Hamilton cycle; if this graph has no Hamilton path, then G has no Hamilton cycle using uv.
   If for no uv we find a Hamilton cycle in this way, then there is none, since it would have to use some outgoing edge of u. Hence we have a polynomial algorithm for determining the existence of Hamilton cycles.

Section 3.3:

1. *Show that, given an algorithm for finding a maximum set edge-disjoint paths in any directed st-graph, you can use it to find a maximum set of vertex-disjoint directed st-paths in any directed graph.*
   Show that there is also a way to do this the other way around.

   Let G be the graph in which to find a maximum set of vertex-disjoint paths. Define a new graph G' by splitting up every $v \neq s, t$ into two vertices $v^{\text{in}}, v^{\text{out}}$, with a single edge

between them. More precisely:

$$V(G') = \{s^{\text{out}} = s, t^{\text{in}} = t\} \cup \{v^{\text{in}}, v^{\text{out}} : v \in V(G) \backslash \{s,t\}\},$$

$$E(G') = \{u^{\text{out}} v^{\text{in}} : uv \in E(G)\} \cup \{v^{\text{in}} v^{\text{out}} : v \in V(G) \backslash \{s,t\}\}.$$

Then paths in G' correspond to paths in G (just contract the split-up vertices), and vertex-disjoint paths in G' corresponds to edge-disjoint paths in G. Indeed, edge-disjoint paths in G' cannot use an edge $v^{\text{in}} v^{\text{out}}$ twice, which means that the corresponding paths in G cannot use the vertex v twice.

Let G' be a graph in which to find a maximum set of edge-disjoint paths. We use a similar construction to above, but a bit more complicated.
For a vertex $v \in G'$ with k incoming vertices and l outgoing vertices, replace it by k vertices v_i^{in} and l vertices v_j^{out}, and add all edges $v_i^{\text{in}} v_j^{\text{out}}$. Whenever there is an edge $uv \in E(G')$, put an edge $u_j^{\text{out}} v_i^{\text{in}}$, in such a way that every u_j^{out} has out-degree 1, and every v_i^{in} has in-degree 1.
Given any path in G, you get a path in G' again by collapsing the split-up vertices. And two paths P, Q in G are vertex-disjoint if and only if the corresponding paths P', Q' in G' are edge-disjoint: If P' and Q' have a common edge uv, then P and Q in G have a common edge $u_j^{\text{out}} v_i^{\text{in}}$, so also a common vertex; if P and Q have a common vertex, say v_i^{in}, then since this vertex has in-degree 1, they also share a common edge, hence so do P' and Q'.

2. *Give an integer program whose optimal solutions correspond to maximum sets of vertex-disjoint st-paths, in a given directed graph G with vertices s, t.*
Give the dual of the relaxation of this program. What objects in the graph do integral dual optimal solutions correspond to?

The form in the notes with path-variables won't work here, because we have a condition at every vertex, that only one path may pass through it. But in the form with edge-variables we can add a (second) constraint per vertex. Fortunately, this automatically forces the $x_e \leq 1$ condition, so we can drop that.

LP for Vertex-Disjoint Paths

$$\text{maximize} \sum_{sw \in \delta^{\text{out}}(s)} x_{sw} \quad \text{with } x \geq 0,\ x \in \mathbb{Z}^{|E|},$$

$$\sum_{e \in \delta^{\text{in}}(v)} x_e - \sum_{e \in \delta^{\text{out}}(v)} x_e = 0 \quad \text{for } v \in V \backslash \{s,t\},$$

$$\sum_{e \in \delta^{\text{in}}(v)} x_e \leq 1 \quad \text{for } v \in V.$$

The dual comes out similar to for the flow problem, except that both the y and z are indexed by the vertices.

Dual of relaxation

$$\text{minimize} \sum_{v \in V} z_v, \quad \text{with } y \in \mathbb{R}^{|V|}, z \geq 0,$$

$$y_v - y_u + z_v \geq 0 \quad \text{for } uv \in E, u \neq s, v \neq t,$$

$$y_w + z_w \geq 1 \quad \text{for } sw \in \delta^{\text{out}}(s),$$

$$-y_w + z_t \geq 0 \quad \text{for } wt \in \delta^{\text{in}}(t).$$

Again, we can get a simpler version by putting $y_s = 1$, $y_t = 0$.

Dual of relaxation
minimize $\sum_{v \in V} z_v$, with $y \in \mathbb{R}^{\lvert V \rvert}, z \geq 0$,
$y_v - y_u + z_v \geq 0$ for $uv \in E$,
$y_s = 1, y_t = 0$.

An integral optimal solutions corresponds to a *vertex cut*: a minimum set of vertices such that if you remove them, there is no longer any st-path. The $y_v = 1$ correspond to vertices on the s-side of the cut, $y_v = 0$ ones are on the t-side, the $z_v = 1$ correspond to the vertices of the cut, and the $z_v = 0$ to the others.

3. *Use flows to give an algorithm for the binary assignment problem: Given a bipartite graph G with $c : E(G) \to \mathbb{Z}_{\geq 0}$ and $d : V(G) \to \mathbb{Z}_{\geq 0}$, find a maximum assignment, i.e. a $\varphi : E(G) \to \mathbb{Z}_{\geq 0}$ such that for all edges e we have $\varphi(e) \leq c(e)$ and for all vertices v we have $\sum_{e \in \delta(v)} \varphi(e) \leq d(v)$.*

   Define a network by directing the edges of G from one partite set, call it A, to the other, B. Then add an s and a t, with edge from s to all vertices in A, and from all vertices of B to t. Give the edges in G the capacities c, and give an edge sa the capacity $d(a)$, an edge bt the capacity $d(b)$.
   A flow φ in this network corresponds exactly to an assignment.

4. *A path flow g is a flow such that $g(e) > 0$ only on the edges of one directed st-path.*
   A cycle flow h is a flow such that $h(e) > 0$ only on the edges of one directed cycle.
   Prove that any flow f can be decomposed into path flows and cycle flows, i.e. there is a set $\mathcal{P}$ of path flows and a set $\mathcal{C}$ of cycle flows, with $\lvert\mathcal{P}\rvert + \lvert\mathcal{C}\rvert \leq \lvert E(G)\rvert$, such that

$$f(e) = \sum_{g \in \mathcal{P}} g(e) + \sum_{h \in \mathcal{C}} h(e) \quad \forall e \in E(G).$$

   Remove all edges with $f(e) = 0$. Find any st-path in the remainder, let β be the minimum $f(e)$ on this path, and let g_1 be the flow with value β along this path, 0 everywhere else. Set $f := f - g$, and repeat the above. Every time, at least one edge of the path is removed (the one where $f(e)$ was β), so eventually there are no more paths. The g_i will be the path flows in $\mathcal{P}$.
   The remaining f still satisfies the flow constraint. Take any edge $e = uv$ where $f(e) > 0$, and trace a path from there. This has to end up back at u, so gives a cycle. Again take the minimum $f(e)$ along that cycle, which gives a cycle flow h_1. Remove h_1 and repeat, until there is no edge with $f(e) > 0$. Then the h_j are the cycle flows in $\mathcal{C}$.
   The g_i and h_j clearly decompose f, and since each time at least one edge was removed, then total number is $\leq \lvert E(G)\rvert$.

5. *Use the first graph below to show that if the augmenting path algorithm chooses the path Q arbitrarily, then its running time is not polynomial.*
   Use the second graph below to show that if there are irrational capacities (here $\phi = (\sqrt{5} - 1)/2$, so $\phi^2 = \phi - 1$), then the same algorithm may not terminate. Also show that it may not even converge to the right flow value.

