CO 353 Course Notes

Computational Discrete Optimization

Kanstantsin Pashkovich • Winter 2023 • University of Waterloo

Table of Contents

1 Shortest Paths

1.1 Preliminaries on Graphs

An **(undirected) graph** G is a pair (V, E), where E is a set of unordered pairs of elements in V. The elements of V are called **vertices** or **nodes**; the elements of E are called **edges**.

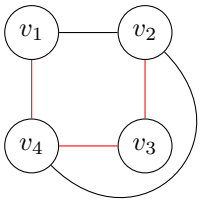
Let $u, v \in V$ and let $e = uv \in E$ be an edge.

- We say that e is **incident** to u and v.
- The vertices u and v are said to be **adjacent**.
- We call u and v the **endpoints** of e.

By default, we assume that there are no parallel edges (i.e. two edges $e = uv$ and $e' = u'v'$ in E with $\{u, v\} = \{u', v'\}$) and no loops (i.e. an edge $e = uv \in E$ with $u = v$).

For distinct $u, v \in V$, a u, v**-path** is a sequence of vertices $w_1, \ldots, w_k$ such that $w_1 = u$, $w_k = v$, and $w_i w_{i+1} \in E$ for all $i = 1, \ldots, k-1$.

For example, consider the following graph $G = (V, E)$ with vertices $V = \{v_1, v_2, v_3, v_4\}$ and edges $E = \{v_1v_2, v_1v_4, v_2v_3, v_2v_4, v_3v_4\}$.



The lines in red form a v_1, v_2-path, namely v_1, v_4, v_3, v_2. Another v_1, v_2-path can be obtained by simply traversing the edge v_1v_2.

A **cycle** in G is a sequence of vertices $w_1, \ldots, w_{k+1}$ such that $w_i w_{i+1} \in E$ for all $i = 1, \ldots, k$, the vertices $w_1, \ldots, w_k$ are all distinct, and $w_1 = w_{k+1}$.

Finally, a graph G is **connected** if for any pair of distinct vertices $u, v \in V$, there exists a u, v-path in G.

1.2 Shortest Paths Problem

Given a *directed* graph $G = (V, E)$ with edge lengths $\ell_e \geq 0$ for each $e \in E$ and a distinguished start vertex $s \in V$, we wish to find shortest paths from s to every other vertex in V. Note that when we work with directed graphs, we will denote the directed edges with (v_1, v_2) as opposed to v_1v_2 in the case of undirected graphs, where the order of the vertices did not matter.

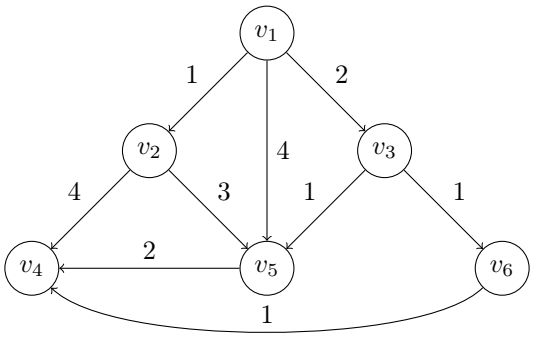
The **length** of a path P given by the sequence $w_1, \ldots, w_k$ is given by

$$\ell(P) := \sum_{i=1}^{k-1} \ell_{(w_i, w_{i+1})} = \sum_{e \in P} \ell_e,$$

where the second sum makes sense because there are no parallel edges. Then the **shortest-path distance** from s to a vertex $u \in V$ is defined to be

$$d(u) := \min_{s,\, u\text{-paths } P} \ell(P).$$

For example, we can consider the following instance of an undirected graph with given edge lengths and starting vertex $s = v_1$.



In this case, we have $d(v_2) = 1$, since the only possible path from v_1 to v_2 is by taking the edge (v_1, v_2). There are multiple paths from v_1 to v_5; the shortest one is v_1, v_3, v_5 giving $d(v_5) = 3$.

Note that we always set $d(s) = 0$. We now make some observations:

(i) If $(u, v) \in E$, then $d(v) \leq d(u) + \ell_{(u,v)}$, since such an s, v-path is always an option.

(ii) For every $v \in V$ distinct from s, there exists $w \in V$ such that $d(v) = d(w) + \ell_{(w,v)}$ and $(w, v) \in E$. This can be seen by chopping off the last edge from a shortest path from s to v.

1.3 Dijkstra's Algorithm

In 1959, Dijkstra came up with the following algorithm to solve the shortest paths problem. The main idea is to maintain a set $A \subseteq V$ of "explored" nodes; that is, a set of nodes for which we already know the shortest-path distances. We'll also maintain labels $d'(v)$ for $v \in V \setminus A$ with upper bounds on the shortest-path distances from s.

Input. A directed graph $G = (V, E)$, edge lengths $\ell_e \geq 0$ for all $e \in E$, and a start vertex $v \in V$.

Output. For all $v \in V$, the length $d(v)$ for the shortest-path from s to v.

Step 1. **(Initialization.)** Set $A \leftarrow \{s\}$, $d(s) \leftarrow 0$, and $d'(v) \leftarrow \infty$ for all $v \in V \setminus A$.

Step 2. While $A \neq V$:

Step 2.1. **(Push down the upper bounds.)** For each $v \in V \setminus A$, compute

$$d'(v) \leftarrow \min \left\{ d'(v), \min_{\substack{u \in A \\ (u,v) \in E}} \{d(u) + \ell_{(u,v)}\} \right\}.$$

Step 2.2. **(Add a new vertex.)** Set $w \leftarrow \arg\min_{v \in V \setminus A} d'(v)$, $A \leftarrow A \cup \{w\}$, and $d(w) \leftarrow d'(w)$.

Suppose that for each vertex $w \in V$, we keep track of the node u determining its upper bound $d'(w)$. That is, the node u is such that $(u, w) \in E$ and $d'(w) = d(u) + \ell_{(u,w)}$. Then at the end of the algorithm, a shortest path from s to w can be obtained as a shortest path from s to u adjoined with the edge $(u, w) \in E$. Moreover, these edges selected by Dijkstra's algorithm form an arborescence, which is a nice graph structure that we'll discuss more later.

Next, let's prove the correctness of Dijkstra's algorithm. In particular, we need to show that for every $v \in V$, the distance from s to v is computed correctly. We'll assume that the graph is connected; that is, for every $v \in V$, there is an s, v-path in G. (Note that the algorithm won't terminate otherwise, but it can be adjusted to deal with this.)

Proof of correctness of Dijkstra's algorithm.

We proceed by induction on $|A|$, and show that at each point in time, $d(v)$ is computed correctly for all $v \in A$. The case where $|A| = 1$ is clear because at the start of the algorithm, we initialize $A = \{s\}$ with $d(s) = 0$, which is correct.

Assume that $d(v)$ is computed correctly for every $v \in A$ when that $|A| = k$. Suppose that we are adding a new vertex w to A in Step 2.2 of the algorithm. Consider the vertex $u \in A$ such that $(u, w) \in E$ and

$$d'(w) = d(u) + \ell_{(u,w)}.$$

Specifically, this is the vertex u determining the upper bound $d'(w)$ which we discussed in the paragraph following the description of the algorithm.