   If the algorithm keeps choosing an augmenting path involving the 1-edge, we will have $\alpha = 1$ every time, and it will take $2K$ iterations until the maximum flow is found. So

it can arbitrarily many steps for a constant number of vertices, which means that the algorithm is not polynomial.

The second is quite tedious to write out, so I will cheat and refer to someone else's writeup, which has nicer pictures anyway. On the course website, go to the notes by Jeff Erickson, and there go to lecture 22, second page.

Section 4.4:

1. *Show that the maximum weight matching problem and the maximum weight perfect matching problem are equivalent, in the sense that if you have a polynomial algorithm for one, then you also have a polynomial algorithm for the other.*

   Given a graph G in which to find a maximum weight perfect matching, let $K = 1 + \sum_{e \in E} w_e$, and define $w'_e = w_e + K$. Now find a maximum weight matching in G with weights w'_e. Because of the size of K, this will be a matching with maximum cardinality, and maximum weight among those with maximum cardinality. If there is a perfect matching, this will be one.

   Given a graph G in which to find a maximum weight matching, define a new graph as follows. Make a copy G' of G, but set all its weights to 0. Then make a new graph H consisting of G and G', with also an edge between any $v \in V(G)$ and its copy $v' \in V(G')$; give such an edge weight 0. Find a maximum weight perfect matching in H, which certainly exists because the edges vv' form a perfect matching (albeit of weight 0). This maximum weight perfect matching will consist of a maximum weight matching in G, together with other edges of weight 0.

2. *Show in two ways that if a bipartite graph is k-regular (every vertex has degree $k \geq 1$), then it has a perfect matching: once using linear programming, and once using König's Theorem.*

   - First note that $|A| = |B|$, since

   $$|E| = k|A| = \sum_{a \in A} \deg(a) = \sum_{b \in B} \deg(b) = k|B|.$$

   - Using König: We observe that any vertex cover C has at least $|A|$ vertices, since each vertex of C covers exactly k of the $k|A|$ edges. Then König tells us that a maximum matching has at least $|A|$ edges, which is only possible if it has exactly $|A| = |B|$ edges, so is perfect.

   - Using LP: Summing up all the dual constraints $y_u + y_v \geq 1$ gives $\sum_{uv \in E}(y_u + y_v) \geq |E|$. Then

   $$k \cdot |A| = |E| \leq \sum_{uv \in E} (y_u + y_v) = k \cdot \sum_{v \in V} y_v,$$

   where the last equality holds because each y_v occurs in k different uv.
   So we have $\sum_{v \in V} y_v \geq |A|$, which implies that the dual minimum is $\geq |A|$, hence so is the primal maximum. There must be an integral optimum primal solution by the theorem from the lecture, which must have $x_e \in \{0, 1\}$, so that will correspond to a perfect matching.

3. *Prove complementary slackness for bipartite maximum weight perfect matchings (Lemma 4.4 in the notes): If we have a primal feasible x and a dual feasible y such that for all $e \in E(G)$ either $x_e = 0$ or $y_a + y_b = w_{ab}$, then both are optimal.*

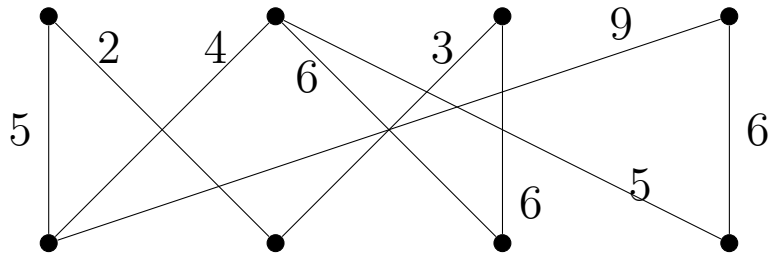
By duality we have

$$\sum_{e\in E} w_e x_e \leq \sum_{v\in V} y_v$$

for any primal feasible x and dual feasible y. We show that the complementary slackness property gives equality, which implies that in that case x and y are optimal.
For every edge $e = uv$ we have $w_e x_e = (y_u + y_v)x_e$, so

$$\sum_{e\in E} w_e x_e = \sum_{uv\in E} (y_u + y_v)x_{uv} = \sum_{v\in V} \left(\sum_{u:uv\in E} x_{uv} \right) y_v = \sum_{v\in V} y_v.$$

4. *Execute the primal-dual algorithm for maximum weight perfect matchings for the graph below. Give proof that the resulting matching is indeed maximum.*
You can write it out however you like, but make sure that the different objects in every step are clear.
This it too annoying to latex. The maximum weight is 22, uniquely achieved by the matching with weights 2,5,6,9.



5. *Let G be a graph, not necessarily bipartite. Prove that a matching M in G is maximum if and only if there is no augmenting path for M. (a path between two unmatched vertices that alternates M-edges and non-M-edges).*
So why can't we use the augmenting path algorithm for nonbipartite graphs?

If M is maximum there cannot be an augmenting path for M, because then there would be a larger matching.
If M is not maximum, let M' be a larger matching, so $|M'| > |M|$. Consider H with $V(H) = V(G)$ and $E(H) = E(M) \cup E(M')$. Its vertices can only have degree 0,1, or 2, which means it consists of paths and cycles (we've seen the argument/algorithm for that before). These paths and cycles must be alternating (i.e. alternately use edges from M and M'), otherwise one of the matchings would have two touching edges. Alternating cycles have as many M-edges as M'-edges, so there must be a path with more M'-edges than M-edges, which means that it is augmenting for M. Done.

The augmenting path algorithm doesn't work like this for nonbipartite graphs, because it's not clear how to find an augmenting path. We can't use the same directed graph trick, because there is no given direction that you could orient the matching edges in (like "from B to A").
The problem is that without this trick we don't have an algorithm for finding alternating paths from an unmatched vertex to another unmatched one. You could just try all alternating paths from a vertex, maybe with depth-first search, but this could take exponential time. The answer to this will come in the next lecture.

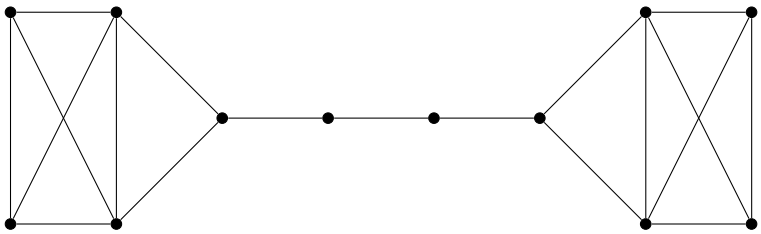
Section 5.4:

1. *Consider the following greedy algorithm for matchings:*

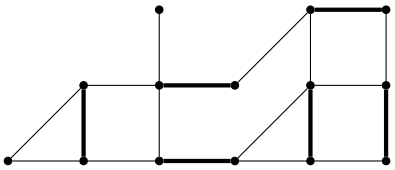
   Repeatedly add uv such that $\deg(u) + \deg(v)$ is minimal, then remove $\delta(u) \cup \delta(v)$.

   Give an example for which this algorithm does cannot return a maximum cardinality matching.

   This is the simplest example I could come up with. All edges have $d(uv) \geq 3$, except for the one in the middle, which has $d(uv) = 2$. After removing that one, and its two touching edges, we'd be left with two odd graphs, which we can't match perfectly. But we can match the whole graph perfectly if we avoid that middle edge.



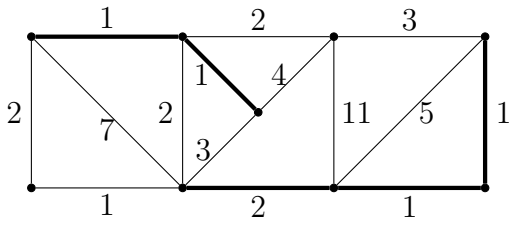
2. *Execute the blossom algorithm to find a maximum cardinality matching for the following example, starting with the given matching.*



   No way I'm doing that in LaTeX.
   I have set it up so that if you really take a shortest path in every step, then you will need 2 or 3 blossom shrinkings.

3. *Execute the postman algorithm to find a minimum postman tour in the following example (find the minimum weight perfect matching by inspection).*

   There are 4 odd-degree vertices. Compute all 6 shortest paths between those (just by inspection...), take the minimum weight perfect matching (by inspection), and for those paths double the edges (they're thicker in the picture here). Now trace an Euler tour, using the thick edges twice (by trial and error).
   The weight of the tour is the sum of all the weights, and then the double ones added. I think that's 52.



4. *Show that Dijkstra's algorithm can be used to find a shortest st-path in an undirected weighted graph, if all weights are nonnegative. Why does this fail when there are negative weights?*
   Give an algorithm for shortest st-paths in an undirected graph G with arbitrary weights,

but without negative cycles.
~~*(Hint: Find a minimum weight perfect matching in a graph constructed from two copies of G.)*~~

- **Dijkstra works for nonnegative weights:** Given an undirected graph G, define a directed graph D_G with $V(D_G) = V(G)$ and

$$D_G = \{ab, ba : \{a, b\} \in E(G)\}.$$

Define weights w' on D_G by $w'(ab) = w(\{a, b\})$. So we replace an edge $\{a, b\}$ by two directed edges ab and ba, both with weight $w(\{a, b\})$.
Then a shortest st-path in D_G, found by Dijkstra, will directly give a shortest st-path in G.

- **Dijkstra fails for negative weights:** When there are negative weights, D_G will have negative cycles. Indeed, if $w(\{a, b\}) < 0$, then aba is a cycle in D_G with weight $w'(aba) = 2w(\{a, b\}) < 0$. So Dijkstra's or Ford's algorithm will not work here.

- **Negative weights without negative cycles**

Note: *The solution to the third part that I originally had in mind does not quite work, and so the hint was not so helpful. Ask me if you want to know what was wrong with it. The construction below does sort of consist of two copies, but with more structure.*

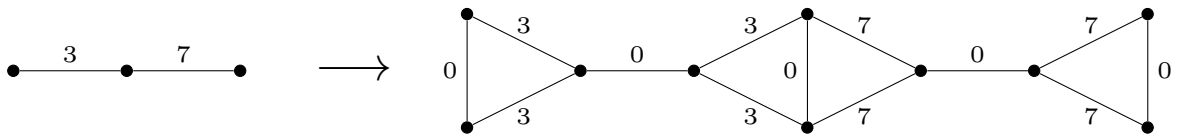
- We define a new graph G' from G as follows.
We leave s and t as they are. For every other vertex v of G, we take two copies v_1 and v_2, with an edge v_1v_2 of weight $w'(v_1v_2) = 0$.
For every edge uv in G, we add the following: Two vertices x_{uv} and y_{uv}, with edges

$$u_1x_{uv}, \quad u_2x_{uv}, \quad x_{uv}y_{uv}, \quad y_{uv}v_1, \quad \text{and} \quad y_{uv}v_2,$$

that have weights

$$w'(u_1x_{uv}) = w'(u_2x_{uv}) = w(uv), \quad w'(x_{uv}y_{uv}) = 0, \quad w'(y_{uv}v_1) = w'(y_{uv}v_2) = w(uv).$$

For an edge sv we add vertices x_{sv} and y_{sv}, with edges $sx_{sv}, x_{sv}y_{sv}, y_{sv}v_1, y_{sv}v_2$, and weights $w'(x_{sv}y_{sv}) = 0, w'(sx_{sv}) = w'(y_{sv}v_1) = w'(y_{sv}v_2) = w(sv)$. For edges tv we do the analogous thing. This completes G'. Here's what it looks like for two edges (not involving s or t, and leaving out vertex labels):



Now we claim that a minimum weight perfect matching M in G' will give a shortest st-path P in G, with $w'(M) = 2w(P)$.

Given a minimum weight perfect matching M in G', we construct a path P as follows. One of the edges sx_{sv} must be in M. Then for that v the edge $x_{sv}y_{sv}$ is not in M, so one of the edges $y_{sv}v_1$ and $y_{sv}v_2$ must be in M. Then v_1v_2 is not in M, and for some neighbor u of v one of the edges v_1x_{vu} and v_2x_{vu} must be in M. It is a bit tricky to describe, and a picture would do miracles, but this must go on like this, and the only way it can finish

is at t. So we have an alternating path from s to t. Then all the original vertices v that we used copies of in this path together form an st-path P. Observe that the total weight of the matching edges in the alternating path in M equals $2w(P)$.
Now consider any edge M not on this alternating path. If it is not of type v_1v_2 or $x_{uv}y_{uv}$, then it will force an alternating path in the same way, but now it can only finish as an alternating cycle, and it similarly gives a cycle in G. Since there are no negative cycles, the total weight of this cycle is ≥ 0. But then it must be $= 0$, because the matching is minimum, and we can replace the cycle by edges of type v_1v_2 or $x_{uv}y_{uv}$, which have weight 0.
So this shows that because M is minimum, the total weight of edges of M outside the alternating path is 0, which means that $w'(M) = 2w(P)$.

We are done if every st-path P in G comes from a matching M in this way, with $w'(M) = 2w(P)$. But this is pretty obvious, by inverting the construction above: The path gives an alternating path from s to t, and every other vertex we match by edges of type v_1v_2 or $x_{uv}y_{uv}$.
So if M is minimum, then P must be shortest, because if there were a shorter path, then it would give a smaller matching.

Section 7.4:

1. *Show that any forest matroid is also a linear matroid. Also show that the uniform matroid $U_{2,4}$ is a linear matroid, but not a forest matroid.*

   Given G, let A be the following matrix: rows indexed by vertices $v_1, \ldots, v_n$, columns indexed by edges, column v_iv_j with $i < j$ has a 1 in row v_i, a -1 in row v_j, and a 0 else (A is not quite the incidence matrix, but the incidence matrix of some directed graph obtained from G; it doesn't actually matter how the edges are directed).
   Then the linear matroid defined by A is the same as the forest matroid defined by G. To see this, we'll show that a forest in the graph corresponds exactly to an independent set of columns of A.
   If $e_1e_2 \cdots e_k$ is a cycle, then the corresponding columns, call them $c(e_i)$, are linearly dependent, because we can write $\sum_{i=1}^{n} \epsilon_i c(e_i) = 0$, for some choice of $\epsilon_i \in \{1, -1\}$. A non-forest in G contains a cycle, hence the corresponding set of columns is linearly dependent.
   On the other hand, suppose the set of columns corresponding to a forest is dependent, i.e. there is a linear relation $\sum a_i c(e_i) = 0$ between them. We can assume that each a_i is nonzero (otherwise we remove that edge, and we still have a forest). But any nontrivial forest must have a vertex of degree 1 (a leaf), so also an edge e_i which is the only one with a nonzero entry in the row corresponding to the leaf. But then $\sum a_i c(e_i)$ will have a nonzero coefficient in that row, a contradiction.

   Here is a matrix whose linear matroid is exactly $U_{2,4}$:

$$A = \begin{pmatrix} 1 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 \end{pmatrix}.$$

   One can easily check that the linearly independent subsets of columns are exactly the subsets with ≤ 2 elements.