For the sake of contradiction, assume that the distance from s to w is not $d'(w)$. Let P_u be a shortest path from s to u, and let P' be a shortest path from s to w. Then by our assumption, we know that

$$\ell(P') < \ell(P_u) + \ell_{(u,w)} = d'(w).$$

Now, let $x, y \in V$ be such that $(x, y) \in E$ lies on the shortest path P' from s to w, with $x \in A$ and $y \in V \setminus A$. (This exists because at some point, the path must exit A to get from s to w.) Then we obtain

$$d'(y) \leq d(x) + \ell_{(x,y)} \leq \ell(P') < \ell(P_u) + \ell_{(u,w)} = d'(w),$$

where the first inequality is because of how $d'(y)$ is computed in Step 2.1, and the second inequality is because the shortest path from x to y adjoined with the edge (x, y) is part of the path P', noting that $\ell_e \geq 0$ for all $e \in E$. But this contradicts our choice of $w = \arg\min_{v \in V \setminus A}\{d'(v)\}$ in Step 2.2 since $y \in V \setminus A$ but $d'(y) < d'(w)$. □

The **running time** of an algorithm is the number of elementary operations that the algorithm performs as a function of the input size. The **input size** is the number of bits needed to specify the input.

- For example, in order to specify an integer $n \geq 0$ in binary, we require about $\log_2(n)$ bits.
- To specify a graph $G = (V, E)$ with integral edge lengths $\ell_e \geq 0$ for each $e \in E$, we require approximately $|V| + |E| + \sum_{e \in E} \log_2(\ell_e)$ bits. Note that we can specify an edge with two pointers to the endpoints.

We will need big-O notation to describe running time, because we are interested in the asymptotic behaviour of algorithms. For two functions $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$, we say that $f(n) = O(g(n))$ if there exist constants $n_0 \in \mathbb{N}$ and $c \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. For example, we have $2n^2 + 1 = O(n^2)$ and $\log(n) = O(n)$. An algorithm is then considered **efficient** if its running time is bounded above by a polynomial function of the input size.

Let's consider the running time of Dijkstra's algorithm by looking at a naive implementation. For ease of notation, we will write $|V| = n$ and $|E| = m$. Note that $G = (V, E)$ is connected, so $n - 1 \leq m \leq n^2$.

- Step 1 can be performed using $O(n)$ operations since we are only assigning values for each vertex.
- In Step 2, there are at most n iterations.
  - Step 2.1 can be performed using $O(m)$ operations because each edge will participate in at most two comparisons throughout the entire iteration.
  - Step 2.2 takes $O(n)$ operations in order to determine the vertex of minimum upper bound.

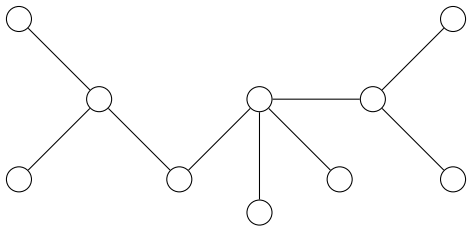
Therefore, the running time of the naive implementation is $O(n + mn + n^2) = O(mn)$, which is polynomial in the input size.

We note that there are better implementations of Dijkstra's algorithm than the naive one that we have just stated. For Step 2.1, we can use m Decrease-Key calls and for Step 2.2, we can use n Extract-Min calls. In particular, by using Fibonacci heaps, Decrease-Key has running time $O(1)$ and Extract-Min has running time $O(\log n)$, which brings the total running time down to $O(m + n \log n)$.

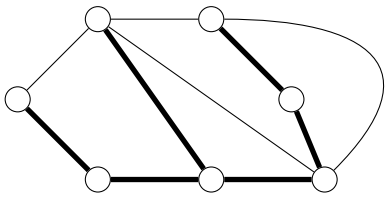
2 Minimum Spanning Trees

2.1 Trees

A **tree** is a connected acyclic graph; that is, a connected graph containing no cycles.



Given a graph $G = (V, E)$, a **spanning tree** of G is a graph $T = (V, F)$ such that $F \subseteq E$ and T is a tree. We illustrate an example of a graph G with a subtree T using bold edges below.



In an introductory graph theory course, such as MATH 239, it is shown that every tree on n vertices has $n - 1$ edges. The following theorem then gives us a useful characterization of trees.

Theorem 2.1: Fundamental Theorem of Trees

Let $T = (V, F)$ be a graph. The following are equivalent:

(i) T is a tree.

(ii) T is connected and $|F| = |V| - 1$.

(iii) T is acyclic and $|F| = |V| - 1$.

In particular, if we know that two of the conditions hold, then the third one is guaranteed.

2.2 Minimum Spanning Trees

Given a connected graph $G = (V, E)$ and edge costs c_e for each $e \in E$, our goal is to find a spanning tree T of minimum cost

$$c(T) := \sum_{e \in T} c_e.$$

First, we'll set some notation. For a vertex $v \in V$, we define $\delta(v)$ to be the set of edges in E incident to v. More generally, given a subset of vertices $S \subseteq V$, the **cut induced by** S is defined to be the set

$$\delta(S) := \{uv \in E : u \in S, v \notin S\}.$$

The following theorem will be extremely important for finding a minimum spanning tree.

Theorem 2.2: Cut Property

Suppose that the costs c_e for $e \in E$ are distinct. Let $S \subseteq V$ be such that $S \neq \varnothing$ and $S \neq V$, and let

$$e = \arg\min_{f \in \delta(S)} c_f.$$

Then every minimum spanning tree contains the edge e.

Proof of Theorem 2.2.

We proceed by contradiction. Let $S \subseteq V$ be such that $S \neq \varnothing$ and $S \neq V$, and let $e = \arg\min_{f \in S} c_f$. Suppose that there is a minimum spanning tree $T = (V, F)$ such that $e \notin F$.

Consider the graph $(V, F \cup \{e\})$. Note that $|F \cup \{e\}| = |V|$ and this graph is connected, so it cannot be a tree by Theorem 2.1. In particular, it must contain a cycle C.

Next, note that $|C \cap \delta(S)|$ must be even because for any edge leaving the cut $\delta(S)$, there must be another edge coming back into the cut. Moreover, since $e \in C \cap \delta(S)$, we have $C \cap \delta(S) \neq \varnothing$. This implies that $|C \cap \delta(S)| \geq 2$, so there is another edge $e' \neq e$ with $e' \in C \cap \delta(S)$.

Consider now the graph $T' = (V, (F \cup \{e\}) \setminus \{e'\})$. Then $|(F \cup \{e\}) \setminus \{e'\}| = |F| = |V| - 1$ and T' is connected because if a path between two vertices used the edge e', then we could go along the edges in $C \setminus \{e'\}$ instead. So by Theorem 2.1, T' is also a spanning tree. Finally, observe that

$$c(T) - c(T') = c_{e'} - c_e > 0$$

because we have $e = \arg\min_{f \in S} c_f$ and $e' \neq e$ with $e' \in \delta(S)$, as well as the assumption that the edge costs c_e for $e \in E$ were distinct. But T' has lower cost than T, contradicting our assumption that T was a minimum spanning tree. □

The following property relating cycles to minimum spanning trees can also be proved similarly to Theorem 2.2. The main idea is to try to create shortcuts using the edges in the cycle.

Theorem 2.3: Cycle Property

Let $G = (V, E)$ be connected with distinct edge costs c_e for $e \in E$. Let C be a cycle in G and let

$$e = \arg\max_{f \in C} c_f.$$

Then no minimum spanning tree contains the edge e.