   Suppose $U_{2,4}$ is a forest matroid. Then the 4 1-element subsets correspond to 4 edges of the graph. The 3-element subsets are 3 edges that are not a forest, which means they

must form a 3-cycle. Take one such 3-cycle; then the fourth edge must also form a 3-cycle with every 2 edges of the first 3-cycle. This is clearly not possible.

2. *Let A be the matrix*

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

Let M be the corresponding linear matroid over $\mathbb{F}_2$, the field with 2 elements. Show that M is not a forest matroid, and not a linear matroid over $\mathbb{R}$. (M is called the Fano matroid.)
Label the columns 1 through 7.

Suppose the matroid is a forest matroid. Since $\{1,2,4\}$ is dependent, the corresponding edges form a 3-cycle. So does $\{1,3,5\}$, and its 3-cycle shares an edge with the first 3-cycle. And $\{2,3,6\}$ is also a 3-cycle, so 6 must connect the 2 vertices that the previous 2 3-cycles did not share. In other words, these 6 edges must form a K_4, a complete graph on 4 vertices. In particular, $\{1,2,3\}$ forms a tree of 3 edges with a common endpoint. But then edge 7 has nowhere to go: It would have to form a forest with any 2 of $1,2,3$, but a non-forest with all 3 of them. This is impossible. (Note that we haven't used $\mathbb{F}_2$, this would have worked for this matrix over any field.)

Suppose the matroid is linear over $\mathbb{R}$, so there is a set of 7 columns in some $\mathbb{R}^m$ with the same linear dependencies, and $m \geq 3$. Then 3 of these columns correspond to columns 1,2,3 of A. We can apply a linear transformation, which preserves the matroid, so that these are given by $c_1 = (1,0,0,\ldots)^T, c_2 = (0,1,0,\ldots)^T, c_3 = (0,0,1,\ldots)^T$. Then it follows that the columns corresponding to 4,5,6 of A are given by $c_4 = (1,1,0,\ldots)^T, c_5 = (1,0,1,\ldots)^T, c_6 = (0,1,1,\ldots)^T$; for instance, the relation $1+2=4$ in A implies $c_4 = c_1+c_2$. All we've done here is shown that after a linear transformation the matrix over $\mathbb{R}$ should be pretty similar to A.
But over $\mathbb{F}_2$, the columns $4,5,6$ are dependent, because

$$\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1+1 \\ 1+1 \\ 1+1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Yet over $\mathbb{R}$ the columns c_4, c_5, c_6 are independent, a contradiction.

3. *In a matroid $M = (X, \mathcal{I})$, a circuit is a minimal dependent set, i.e. a C such that $C \notin \mathcal{I}$, but if $D \subsetneq C$ then $D \in \mathcal{I}$.*
Circuits are the generalization of cycles in graphs. The following two facts are the generalizations of standard facts about cycles.

   a) *Prove that if $C_1 \neq C_2$ are two circuits, and $x \in C_1 \cap C_2$, then there is a circuit $C_3 \subset (C_1 \cup C_2) \backslash \{x\}$.*

   b) *Prove that if $Y \in \mathcal{I}$, but $Y \cup \{x\} \notin \mathcal{I}$, then $Y \cup \{x\}$ contains exactly one circuit.*

a) Let $C_1 \neq C_2$ with $x \in C_1 \cap C_2$. There is an $x_1 \in C_1 \backslash C_2$, since C_2 is minimal; then also $x_1 \neq x$. Then there is a basis B of $C_1 \cup C_2$ such that

$$C_1 \backslash \{x_1\} \subset B \subsetneq C_1 \cup C_2,$$

since $C_1\backslash\{x_1\} \in \mathcal{I}$ but $C_1 \cup C_2 \notin \mathcal{I}$. It follows that $x_1 \notin B$, and there is an $x_2 \in C_2$ such that $x_2 \notin B$ and $x_2 \neq x_1$. Then

$$|B| \leq |(C_1 \cup C_2)\backslash\{x_1, x_2\}| < |(C_1 \cup C_2)\backslash\{x\}|,$$

so $(C_1 \cup C_2)\backslash\{x\} \notin I$, since it is larger than B, so also larger than any basis of $C_1 \cup C_2$. Hence there must be a circuit in $(C_1 \cup C_2)\backslash\{x\}$.

b) Since $Y \cup \{x\} \notin \mathcal{I}$, there must be some circuit in $Y \cup \{x\}$. Suppose there are two circuits $C_1 \neq C_2$. Then $x \in C_1 \cap C_2$, otherwise one of the circuits would be in Y. So by a) we have that there is a circuit $C_3 \subset (C_1 \cup C_2)\backslash\{x\}$. But this contradicts the minimality of C_1 (and C_2).

4. *Describe a greedy removal algorithm for maximum weight indepent sets in a matroid, which starts with X and greedily removes elements. Prove that it works.*

We'll assume that the weights are positive; the algorithm could just remove all nonpositive elements first.
To compare, note that in a greedy removal algorithm for maximum weight forests in graphs, you would remove an edge only if the result still contained a maximal forest, because you know that a maximum weight forest would also be maximal (assuming the weights are positive). For a matroid, this corresponds to checking if the result of a removal still contains a basis.

Greedy removal algorithm

(1) Set $Y = X$ and $S = \emptyset$;
(2) Find $x \in Y\backslash S$ with minimum $w(x)$; if $Y\backslash S = \emptyset$ go to 4;
(3) If $Y\backslash\{x\}$ contains a basis of X, set $Y := Y\backslash\{x\}$;
set $S := S \cup \{x\}$; go to 2;
(4) Return Y.

Proof of correctness: If $Y \subset X$ contains a basis, call it *good* if it contains a maximum weight basis of X. We'll show that the final Y in the algorithm is good, which implies that it is a basis (otherwise more elements could be removed), so independent, and that it is maximum weight.
So suppose Y is good, so it contains some maximum weight basis B of X, and x is chosen by the algorithm, so $Y\backslash\{x\}$ contains a basis B' of X. By M2′ we have $|B| = |B'|$.
If $x \notin B$, then $B \subset Y\backslash\{x\}$, and we are done. So assume $x \in B$; we also know that $x \notin B'$. Then $|B\backslash\{x\}| < |B'|$ by M2′, so by M2 there is a $y \in B'\backslash B$ such that $B'' = (B\backslash\{x\}) \cup y \in \mathcal{I}$. This implies that B'' is a basis, since $|B''| = |B|$.
Since y is still in Y, it either hasn't been considered yet, or it has already been rejected. If it had been rejected, a $Y' \supset Y$ would have occurred earlier in the algorithm, such that $Y'\backslash\{y\}$ does not contain a basis; but $B \subset Y\backslash\{y\} \subset Y'\backslash\{y\}$. So y has not been considered yet, hence $w(y) \geq w(x)$. Then $w(B'') \geq w(B)$, which implies that B'' is also maximum, so $Y\backslash\{x\}$ is good.

5. *Let $X_1, \ldots, X_m$ be a partition of X. Define*

$$\mathcal{I} = \{Y \subset X : \forall i \ \ |Y \cap X_i| \leq 1\}.$$

Prove that $(X, \mathcal{I})$ is a matroid. Such a matroid is called a partition matroid.
Let $G = (A \cup B, E)$ be a bipartite graph. Show that the set of matchings of G is an intersection of two partition matroids with $X = E$, i.e. $\{matchings\} = \mathcal{I}_1 \cap \mathcal{I}_2$.

M1 is obvious. For M2′, observe that the bases are exactly the $B \subset X$ with all $|B \cap X_i| = 1$. Since the X_i are a partition, all bases have $|B| = m$.