2.3 Prim's Algorithm

Prim's algorithm takes the idea of the cut property (Theorem 2.2) and uses it to compute a minimum spanning tree starting from an arbitrary vertex $s \in V$. At each iteration of the algorithm, we will keep track of a set of a partial tree construction T and vertices $A \subseteq V$ that are connected to s in T. We finish once we have reached all the vertices in G.

Input. A connected graph $G = (V, E)$ and edge costs c_e for all $e \in E$.

Output. The edges T of a minimum spanning tree of G.

Step 1. **(Initialization.)** Let $s \in V$ be an arbitrary vertex, then set $A \leftarrow \{s\}$ and $T \leftarrow \varnothing$.

Step 2. While $A \neq V$:

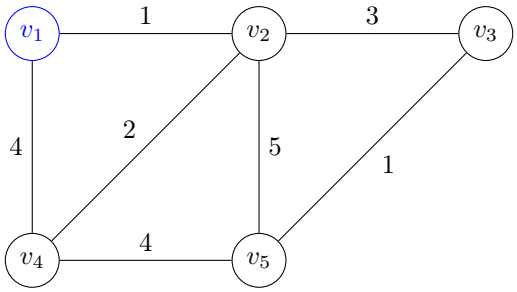
Step 2.1. **(Pick an edge for the minimum spanning tree.)** Set

$$e \leftarrow \arg\min_{f \in \delta(A)} c_f,$$

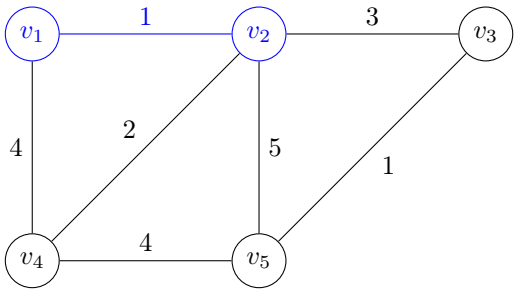
where $e = uv$ with $u \in A$ and $v \notin A$.

Step 2.2. **(Update values.)** Set $A \leftarrow A \cup \{v\}$ and $T \leftarrow T \cup \{e\}$.

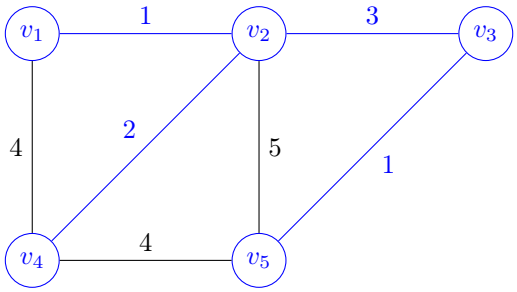
Let's run Prim's algorithm on a simple example. Consider the following graph $G = (V, E)$, and suppose that at Step 1, we pick v_1 to be the initial vertex.



Then at the first iteration of Step 2.1, we have $\delta(A) = \{v_1v_2, v_1v_4\}$, so we pick $e = v_1v_2$ because it has minimum cost. At Step 2.2, we set $A = \{v_1, v_2\}$ and $T = \{v_1v_2\}$.



The edge v_2v_4 is of minimum cost in the cut induced by A, so we set $A = \{v_1, v_2, v_4\}$ and $T = \{v_1v_2, v_2v_4\}$ at the next iteration. Continuing in this fashion, we obtain the following minimum spanning tree.



Note that Prim's algorithm may seen similar to Dijkstra's algorithm which also yields a spanning tree in the process of computing shortest paths. However, these algorithms fundamentally solve different problems, and there are examples where the resulting spanning trees do not coincide. Moreover, Dijkstra's algorithm takes a directed graph as input, whereas Prim's algorithm takes an undirected graph.

Next, let's prove that Prim's algorithm always gives us a minimum spanning tree. For simplicity, we will assume that all edge costs c_e for $e \in E$ are distinct.

Proof of correctness of Prim's algorithm.

> At every iteration of Step 2, we add one edge to T, and we run through $|V| - 1$ iterations. Therefore, T has exactly $|V| - 1$ edges. Moreover, an invariant of Prim's algorithm is that all vertices in A remain connected to s, so the final output is also connected. Therefore, we indeed obtain a spanning tree by Theorem 2.1. Finally, by the cut property (Theorem 2.2) and our assumption that all the edge costs are distinct, T contains only the edges that are in every minimum spanning tree, so T itself is a minimum spanning tree. □

As a consequence, we also have the following useful result.

COROLLARY 2.4

Let $G = (V, E)$ be a connected graph with distinct edge costs c_e for $e \in E$. Then G has a unique minimum spanning tree.

As usual, let's consider the running time, denoting $|V| = n$ and $|E| = m$. For an efficient implementation, we keep track of a key $d'(v)$ for each $v \in V \setminus A$. Once a vertex w is added to A in Step 2.2, we update the keys via $d'(v) \leftarrow \min\{d'(v), c_{wv}\}$ for each $v \in V \setminus A$.

Note that Step 1 takes $O(1)$ time, and there are $n - 1$ iterations of Step 2. Implementing Prim's algorithm using priority queues, we note that each iteration of Step 2.1 involves one EXTRACT-MIN call, and going through all iterations of Step 2.2 takes at most m DECREASE-KEY calls in total. We recall that by using Fibonacci heaps, DECREASE-KEY is $O(1)$ and EXTRACT-MIN is $O(\log n)$, so we have a total running time of $O(n \log n + m)$.

2.4 Kruskal's Algorithm

Kruskal's algorithm is another greedy algorithm that finds a minimum spanning tree. It first sorts the edges by ascending edge costs and continually adds an edge to a partial tree construction T as long as no cycle is induced by that edge. The algorithm we give stops once we go through every edge, but note that it is enough to stop once the number of edges we've added hits $|V| - 1$.

> **Input.** A connected graph $G = (V, E)$ and edge costs c_e for all $e \in E$. (We write $m = |E|$ and $n = |V|$.)
>
> **Output.** The edges T of a minimum spanning tree of G.
>
> Step 1. **(Initialization.)** Sort the edges such that $c_{e_1} \leq c_{e_2} \leq \cdots \leq c_{e_m}$. Set $T \leftarrow \varnothing$ and $j \leftarrow 1$.
>
> Step 2. While $j \neq m + 1$:
>
> **(Add an edge if it does not form a cycle.)** If $T \cup \{e_j\}$ is acyclic, set $T \leftarrow T \cup \{e_j\}$.
>
> **(Go to the next edge.)** Regardless if we add an edge or not, increment $j \leftarrow j + 1$.

To prove the correctness of Kruskal's algorithm, we'll again use the cut property (Theorem 2.2). As with Prim's algorithm, the proof will consist of two parts: proving that the result T is a spanning tree, and showing that it is of minimum cost. We will also assume that the edge costs c_e are distinct.

We use a couple of facts that will be repeated many times throughout the course. These concern equivalent notions to the definitions we have.

(1) A graph $G = (V, E)$ is disconnected if and only if there exists a nontrivial cut $\delta(S)$ for some $\varnothing \subsetneq S \subsetneq V$ such that $\delta(S) \cap E = \varnothing$.

(2) Let $G = (V, E)$ be a graph and let $u, v \in V$ be distinct vertices. Then G has no u, v-path if and only if there exists $S \subseteq V$ such that $u \in S$, $v \notin S$, and $\delta(S) \cap E = \varnothing$.

Proof of correctness of Kruskal's algorithm.

Let T denote the output set consisting of edges. We know that T is acyclic by construction because Step 2 of the algorithm rules out all edges that create a cycle. We will show that (V, T) is connected by way of contradiction, and thus it is a spanning tree by Theorem 2.1.

Suppose not, so there is a nontrivial cut $\delta(S)$ for some $\varnothing \subsetneq S \subsetneq V$ such that $\delta(S) \cap T = \varnothing$ by (1) above. Since G is connected, we have $\delta(S) \neq \varnothing$, so there exists some edge $e \in \delta(S)$. Let's look at the moment where e was considered in the algorithm. Since e was rejected, it must be that $T \cup \{e\}$ contains a cycle C with $e \in C$. But $|C \cap \delta(S)|$ is even and $e \in C \cap \delta(S)$, so $|C \cap \delta(S)| \geq 2$. We then have $|(C \setminus \{e\}) \cap \delta(S)| \geq 1$. But $C \setminus \{e\} \subseteq T$, which contradicts the fact that $\delta(S) \cap T = \varnothing$.

Now, we show that we have a minimum spanning tree. Consider the moment an arbitrary edge $e = uv$ is added to T; let T' be the set of edges in T before the addition of e to T. Then T' has no u, v-path (else a u, v-path adjoined with e would form a cycle). Hence, by (2), there exists $S \subseteq V$ such that $u \in S$, $v \notin S$, and $\delta(S) \cap T' = \varnothing$. Due to the way that the edges are sorted in Step 1, we have $e = \arg\min_{f \in \delta(S)} c_f$. It follows from the cut property (Theorem 2.2) that including e is the correct decision. □

To implement Kruskal's algorithm, first observe that Step 1 takes $O(m \log m)$ time to sort m numbers. For Step 2, we can make use of the UnionFind data structure, which helps maintain a list of connected components. It has the following methods:

- MakeUnionFind(V) creates a UnionFind data structure for V and takes $O(n)$ time.
- Find(v) determines the name of the connected component where v lies. This takes $O(\log n)$.
- Union(A, B) merges two connected components and takes $O(1)$ via a change of pointer.

We initialize the data structure once with MakeUnionFind. We use Find in every iteration of Step 2 to check that Find(u) $\neq$ Find(v) for the edge $e = uv$. In this case, u and v are in different components and we can join them with Union; otherwise we don't have to do anything. There are m iterations of Step 2, so Kruskal's algorithm takes $O(m \log n) = O(m \log m)$ time (since $n - 1 \leq m \leq n^2$ when G is connected).

2.5 Maximum Spacing Clustering

Given a set $U = \{p_1, \ldots, p_n\}$ of n objects and a distance $d(p_i, p_j) = d(p_j, p_i) \geq 0$ between points, we wish to find a **k-clustering** (a partition $C_1, \ldots, C_k$ of the set U) with maximum spacing

$$\max_{C_1,\ldots,C_k \text{ partition of } U} \left\{ \min_{1 \leq i < j \leq k} \left\{ \min_{p \in C_i, q \in C_j} d(p, q) \right\} \right\}.$$

Note that k is fixed above, and the term inside the maximum is the **spacing** of the k-clustering $C_1, \ldots, C_k$.

We can solve this problem with an adaptation of Kruskal's algorithm, called the **single linkage clustering algorithm**. We can view it as being equivalent to working over the complete graph K_n on the points $p_1, \ldots, p_n$, and the edge costs corresponding to the distances. Here, we can stop early once we hit k clusters.

Input. A set $U = \{p_1, \ldots, p_n\}$ with distances $d(p_i, p_j) \geq 0$ between points, and a fixed integer k.

Output. A k-clustering of U with maximum spacing.

Step 1. **(Initialization.)** Start with n clusters, each containing its own object from $p_1, \ldots, p_n$.

Step 2. While #clusters $> k$:

(Merge clusters.) Merge the clusters C and C' that achieve

$$\min_{C \neq C' \text{ clusters}} \left\{ \min_{p \in C, q \in C'} d(p, q) \right\}.$$

3 Minimum Cost Arborescences

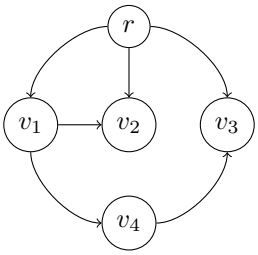
3.1 Arborescences and a Characterization

Let $G = (V, E)$ be a directed graph and let r be a distinguished node, which is commonly called a root. An **arborescence** with respect to r (or rooted at r) is a directed subgraph $T = (V, F)$ with $F \subseteq E$ such that

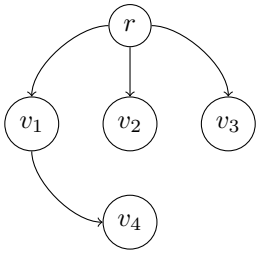
(i) undirected T (that is, T obtained from disregarding all directions) is a spanning tree; and

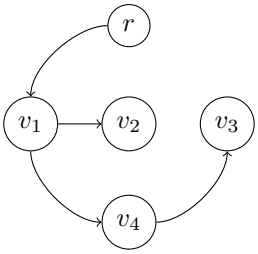
(ii) for every $v \in V$ with $v \neq r$, there is a directed path in T from r to v.

For example, consider the following graph $G = (V, E)$.

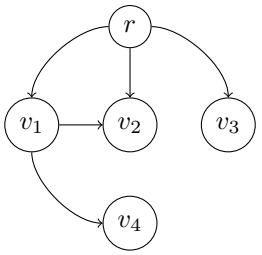


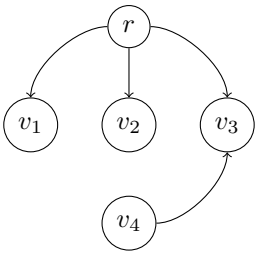
Then the following two subgraphs are arborescences rooted at r.





On the other hand, the following two subgraphs are not arborescences rooted at r: the first one is not a tree, and the second has no directed path from r to v_4.





The following theorem gives us a useful characterization of arborescences.

Theorem 3.1: Characterization of Arborescences

Let $G = (V, E)$ be a connected graph and let $T = (V, F)$ be a subgraph. Then T is an arborescence rooted at r if and only if both of the following conditions hold:

(1) every $v \in V$ with $v \neq r$ has exactly one incoming edge in T; and

(2) T has no directed cycles.

Proof of Theorem 3.1.

($\Rightarrow$) First, we check that condition (1) holds. Consider a vertex $v \in V$ with $v \neq r$. Since undirected T is a spanning tree, there is a unique (simple) path from r to v in undirected T. The last edge on this path is incoming for v. Hence, all other edges incident to v in undirected T should be outgoing edges for v (because the neighbours of v also have unique simple paths from v to them in undirected T). This proves (1). To see that condition (2) holds, note that undirected T is acyclic as it is a spanning tree, so it could not possibly have any directed cycles either.

($\Leftarrow$) Suppose that $V = (T, F)$ satisfies conditions (1) and (2). First, we show that for all $v \in V$ with $v \neq r$, there exists a directed path from r to v in T, which is condition (ii) of the definition of an arborescence. Let (v_1, v) be the unique edge incoming to v by condition (1), let (v_2, v_1) be the unique edge incoming to v_1, and so on. By contradiction, suppose that we cannot reach r by backtracking in this way. Then at some point, we must visit the same node at least twice because G is a finite graph. But this creates a directed cycle, contradicting condition (2). Thus, there is a directed path from r to v in T.