Let $A = \{a_1, \ldots, a_k\}$ and $B = \{b_1, \ldots, b_l\}$. Partition E in 2 ways:

$$E = \bigcup \delta(a_i), \quad E = \bigcup \delta(b_i).$$

Let

$$\mathcal{I}_1 = \{D \subset E : \forall i,\ |D \cap \delta(a_i)| \leq 1\},$$
$$\mathcal{I}_2 = \{D \subset E : \forall i,\ |D \cap \delta(b_i)| \leq 1\}.$$

Then

$$\mathcal{I}_1 \cap \mathcal{I}_2 = \{D \subset E : \forall v \in A \cup B,\ |D \cap \delta(v)| \leq 1\} = \{ \text{ matchings } \}.$$

Note that this is not a partition matroid, since the $\delta(v)$ for all v do not form a partition. In fact, in the notes we saw that the set of matchings cannot be a matroid.

Section 8.3:

1. *Show that the problem of finding a Hamilton path from s to t in a given directed graph D can be solved using an intersection of 3 matroids.*

   First remove all edges in $\delta^{\text{in}}(s)$ and $\delta^{\text{out}}(t)$ from D.
   We take the following three matroids with $X = E(D)$:

   $$\mathcal{M}_1 = P(\{\delta^{\text{in}}(v)\}),$$
   $$\mathcal{M}_2 = P(\{\delta^{\text{out}}(v)\}),$$
   $$\mathcal{M}_3 = F(G).$$

   An edge set that is independent with respect to $\mathcal{M}_1$ and $\mathcal{M}_2$ has at each vertex at most one incoming edge and at most one outgoing edge. Hence it consists of cycles and paths. If it is also independent with respect to $\mathcal{M}_3$, then there are no cycles, so it is a disjoint union of paths.
   We claim that D has a Hamilton path if and only if a maximum independent set in this intersection of 3 matroids has $|V(D)| - 1$ edges.
   Let I be a maximum common independent set. If it is a spanning tree, then it must be one single path, which must then be a Hamilton path from s to t (because we removed those edges at the start).
   On the other hand, if I is not spanning tree, then it has at most $|V(D)| - 2$ edges. That means there is no Hamilton cycle, because that would be a common independent set with $|V(D)| - 1$ edges.

2. *Given an undirected graph $G = (V, E)$, an orientation is a directed graph $D = (V, E')$ with a bijection $\varphi : E' \to E$ such that $\varphi(ab) = \{a, b\}$. In other words, each edge $\{a, b\} \in E$ is given a direction, either ab or ba.*
   Given $k : V \to \mathbb{N}$, show that the problem of finding an orientation such that

   $$\delta^{in}(v) = k(v)$$

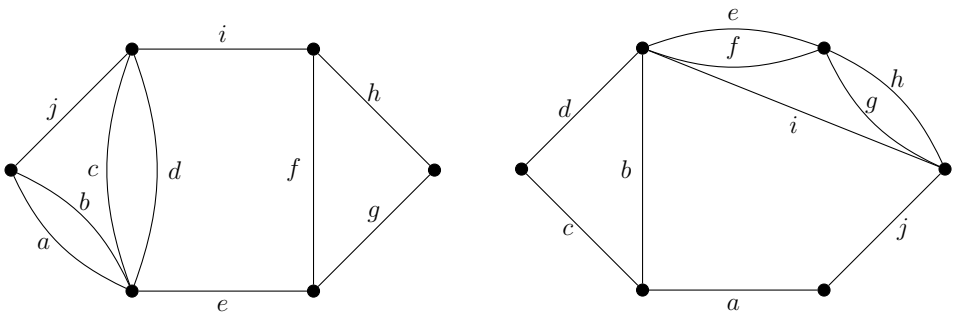
for each $v \in V$, or showing that none exists, can be solved using the matroid intersection algorithm.

We just have to give two matroids such that a common independent set is an orientation satisfying that condition. Let X be the set of directed edges ab and ba for each edge $\{a, b\} \in E(G)$. Then the matroids are

$$\mathcal{M}_1 = P(\{uv, vu\}_{\{u,v\} \in E(G)}),$$
$$\mathcal{M}_2 = \{Y \subset X : |Y \cap \delta^{\text{in}}(v)| \leq k \ \forall v \in V(G)\}$$

That the second one is a matroid is proved just like for partition matroids.
A common independent set would be an orientation because of the first matroid (one direction per edge), and it would satisfy $\delta^{\text{in}}(v) \leq k(v)$ for all v due to the second matroid. Then an orientation as required exists if and only if there is a common independent set of size $\sum_{v \in V(G)} k(v)$.

3. *Use the matroid intersection algorithm to show that there is no simultaneous spanning tree in the following two graphs (i.e., there is no $T \subset \{a, b, \ldots, j\}$ that is a spanning tree in both).*



If we do the greedy part of the algorithm in alphabetical order, then it would find the common independent set $\{a, c, e, g\}$ (if we used a different order, we would get something "isomorphic"). It is not a spanning tree in either graph. If we now draw the directed graph as in the matroid intersection algorithm, we'll see that there is no path from $X_1 = \{f, h, i\}$ to $X_2 = \{b, d, j\}$.
Alternatively, we could observe that

$$U = \{a, b, c, d, j\}$$

(the set of vertices in D_I from which X_2 can be reached) gives equality in $|I| \leq r_{M_1}(U) + r_{M_2}(X - U)$, which implies that I is maximal (see the matroid intersection theorem).

4. *Make up 2 matroids such that the matroid intersection algorithm needs at least 2 non-greedy steps (i.e. with $|Q| > 1$) to get a maximum common independent set.*

Section 9.3:

1. *Show that the following two optimization problems are $\mathcal{NP}$-hard:*

> INDEPENDENT SET: Given an undirected graph G, find a maximum cardinality independent set, i.e. a set $I \subset V(G)$ such that $E(I) = \{uv \in E(G) : u, v \in I\} = \emptyset$.

CLIQUE: Given an undirected graph G, find a maximum cardinality clique, i.e. a $K \subset V(G)$ such that $uv \in E(G)$ for all $u, v \in K$.

We first reduce VERTEX COVER to INDEPENDENT SET, which proves that INDEPENDENT SET is $\mathcal{NP}$-hard. Given G in which to find a minimum vertex cover, consider G as an instance of INDEPENDENT SET.
If I is an independent set, then $C = V(G) \backslash I$ is a vertex cover. Indeed, if $e \in E(G)$, then one of its endpoints is not in I, so it is in C, hence it covers e. And $|C| = |V(G)| - |I|$, so C is minimum if and only if I is maximum.

Next we reduce INDEPENDENT SET to CLIQUE. Given G in which to find a maximum independent set, we consider its complement $\overline{G}$ as an instance of CLIQUE. Recall that $V(\overline{G}) = V(G)$ and

$$E(\overline{G}) = \{uv : u, v \in V(G), uv \notin E(G)\}.$$

Now if $K \subset V(G)$ is a clique in $\overline{G}$, then K is an independent set in G. So clearly maximum cliques in $\overline{G}$ correspond to maximum independent sets in G.

Note that in both cases, the reduction can easily be reversed, so in fact these 3 problems are what is called *polynomially equivalent*.

2. *Show that the following optimization problem is $\mathcal{NP}$-hard:*

LONGEST PATH: Given a directed graph G with weights $w : E(G) \to \mathbb{R}$, and $s, t \in V(G)$, find a directed path from s to t of maximum weight.

We will reduce SHORTEST PATH (find a shortest path in a weighted directed graph between two given vertices, allowing negative cycles) to LONGEST PATH. In Problem Set 3, we already saw that HAMILTON CYCLE reduces to SHORTEST PATH, so this implies that LONGEST PATH is $\mathcal{NP}$-hard.
Given G, w for which to find a shortest path, define w' by $w'(e) = -w(e)$, and consider G, w' as an instance of LONGEST PATH. Clearly, a longest path for w' corresponds exactly to a shortest path for w.