Now, we verify condition (i) that undirected T is a spanning tree. Note that we can get from r to any other vertex v, so undirected T is connected. Moreover, r has no incoming edges because if (v, r) is an incoming edge, then the directed path from r to v followed by (v, r) forms a directed cycle, contradicting condition (2). Since every edge is incoming for exactly one of its endpoints, the total number of edges in T is

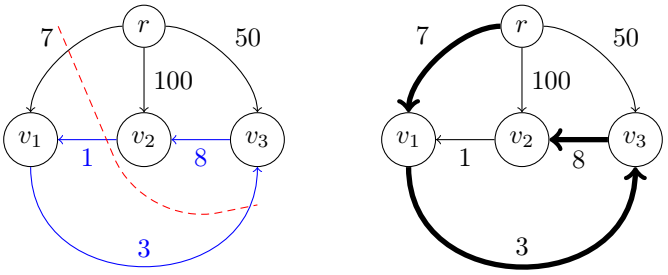
$$\sum_{u=r} 0 + \sum_{\substack{u \in V \\ u \neq r}} 1 = |V| - 1.$$

By the fundamental theorem of trees (Theorem 2.1), it follows that undirected T is a spanning tree. □

3.2 Minimum Cost Arborescences

Given a directed graph $G = (V, E)$, a distinguished node r, and edge costs $c_e \geq 0$ for each $e \in E$, our goal is to find an arborescence rooted at r so that the total edge cost is minimized.

Let's try to transfer our knowledge from the minimum spanning tree problem. Do the analogues of the cycle and cut properties hold for arborescences? It turns out we can find a counterexample to both in the same graph. Consider the following graph $G = (V, E)$ with associated edge costs.



The minimum cost arborescence is given to the right with cost $7 + 3 + 8 = 18$. Consider the cut (in red) and cycle (in blue) above. Then the cut property fails to hold because the edge (v_2, v_1) of cost 1 was not picked in a minimum cost arborescence, and the cycle property fails to hold because the edge (v_3, v_2) of maximum cost 8 in the cycle was picked in a minimum cost arborescence.

Can we instead consider the union of minimum cost incoming edges (vertex by vertex), excluding the root r? For the above example, if we start with v_1, we'd pick (v_2, v_1) as it is the minimum cost edge incoming to v_1. Then (v_1, v_3) is the minimum cost edge incoming to v_3 and (v_3, v_2) is the minimum cost edge incoming to v_2. This yields the same directed cycle highlighted in blue above!

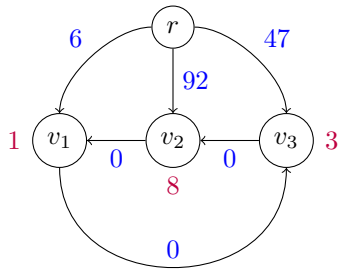
So this strategy does not immediately work. But if the strategy did happen to give us an arborescence, then we are already done! All we need to do is tweak it slightly. The idea is to compute

$$y_v = \min_{(u,v)\in E} c_{(u,v)}$$

for all vertices $v \in V$ with $v \neq r$. In particular, any edge using this vertex needs to pay this cost anyways. Then for all $(u, v) \in E$, we can define new edge costs

$$c'_{(u,v)} = c_{(u,v)} - y_v.$$

We give the edge costs c'_e (in blue) and the values y_v (in purple) for the above example. Notice that some of the edges turn out to be free.



Let's prove a useful result regarding these reduced edge costs. Its corollary will help us reason about the correctness of a minimum cost arborescence algorithm.

Lemma 3.2

For every arborescence T rooted at r, we have

$$c(T) = c'(T) + \sum_{\substack{v\in V\\ v\neq r}} y_v.$$

Proof of Lemma 3.2.

Splitting the sum to correspond to the individual vertices, we have

$$c(T) = \sum_{e\in T} c_e = \sum_{(u,v)\in T} c_{(u,v)} = \sum_{(u,r)\in T} c_{(u,r)} + \sum_{\substack{v\in V\\ v\neq r}} \sum_{(u,v)\in T} c_{(u,v)}.$$

There are no edges directed to the root r in an arborescence, so the first sum is 0. For the double summation, note that in an arborescence, there is exactly one edge directed to each $v \in V$ with $v \neq r$ by Theorem 3.1, so we only need to recompensate y_v once. This gives us

$$\begin{aligned} c(T) &= \sum_{\substack{v\in V\\ v\neq r}} \left(\sum_{(u,v)\in T} (c_{(u,v)} - y_v) + y_v \right) \\ &= \sum_{\substack{v\in V\\ v\neq r}} \sum_{(u,v)\in T} c'_{(u,v)} + \sum_{\substack{v\in V\\ v\neq r}} y_v = c'(T) + \sum_{\substack{v\in V\\ v\neq r}} y_v, \end{aligned}$$

which is the desired result. □

Since the sum of the vertex costs does not depend on the arborescence T, we obtain the following corollary.

Corollary 3.3

An arborescence T rooted at r is of minimum cost with respect to edge costs c_e if and only if it is of minimum cost with respect to the reduced edge costs c'_e.

3.3 Edmonds' Algorithm

From the ideas in the previous section, let's build some intuition for Edmonds' algorithm before we state it. For each $v \neq r$, let $f_v = \arg\min_{(u,v)\in E} c_{(u,v)}$ be an edge of minimum cost entering v. Let $F^* = \{f_v : v \neq r\}$ be the set of all of these minimum cost edges. Observe that by construction, all edges in F^* have reduced edge costs $c'_e = 0$ since they attain the minimum cost and we are subtracting that minimum cost afterwards.

If F^* does not contain a cycle, then it is a minimum cost arborescence rooted at r by our characterization from Theorem 3.1 and we are done. Otherwise, let C be a cycle in F^*. Noting that all of the edges in F^* are free with respect to the reduced edge costs, we can use as many of these edges as we want. So we can form a new graph $G' = (V', E')$ by contracting all the edges in C into a supernode, and then recursively solve the problem for G'. After solving the problem for G', we can extend it to an arborescence for G by adding edges from the cycle C.

Input. A directed graph $G = (V, E)$, edge costs $c_e \geq 0$ for all $e \in E$, and a root node r.

Output. A minimum cost arborescence of G.

Step 1. **(Initialization.)**

Step 1.1. **(Find the minimum cost entering each vertex.)** For each $v \in V$ with $v \neq r$, set

$$y_v \leftarrow \min_{(u,v)\in E} c_{(u,v)},$$
$$f_v \leftarrow \arg\min_{(u,v)\in E} c_{(u,v)}.$$

Step 1.2. **(Compute reduced edge costs.)** For each $(u, v) \in E$ where $v \neq r$, set

$$c'_{(u,v)} \leftarrow c_{(u,v)} - y_v.$$

Step 1.3. **(Group edges of minimum cost.)** Set

$$F^* \leftarrow \bigcup_{\substack{v\in V \\ v\neq r}} f_v.$$

An equivalent way to see this is that for each $v \neq r$, we are picking one edge $(u, v) \in E$ satisfying $c'_{(u,v)} = 0$.

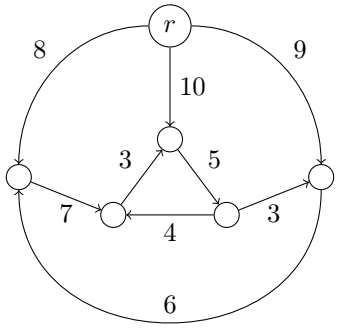
Step 2. **(Recursive part of the algorithm.)**

Step 2.1. **(Base case.)** If F^* is an arborescence rooted at r, then stop and output F^*.

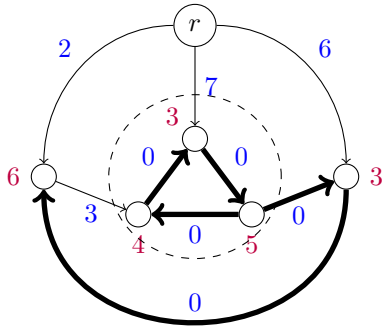
Step 2.2. **(Recursive call.)** Let C be a directed cycle in F^* (which exists by Theorem 3.1). Contract C to a supernode to obtain a graph $G' = (V', E')$, keeping the reduced costs c'_e. Recursively call Edmonds' algorithm to find a minimum cost arborescence F' for G'. Extend F' to an arborescence for G by adding all but one edge in C. Then stop and return this arborescence.

Before we prove correctness, we'll first give a simple example of Edmonds' algorithm in action.

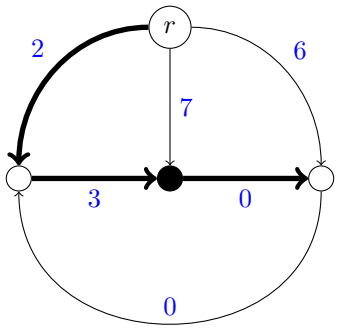
Consider the following graph $G = (V, E)$ with associated edge costs c_e.



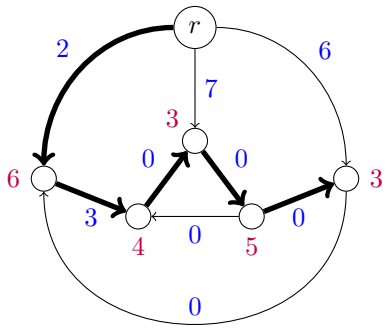
In Step 1, we compute the minimum cost y_v of entering each vertex $v \neq r$ and keep track of a set of edges $F^* = \{f_v : v \neq r\}$ that attain these minimums. We then set reduced edge costs c'_e. We'll label the edges in F^* in bold, the values y_v in purple, and the reduced edge costs c'_e in blue.



We have a directed cycle C in F^* indicated by the dotted circle above, so we must proceed to Step 2.2. This step tells us to contract C to a supernode to obtain a graph $G' = (V', E')$ with the reduced edge costs c'_e.



Here, we can find by inspection a minimum cost arborescence F' for G' of cost 5, highlighted in bold. Now we can extend F' to an arborescence for G by adding all but one edge in C.



Corollary 3.3 tells us that working with the reduced edge costs c'_e still gives us a minimum cost arborescence for the original edge costs c_e, so Step 1 is valid.

Moreover, note that the reduced edge costs c'_e are all nonnegative and $c'_e = 0$ for all $e \in F^*$, meaning that if F^* happened to be an arborescence rooted at r at Step 2.1, then it is an arborescence of cost 0. In particular, every other arborescence also has nonnegative cost with respect to c'_e, so F^* would be a minimum cost arborescence.

Arguing the correctness of Step 2.2 is trickier. One concern is that contracting the cycle C places extra constraints on the arborescence. A minimum cost arborescence in G' must have exactly one edge entering the contracted supernode corresponding to C. However, a minimum cost arborescence for G might have several edges entering C. To resolve this, we have the following lemma.

Lemma 3.4

Let C be a cycle consisting of edges e with $c'_e = 0$, and suppose that $r \notin C$. Then there is a minimum cost arborescence rooted at r that has exactly one edge incoming to C.

Proof of Lemma 3.4.

Let T be a minimum cost arborescence in G. By definition, there is a directed r, v-path to every vertex $v \neq r$ in T, so at least one edge enters C. If exactly one edge enters C, then we are done by taking T.

Suppose now that there are at least two edges entering C. We will construct another minimum cost arborescence T' that has exactly one edge entering C. Let (u, v) be an edge in T entering C that lies on a shortest path from r to C. Note that this path from r to C uses only one vertex in C. Delete all the edges that enter a vertex in C except for (u, v) from T, and add in edges of C except for the one that enters v.

We claim that T' is a minimum cost arborescence. Note that we are only adding edges with cost $c'_e = 0$, so $c'(T) \geq c'(T')$. By construction, T' has no directed cycles because T had no cycle before and had no cycles within C, and now only (u, v) enters C. Moreover, there is exactly one edge entering each vertex $v \neq r$ because for every vertex on C, we either did nothing or added and removed an edge entering v. By Theorem 3.1, T' is also a minimum cost arborescence with exactly one edge incoming to C. □

Another potential concern: could it be that an arborescence F' rooted at r in the graph G' doesn't lead to an arborescence F rooted at r in G? It turns out that this is impossible; we leave the proof of the following lemma as an exercise. The idea is to consider the edges in the cycle C, which we know have reduced edge costs $c'_e = 0$.

Lemma 3.5

Every arborescence F' rooted at r in the graph G' leads to an arborescence F rooted at r in G such that $c'(F') = c'(F)$.

4 Maximum Weight Independent Sets in Matroids

4.1 Matroids

A **matroid** M is a pair $(S, \mathcal{I})$ where S is a ground set and $\mathcal{I} \subseteq \mathcal{P}(S)$ is a family of sets over the ground set S satisfying the following properties:

(1) We have $\varnothing \in \mathcal{I}$.

(2) **Downward closed:** For every $A \in \mathcal{I}$ and $B \subseteq A$, we have $B \in \mathcal{I}$.

(3) **Exchange property:** For every $A, B \in \mathcal{I}$ with $|A| > |B|$, there exists $e \in A \setminus B$ such that $B \cup \{e\} \in \mathcal{I}$.

The sets in the family $\mathcal{I}$ are called the **independent sets**.

This definition is rather abstract, so we'll go over some examples.

Uniform matroids. Let $S = \{1, \dots, n\}$ and let $\mathcal{I} = \{A \subseteq S : |A| \leq k\}$ for some fixed $k \in \mathbb{Z}^+$.

(1) We have $\varnothing \in \mathcal{I}$ since $|\varnothing| = 0 \leq k$.

(2) If $A \in \mathcal{I}$ and $B \subseteq A$, then $|B| \leq |A| \leq k$, so $B \in \mathcal{I}$.

(3) For every $A, B \in \mathcal{I}$ with $|A| > |B|$, we have $A \setminus B \neq \varnothing$, so we can pick an element $e \in A \setminus B$. Note that $|B| \leq k - 1$, so $|B \cup \{e\}| \leq k$, which implies that $B \cup \{e\} \in \mathcal{I}$.