3. *Show that the following optimization problem is $\mathcal{NP}$-hard:*

INTEGER PROGRAMMING: Given a matrix $A \in \mathbb{Z}^{m \times n}$ and vectors $b \in \mathbb{Z}^m, c \in \mathbb{Z}^m$, find a vector $x \in \mathbb{Z}^n$ such that $Ax \leq b$ and cx is maximum, if possible.

Probably the easiest way for us to do this is to reduce INDEPENDENT SET to INTEGER PROGRAMMING. Given a graph G in which to find a maximum independent set, we take the integer program that maximizes $\sum_{v \in V(G)} x_v$ subject to $x_v \in \mathbb{Z}, 0 \leq x_v \leq 1$ for all $v \in V(G)$ and $x_u + x_v \leq 1$ for all $uv \in E(G)$. This works because a vertex set is independent if and only if it contains at most one endpoint of every edge.
More precisely, we take $n = |V(G)|$, $m = |E(G)| + 2|V(G)|$, $c = \overline{1}$, and

$$A = \begin{bmatrix} M^T \\ I \\ -I \end{bmatrix}, \quad b = \begin{bmatrix} \overline{1} \\ \overline{1} \\ \overline{0} \end{bmatrix},$$

where M is the incidence matrix of G.

4. *Show that the following optimization problem is $\mathcal{NP}$-hard:*

> METRIC TSP: Let G be a complete undirected graph G with a weight function $d : E(G) \to \mathbb{R}_{>0}$ that satisfies the triangle inequality
>
> $$d(uw) \leq d(uv) + d(vw)$$
>
> for all $u, v, w \in V(G)$.
> Find a minimum weight Hamilton cycle in G.

We reduce HAMILTON CYCLE to METRIC TSP by almost the same construction that we used in the notes, but we give the edges weights 1 and 2 instead of 1 and 10.
More precisely, given G in which to find a Hamilton cycle, give all its edges weight 1, then add edge between any two unconnected vertices, and give these new edges weight 2. Call this new graph G'.
Then the minimum weight Hamilton cycle in G' has weight $|V(G)|$ if and only if it is also a Hamilton Cycle in G. And G' is and instance of METRIC TSP, since its weights satisfy the triangle inequality, because of the deep theorem $2 \leq 1 + 1$.

Section 10.5:

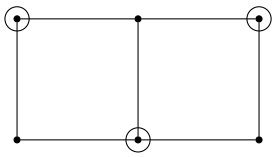
1. *Show that for* VERTEX COVER *the greedy approach (repeatedly add a vertex if the result is still a vertex cover) does not give a k-approximation for any k.*
Show by an example that the Cover-from-Matching algorithm for VERTEX COVER *is not a k-approximation algorithm for any $k < 2$.*

For the first part, consider star-graphs S_n, which consist of 1 vertex of degree $n - 1$ and $n - 1$ vertices of degree 1, all connected to the first one.
On such a graph, if the greedy approach does not take the high-degree vertex first, it will be forced to take all the degree-1 vertices. But the minimum cover consists of just the high-degree vertex.
So on S_n the greedy algorithm would have an approximation factor of $|C|/|C^*| = n - 1$, so it cannot be a k-approximation algorithm for any k.

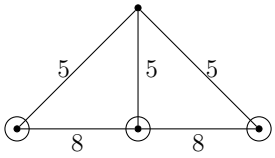
For the second part, take the graph consisting of n disjoint edges. The minimum vertex cover has n vertices, one from each edge, but the Cover-from-Matching algorithm would take all of the $2n$ vertices. So its approximation factor on these examples is $2n/n = 2$.

2. *Give an example where the* STEINER TREE *approximation algorithm does not give a minimum Steiner Tree.*
Show that it is not a k-approximation algorithm for any $k < 2$.

In the following example, the minimum Steiner has 3 edges, using only the top-middle vertex. But the algorithm could end up with a 4-edge Steiner tree, for instance using the bottom-left and bottom-right vertices.
Basically, the tree always consists of 2 shortest paths of two edges, but in the minimum case these overlap in one edge. The algorithm does not see this distinction.

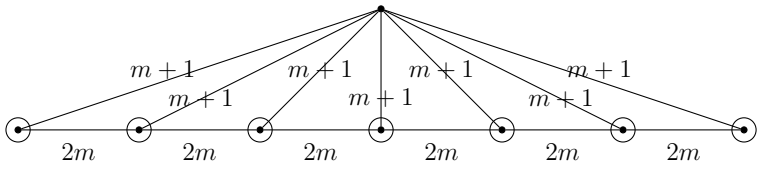


But this example is not quite great, because the algorithm could still find the minimum one. Here is an example for which the algorithm is guaranteed to fail. It is weighted, but one could make it unweighted by replacing an edge of weight w by a path of w edges.



The minimum Steiner tree has weight 15, but the algorithm will give one of weight 16.

This example we can generalize to get ones with approximation factor arbitrarily close to 2, like this:



Let k be the number of vertices in such a graph. Then the minimum Steiner Tree has weight $(k-1)(m+1)$, while the algorithm will find one of weight $(k-2)\cdot 2m$. So the approximation factor for k vertices will be

$$\frac{(k-2)\cdot 2m}{(k-1)(m+1)} = 2\cdot\frac{m}{m+1}\cdot\frac{k-2}{k-1},$$

which is < 2 but can be made arbitrarily close to 2 for k and m large enough.

3. *The* Metric Steiner Tree *problem is the special case of* Steiner Tree *where the graph is complete and the weights d are ≥ 0 and satisfy the triangle inequality*

$$d(u,w) \leq d(u,v) + d(v,w).$$

Show that any algorithm that simply returns a minimum spanning tree of the special set S is a 2-approximation algorithm for Metric Steiner Tree.

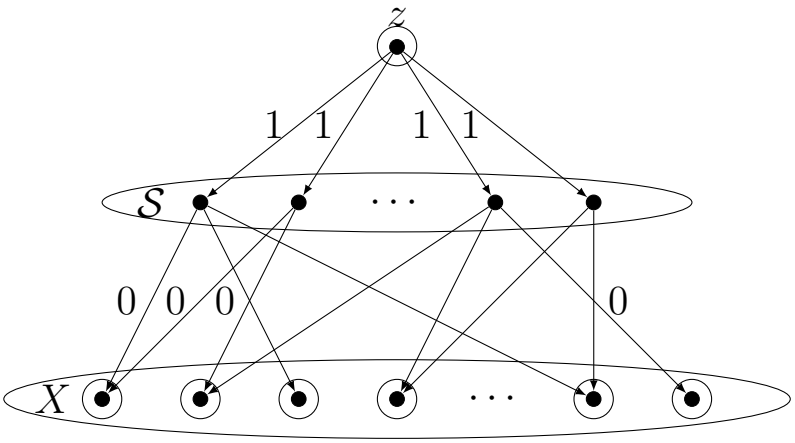
Given a Steiner Tree T^*, we show that there is a spanning tree T of weight $d(T) \leq 2d(T^*)$. We double the edges of T^* and let E be an Euler tour through this graph, so $d(E) = 2d(T^*)$. Let $S = \{s_1, \ldots, s_m\}$, in the order in which the s_i appear in E. Let T be the path $s_1 s_2 \cdots s_m$.
If we write Q_i for the subpath of E between s_i and s_{i+1}, then we crucially have $d(s_i s_{i+1}) \leq d(Q_i)$ by (repeated applications of) the triangle inequality. So

$$w(T) \leq \sum d(s_i s_{i+1}) \leq \sum Q_i \leq d(E).$$

Remark: Note that the Steiner Tree algorithm works by constructing a metric graph on S from the non-metric graph it is given, by taking the complete graph on S with weights the distances between them. Since these distances satisfy the triangle inequality, this simple Metric Steiner Tree algorithm can be applied, and the result can be converted back into a tree in the original graph.