Partition matroids. Let $S = \{1, \dots, n\}$ as before, and let $S_1, \dots, S_t$ be a partition of S. That is, we have $\bigcup_{j=1}^{t} S_j = S$ and $S_i \cap S_j = \varnothing$ whenever $i \neq j$. Let $\mathcal{I} = \{A \subseteq S : |A \cap S_j| \leq r_j \text{ for all } j = 1, \dots, t\}$, where $r_j \in \mathbb{Z}^+$ are fixed. (Usually, we pick $r_j < |S_j|$, for otherwise this does not put any constraints on the independent sets.)

(1) We have $\varnothing \in \mathcal{I}$ since $|\varnothing \cap S_j| = 0 \leq r_j$ for all $j = 1, \dots, t$.

(2) If $A \in \mathcal{I}$ and $B \subseteq A$, then $|B \cap S_j| \leq |A \cap S_j| \leq r_j$ for all $j = 1, \dots, t$, so $B \in \mathcal{I}$.

(3) If $A, B \in \mathcal{I}$ with $|A| > |B|$, then there must exist some $j \in \{1, \dots, t\}$ such that $|A \cap S_j| > |B \cap S_j|$. Then $(A \cap S_j) \setminus (B \cap S_j) \neq \varnothing$, so we can find an element $e \in (A \cap S_j) \setminus (B \cap S_j)$. Observe that

$$|(B \cup \{e\}) \cap S_k| = \begin{cases} |B \cap S_k|, & \text{if } k \neq j, \\ |B \cap S_k| + 1, & \text{if } k = j. \end{cases}$$

In the first case, we have $|B \cap S_k| \leq r_k$ since $B \in \mathcal{I}$ and $e \notin S_k$. For the second case, we have $|B \cap S_j| + 1 \leq |A \cap S_j| \leq r_j$ since $A \in \mathcal{I}$ and $|A \cap S_j| > |B \cap S_j|$. It follows that $B \cup \{e\} \in \mathcal{I}$.

The following example is the setting where matroids originated from. This is the prototypical example of a matroid given in introductory talks, and is one we can build our intuition from.

Linear matroids. Let $\mathbb{F}$ be a field. (If you don't remember what a field is, it's a set together with two operations satisfying some nice axioms; for example, $\mathbb{F} = \mathbb{R}$ is a field under the usual addition and multiplication.) Let S be a set of vectors in $\mathbb{F}^d$ and let $\mathcal{I} = \{A \subseteq S : A \text{ is a linearly independent set over } \mathbb{F}\}$. Verifying that this is a matroid is an exercise in linear algebra, which we won't do here. Such a matroid is said to be representable over $\mathbb{F}$.

Graphic matroids. Given a graph $G = (V, E)$, let $S = E$, and let $\mathcal{I} = \{A \subseteq S : (V, A) \text{ is acyclic}\}$. Then $\mathcal{I}$ consists of all forests (acyclic subgraphs) of G.

(1) We see that $(V, \varnothing)$ is acyclic, so $\varnothing \in \mathcal{I}$.

(2) If $A \in \mathcal{I}$ and $B \subseteq A$, then (V, B) has no cycles since (V, A) has no cycles, so $B \in \mathcal{I}$.

(3) Let $A, B \in \mathcal{I}$ with $|A| > |B|$. Then (V, A) has $|V| - |A|$ connected components. Similarly, (V, B) has $|V| - |B|$ connected components, and we have $|V| - |B| > |V| - |A|$. Hence, there is a connected component C of (V, A) that intersects at least two connected components of (V, B), say C_1' and C_2'. Then (V, A) has a u, v-path P where $u \in C_1'$ and $v \in C_2'$. Let $e \in P \cap \delta(C_1')$, which exists because an edge must cross the boundary of the connected component to get to C_2'. Then $B \cup \{e\} \in \mathcal{I}$.

A maximal independent set $A \in \mathcal{I}$ is called a **basis**. It is important to note that maximal is different from maximum. Here, we just mean that $A \cup \{e\}$ cannot be independent for any $e \in S \setminus A$.

In the graphic matroid example, the bases correspond to the spanning trees.

Before we move on, we give a few non-examples of matroids.

(1) Let $S = \{1, 2\}$ and $\mathcal{I} = \{\{1\}, \{2\}, \{1, 2\}\}$. Then $(S, \mathcal{I})$ is not a matroid since $\varnothing \notin \mathcal{I}$. It fails the downward closed property too by considering $\varnothing \subseteq \{1\}$, where $\{1\} \in \mathcal{I}$.

(2) Let $S = \{1, 2, 3\}$ and $\mathcal{I} = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}\}$. Then $|\{1, 2\}| > |\{3\}|$, but neither $\{1, 3\}$ nor $\{2, 3\}$ are independent sets. Hence, $(S, \mathcal{I})$ fails the exchange property and is not a matroid.

Lemma 4.1

All bases for a matroid have the same cardinality.

Proof of Lemma 4.1.

Let $M = (S, \mathcal{I})$ be a matroid and let $A, B \in \mathcal{I}$ be two bases. Suppose by way of contradiction that $|A| > |B|$. Then by the exchange property, we can find $e \in A \setminus B$ such that $B \cup \{e\} \in \mathcal{I}$, contradicting the maximality of B. □

4.2 Maximum Weight Independent Sets

In the maximum weight independent sets problem, we are given a matroid $M = (S, \mathcal{I})$ and weights w_e for each $e \in S$. The goal is to find a maximum weight independent set; that is, a set $A \in \mathcal{I}$ achieving the maximum value of

$$w(A) := \sum_{e \in A} w_e.$$

Note that we may assume that $w_e > 0$ for all $e \in S$. Indeed, if $A \in \mathcal{I}$, then by the downward closed property, we have $A \setminus \{e \in S : w_e \leq 0\} \in \mathcal{I}$ as well. But

$$w(A \setminus \{e \in S : w_e \leq 0\}) \geq w(A),$$

so we can simply throw away any elements in S with $w_e \leq 0$ and still remain independent.

We now give a greedy algorithm to solve the maximum weight independent sets problem under the assumption that $w_e > 0$ for all $e \in S$. We notice that it is extremely similar to Kruskal's algorithm.

Input. A matroid $M = (S, \mathcal{I})$ and weights $w_e > 0$ for all $e \in S$.

Output. A maximum weight independent set of M.

Step 1. Sort the elements in S such that $w_{e_1} \geq w_{e_2} \geq \cdots \geq w_{e_{|S|}}$. Set $A \leftarrow \varnothing$ and $i \leftarrow 1$.

Step 2. While $i \neq |S| + 1$:

If $A \cup \{e_i\} \in \mathcal{I}$, then set $A \leftarrow A \cup \{e_i\}$.

Regardless if we add an element or not, increment $i \leftarrow i + 1$.

It is crucial that we have the assumption $w_e > 0$ for $e \in S$ here, because otherwise our greedy algorithm may keep adding negative weight elements that keep the set independent, lowering the total weight.

Proof of correctness of the greedy algorithm.

Let A denote the output the greedy algorithm. Let A^* be a maximum weight independent set of M. It is clear from the construction in Step 2 that $A \in \mathcal{I}$. Moreover, A is a basis of M because we iterate through every element of S and no element can be further included in A to keep it independent. Similarly, A^* is a basis of M. Otherwise, if $A^* \cup \{e\} \in \mathcal{I}$ for some $e \in S$, then $w(A^* \cup \{e\}) > w(A^*)$ using the assumption that $w_e > 0$, and so A^* was not a maximum weight independent set in the first place. Therefore, we have $|A| = |A^*| = k$ since bases of matroids share the same cardinality by Lemma 4.1.