4. *Show that* Set Cover *reduces to* Directed Steiner Tree *(which is like* Steiner Tree*, but the graph is directed, and the tree should be directed).*

   Given a Set Cover instance $X, \mathcal{S}, c$, we construct a directed graph like in the picture below. We take a vertex for each set S_j and also for each element x_i. Then we put an edge of weight 0 from S_j to x_i if $x_i \in S_j$. We also put an extra vertex z with edges of weight 1 to each S_j. The set S for the Steiner tree consists of z and all x_i.
   Now one can check that a minimum Steiner Tree for S corresponds exactly to a minimum set cover.
   Note that this implies that there is no k-approximation algorithm for Directed Steiner Tree for any k. The same construction does not work with undirected Steiner Tree, because then one could 'cheat' by including an S_j without its weight-1 edge to z.



Section 11.4:

1. *Give tight examples for the Double-Tree algorithm and Christofides' algorithm (i.e. give examples which show that these are not k-approximation algorithms for any $k < 2$ and $k < 3/2$, respectively).*

   Double-Tree: In a complete graph G on n vertices, consider a subgraph H consisting of an $n-1$-cycle, and the remaining vertex with edges to each of the $n-1$ vertices on the cycle (so H is a union of a cycle and a star). Give the edges of H weight 1, and give all other edges of G weight 2.
   There is a Hamilton cycle in H of weight n, which is clearly minimum. But the Double-Tree algorithm may end up with a Hamilton cycle of weight $2n-2$, which proves the statement.
   Indeed, one of the MSTs of G is the star from H. If the Euler tour goes through the doubled star in the wrong way, not visiting after another any of the vertices that are adjacent on the cycle (you might want to draw a picture here), then shortcutting will give a Hamilton cycle that uses $n-2$ edges of weight 2, and 2 of weight 1.

   Christofides: We will force the algorithm to pick a bad edge, by designing the graph so that there is an MST which is a path whose endpoints are the vertices of that bad edge. Then the bad edge is the only matching of the odd vertices on the path, so Christofides will add the bad edge. At the same time we make sure there is a shorter Hamilton cycle which avoids the bad edge. Actually, for the construction it is a bit simpler to start with the cycle and then add the path.

Start with a cycle $C = v_1v_2 \cdots v_nv_1$ on n vertices. Now add the following path P of length $n-1$ from v_1 to its "antipode" $v_{\lfloor n/2 \rfloor}$:

$$P = v_1v_2v_nv_3v_{n-1}v_4v_{n-2} \cdots v_{\lfloor n/2 \rfloor}.$$

Call this graph H.
Construct the distance graph G of H: A complete graph with $V(G) = V(H)$, with weights $d(uv) = \text{dist}_H(u, v)$, the length of the shortest path between u and v. This graph automatically satisfies the triangle inequality, and the minimum Hamilton cycle is C, of weight n. Note that $d(v_1v_{\lfloor n/2 \rfloor}) = \lfloor n/2 \rfloor$, because P creates no shorter path from v_1 to $v_{\lfloor n/2 \rfloor}$ than going around one half of C.
If the MST used in Christofides is the path P, then the only odd-degree vertices are v_1 and $v_{\lfloor n/2 \rfloor}$. The only possible matching is then $M = \{v_1v_{\lfloor n/2 \rfloor}\}$. Adding $v_1v_{\lfloor n/2 \rfloor}$ to P gives a cycle, which will be the resulting Hamilton cycle, with weight $d(P) + d(v_1v_{\lfloor n/2 \rfloor}) = n - 1 + \lfloor n/2 \rfloor$, which goes to $3n/2$ as $n \to \infty$.

2. *Show that a k-approximation algorithm for Metric TSP gives a k-approximation algorithm for TSPR.*

> Travelling Salesman Problem with Repetitions (TSPR): Given an undirected complete graph G, with weights $w : E(G) \to \mathbb{R}_{\geq 0}$, find a tour (closed walk) that visits every vertex *at least* once.

Let G be a graph with weights w in which to find a tour that visits every vertex at least once. Construct its distance graph G': A complete graph with $V(G') = V(G)$, with weights $w'(uv)$ defined as the weight of the shortest path between u and v. This can be constructed in polynomial time because the weights are nonnegative, by an earlier problem. Clearly G' is metric. We claim that a minimum tour visiting every vertex in G at least once corresponds to a minimum Hamilton cycle in G' of the same weight. This proves the statement.
Let T be a tour in G that visits every vertex. Let $\{v_1, \ldots, v_n\}$ be all the vertices of G, in the order in which they appear in T. Then $H = v_1 \cdots v_nv_1$ is a Hamilton cycle in G'. T can be split into subwalks P_i from v_i to v_{i+1}, and then we have $w'(v_iv_{i+1}) \leq w(P_i)$ for all i by definition of w'. Hence

$$w'(H) = \sum w'(v_iv_{i+1}) \leq \sum w(P_i) = w(T).$$

Note that we do not necessarily have equality here.
Let $H = u_1u_2 \cdots u_nu_1$ be a Hamilton cycle in G'. Then for every u_iu_{i+1}, there is a path Q_i in G from u_i to u_{i+1} of length $w(Q_i) = w'(u_iu_{i+1})$. Let T be the tour obtained by concatenating all Q_i. Then $w(T) = w(H)$.
This proves the claim: A minimum such tour T gives a Hamilton cycle H with $w'(H) \leq w(T)$. If we had $w'(H) < w(T)$, then we could use H to construct such a tour $\widetilde{T}$ with $w(\widetilde{T}) = w'(H) < w(T)$, contradicting minimality of T. So $w'(H) = w(T)$.

3. *Find a 4/3-approximation algorithm for 1-2-TSP.*
(Hint: Cover the vertices by cycles of minimum total weight, then patch them together.)

Travelling Salesman Problem (1-2-TSP): Given an undirected complete graph G, with weights $w : E(G) \to \mathbb{R}_{\geq 0}$ such that all $w(e) \in \{1, 2\}$, find a Hamilton cycle of minimum weight.

Below we will show how to find a minimum cycle cover. First suppose we have a minimum cycle cover C^*, and let H^* be the minimum Hamilton cycle. Since H^* is also a cycle cover, we have $w(C^*) \leq w(H^*)$. Also $n \leq w(H^*)$ because all weights are ≥ 1.
We will repeatedly patch together two cycles two create a new cycle. Since every cycle has at least 3 vertices, we will need at most $\lfloor n/3 \rfloor - 1$ patching steps. We'll show that we can do this in such a way that the total weight of the cycles increases by at most $\lfloor n/3 \rfloor$. Then we will get a Hamilton cycle of weight

$$w(C^*) + \lfloor n/3 \rfloor \leq w(H^*) + n/3 \leq w(H^*) + w(H^*)/3 = \frac{4}{3} w(H^*).$$

That leaves two things to work out: how to patch the cycles while not increasing the weight too much, and how to find a minimum cycle cover.

- *Patching the cycles:* We patch two cycles together by removing one edge from each, say e_1 and e_2, then adding in two new edges (possibly because the graph is complete) to create one large cycle, say e_3 and e_4. The only way that this could increase the weight by more than 1 is if e_1 and e_2 both had weight 1, and e_3 and e_4 both had weight 2; the total weight would then increase by 2. We can avoid this as long as we choose e_1 to have weight 2, which is possible when there is some cycle in the cycle cover that has some edge with weight 2. The only situation where we cannot avoid this is if all edges in the cycle cover have weight 1.
So we might have to increase the weight by 2 in one step. But after that, there will be an edge of weight 2, so we can avoid this from then on. Well, actually, it is possible that we return to the situation where all edges in the cycles have weight 1, but only after a step in which the total weight decreased. So anyway, we can guarantee that after k patching steps, the total weight has increased by at most $k + 1$.
So the $\lfloor n/3 \rfloor - 1$ patching steps will increase the total weight by at most $\lfloor n/3 \rfloor$.

- *Minimum Cycle Cover:* We construct a new graph G'. Replace each vertex v as follows: Put two vertices z_1 and z_2; for each incoming edge e, connect it to a vertex x_e, connect x_e to a vertex y_e, and connect y_e to both z_1 and z_2. The incoming edge has the same weight as in G, every new edge has weight 0. So, for a vertex with k incoming edges, there are k vertices x_e, k vertices y_e, and 2 vertices z_1, z_2.
Find a minimum weight perfect matching for G'. We claim that this corresponds to a minimum weight cycle cover. The weight is clearly the same. For each vertex v, the z_1 and z_2 are matched to some y_{e_1} and y_{e_2}. The other y_e must be matched to the corresponding x_e, so x_{e_1} and x_{e_2} must be matched by the edge with nonzero weight. Viewing this whole thing as one vertex, we see that the vertex is matched by exactly two edges. Hence this is a cycle cover (aka 2-matching or 2-factor).

The cycle cover part seems overly complicated, please let me know if you find something simpler. I should have asked the question for directed TSP, where this part is much easier...

Section 12.5:

1. *Show that Next Fit is not a k-approximation algorithm for any $k < 2$.*
Find an example for which First Fit (without sorting) has an approximation factor 5/3.

We can force Next Fit to be very inefficient by making it leave bins mostly unfilled. A sequence like the following would do this, for some small $\varepsilon > 0$:

$$2\varepsilon, 1-\varepsilon, 2\varepsilon, 1-\varepsilon, 2\varepsilon.$$

Next Fit would give each number its own bin, because no two consecutive bins fit together. But there is a more efficient packing (which First Fit would find): give each $1-\varepsilon$ its own bin, and put all the 2ε together in one bin. For this we have to make ε small enough; for example, for the sequence above, $\varepsilon = 1/6$ would do. Then in this case we would have an approximation factor of 5/3.
To generalize this, we take a sequence like above with m times 2ε, and we set $\varepsilon = 1/2m$. Then Next Fit will use $2m-1$ bins (m for the 2ε, $m-1$ for the $1-\varepsilon$), but the best packing uses m bins (1 for the 2ε, $m-1$ for the $1-\varepsilon$). So the approximation factor is $2-1/m$, which proves that no approximation factor < 2 is possible.

This example does what is required (for bins of capacity 24):

$$7, 7, 7, 4, 4, 4, 13, 13, 13.$$

First Fit will need 5 bins, but 3 bins is possible.
Here's how you might come up with a small example like this. After some playing around you could guess at a "symmetric solution", with 3×3 items of sizes α, β, γ filling 3 bins exactly, so $\alpha + \beta + \gamma = 1$. The way to make First Fit do badly is to have $\gamma = 1/2 + \epsilon$, so that the γ items will take up the last 3 bins of 5, and so that the 3 α items end up in the first bin, the 3 β items in the second.
Now you can deduce what they should be. The two constraints are that a β does not fit in with the 3 α's, and that a γ does not fit in with the 3 β's. So

$$3\alpha + \beta > 1, \quad 3\beta + \gamma > 1.$$

The second together with $\gamma = 1/2 + \epsilon$ gives $\beta > 1/6 - \epsilon/3$. We take

$$\beta = \frac{1}{6} - \epsilon/3 + \delta, \quad \text{so} \quad \alpha = 1 - \beta - \gamma = \frac{1}{3} - \frac{2}{3}\epsilon - \delta.$$

Then $3\alpha + \beta > 1$ gives $14\epsilon + 12\delta < 1$. Any pick of small ϵ, δ that satisfies this will work, but to get nice integers we pick $\epsilon = 1/24, \delta = 1/72$ (found by playing with the divisibility a little). Then $24\alpha = 7, 24\beta = 4, 24\gamma = 13$.

2. *Show that Next Fit is a 2-approximation algorithm for* Bin Packing.

Let k be the number of bins used by Next Fit, and k^* the minimal number of bins, so we want to show $k \leq 2k^*$. Write $S = \sum_{i \in I} s_i$, so $k^* \geq S$.
We use the fact that Next Fit ensures that any two adjacent used bins together contain items of size larger than 1, since otherwise the items in the second bin would have been place in the first. So for every odd j, except maybe the last, we have

$$\sum_{i:a(i)=j \text{ or } j+1} s_i > 1.$$

We have $\lfloor k/2 \rfloor$ such inequalities. Summing all of them gives $S > \lfloor k/2 \rfloor$, and since these are integers we get

$$S \geq \lfloor k/2 \rfloor + 1 \geq \frac{k-1}{2} + 1,$$

which implies

$$k \leq 2S - 1 \leq 2S \leq 2k^*.$$

Alternative proof: Take the first element of bins 2 to k, and add that element to the bin before it. That will give $k-1$ overfilled bins. Hence we have

$$k - 1 < S + \sum (\text{first element of each bin}) \leq 2S \leq 2k^*.$$

Since these are integers, $k - 1 < 2k^*$ implies $k \leq 2k^*$

3. *We call an algorithm for Bin Packing monotonic if the number of bins it uses for packing a list of items is always larger than for any sublist.*
Show that Next Fit is monotonic, but First Fit is not.

   If Next Fit were not monotonic, then there would be an instance I such that removing one of its items would lead to more bins. It's pretty easy to see that this is not possible. Let i be the item removed. The items $i' < i$ remain in the same bin, while the items $i'' > i$ can only be moved to an earlier bin. So clearly the number of bins used can only become less.

   The following example (with bin size 10) shows that First Fit is not monotonic:

   $$4, 6, 4, 1, 6, 1, 4, 4.$$

   First Fit uses 3 bins, but if we remove the first 1, it will use 4 bins.

4. *Show that instances of* Bin Packing *for which $s_i > 1/3$ for all item sizes s_i can be solved exactly using a polynomial algorithm that we saw earlier in the course.*
Show that this is not that useful, because FFD solves these instances exactly as well.

   Construct a graph with the items for vertices, and an edge between items i and j if $s_i + s_j < 1$. Then a matching in this graph will correspond to a bin packing, by putting any two matched items into a bin together. Because of the minimum size 1/3, no bin can contain more than 3 items, so every bin packing corresponds to a matching. The number of bins equals the number of matching edges plus the number of unmatched vertices. This number is minimized when the matching has maximum cardinality, so we can use the blossom algorithm for maximum cardinality matchings.

   The first part is not so useful because FFD is much faster.
   Let the *large* items be those with $s_i > 1/2$ and the *medium* items $1/3 < s_i \leq 1/2$.
   In any packing, each large item will get its own bin, some of the medium items will be added to a bin with a large item, and the remaining medium items are either paired off, or get their own bins. In a minimum packing, these remaining medium items will all be paired off arbitrarily, except possibly a single one. This is exactly what FFD will do, and the number of bins used only depends on the number of remaining medium items. It is not hard to see that FFD will maximize the number of the large-medium pairs, so it will minimize the number of remaining medium items, hence also the total number of bins.