Write $A = \{g_1, \dots, g_k\}$ and $A^* = \{g_1^*, \dots, g_k^*\}$ where the elements are sorted such that $w_{g_1} \geq \cdots \geq w_{g_k}$ and $w_{g_1^*} \geq \cdots \geq w_{g_k^*}$. Our aim is to show that $w(A) \geq w(A^*)$. Towards a contradiction, suppose that $w(A) < w(A^*)$ and consider the smallest $j \in \{1, \dots, k\}$ such that

$$w(\{g_1, \dots, g_j\}) < w(\{g_1^*, \dots, g_j^*\}).$$

Let $A_{j-1} = \{g_1, \dots, g_{j-1}\}$ and $A_j^* = \{g_1^*, \dots, g_j^*\}$. These are both independent by the downward closed property and we have $|A_j^*| = j > j - 1 = |A_{j-1}|$, so it follows by the exchange property that there is some $e \in A_j^* \setminus A_{j-1}$ such that $A_{j-1} \cup \{e\} \in \mathcal{I}$. Then we obtain

$$w_e \geq w_{g_j}^* > w_{g_j}.$$

The first inequality is because $e \in \{g_1^*, \dots, g_j^*\}$ and the edges are sorted by descending weight, and the second inequality is because $w(\{g_1, \dots, g_j\}) < w(\{g_1^*, \dots, g_j^*\})$ while $w(\{g_1, \dots, g_{j-1}\}) \geq w(\{g_1^*, \dots, g_{j-1}^*\})$ by the way we picked j. In particular, let's consider the iteration where e was considered for the greedy algorithm. It must have been before g_j because $w_e > w_{g_j}$. But if $B \cup \{e\} \notin \mathcal{I}$ for $B \subseteq A_{j-1}$, then $A_{j-1} \cup \{e\} \notin \mathcal{I}$ as well, which is a contradiction. □

The running time of Step 1 is $O(|S| \log |S|)$ for sorting the elements in S. In Step 2, there are $|S|$ calls to an independence oracle that tells us "yes" or "no" to whether a set is independent, but we do not necessarily know the complexity of this oracle. When we discussed Kruskal's algorithm, we could efficiently play the role as oracle by using the UnionFind data structure, but this is not always the case.

4.3 Maximum Weight Common Independent Sets

In the maximum weight *common* independent set problem, we are given two matroids $M' = (S, \mathcal{I}')$ and $M'' = (S, \mathcal{I}'')$ sharing a ground set S with weights w_e for $e \in S$. The goal is to find a set A such that $A \in \mathcal{I}'$ and $A \in \mathcal{I}''$ and achieving maximum value

$$w(A) = \sum_{e \in A} w_e.$$

We note that this problem can be solved efficiently with linear programming, but we won't state the algorithm here. The extension of this problem to even just three matroids becomes **NP**-hard. For now, we'll give a few examples of applications of the two matroid problem.

Bipartite matchings. Let $G = (V, E)$ be a bipartite graph with bipartition (U, W). Recall that a bipartite graph is such that the vertices are partitioned as $V = U \cup W$, and every edge has one endpoint in U and one endpoint in W. A **matching** is a set of edges $M \subseteq E$ such that for every $v \in V$, we have $|M \cap \delta(v)| \leq 1$. In other words, every vertex participates in at most one edge.

Suppose that for a bipartite graph, we want to find a matching $M \subseteq E$ achieving the maximum weight

$$w(M) = \sum_{e \in M} w_e.$$

We can formulate this problem as a maximum weight common independent set instance. Define matroids $M' = (E, \mathcal{I}')$ and $M'' = (E, \mathcal{I}'')$, where

$$\mathcal{I}' = \{A \subseteq E : |A \cap \delta(u)| \leq 1 \text{ for all } u \in U\},$$
$$\mathcal{I}'' = \{A \subseteq E : |A \cap \delta(w)| \leq 1 \text{ for all } w \in W\}.$$

We see that $\{\delta(u)\}_{u \in U}$ and $\{\delta(w)\}_{w \in W}$ define partitions on E, and so M' and M'' are particular examples of partition matroids. Moreover, M is a matching if and only if $M \in \mathcal{I}'$ and $M \in \mathcal{I}''$.

Arborescences. Let $G = (V, E)$ be a directed graph with edge costs $c_e \geq 0$ for all $e \in E$. Let $r \in V$ be a distinguished vertex. We can define matroids $M' = (E, \mathcal{I}')$ and $M'' = (E, \mathcal{I}'')$ where

$$\mathcal{I}' = \{A \subseteq E : (V, A) \text{ is acyclic when ignoring directions}\},$$
$$\mathcal{I}'' = \{A \subseteq E : |\delta^{\text{in}}(v) \cap A| \leq 1 - \delta_{vr} \text{ for all } v \in V\}.$$

Here, $\delta^{\text{in}}(v)$ denotes the set of edges incoming to v, and δ_{vr} is the Kronecker delta so that

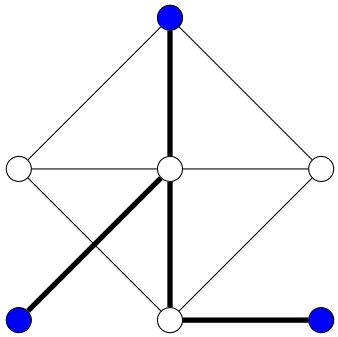
$$1 - \delta_{vr} = \begin{cases} 1, & \text{if } v \neq r, \\ 0, & \text{if } v = r. \end{cases}$$

We see that $M' = (E, \mathcal{I}')$ is a graphic matroid and $M'' = (E, \mathcal{I}'')$ is a partition matroid, with T being an arborescence rooted at r if and only if T is a basis for both M' and M''.

5 Minimum Cost Steiner Trees

5.1 Minimum Steiner Tree Problem

Given an (undirected) graph $G = (V, E)$, edge costs $c_e \geq 0$ for all $e \in E$, and **terminals** $T \subseteq V$, the goal is to find a minimum cost tree that connects all the terminals in T. For example, the vertices in blue below denote the terminals $T \subseteq V$, and the bold edges form a Steiner tree connecting the terminals in T.



Note that we have already seen some particular instances of this problem before.

- If T is empty or consists of a single vertex, then this problem is trivial; just don't take any edges.
- If $|T| = 2$, then this is the shortest path problem.
- If $T = V$, then this is the minimum spanning tree problem.

However, it turns out that this problem is **NP**-hard in general!

5.2 Minimum Metric Steiner Tree Problem

This problem is a minimum Steiner tree problem where we impose the additional conditions that

(a) $G = (V, E)$ is complete; and

(b) $c_{uw} \leq c_{uv} + c_{vw}$ for all distinct $u, v, w \in V$.

Since the edge costs satisfy the triangle inequality, we are living in a metric space in some sense, which explains the name of the problem.

It turns out that these problems are equivalent, even though the metric version looks like a much more specific case than the original version! In our reluctance to call this the MST problem due to minimum spanning trees, let's call them MSteinerT and MMSteinerT respectively.

If we have an oracle for MSteinerT, then given an instance of MMSteinerT, we can simply feed it to the oracle for MSteinerT as it solves the more general problem. The hard part of the equivalence is proving the other direction where if we have an oracle for MMSteinerT, then we can also solve MSteinerT efficiently.

Suppose we are given an instance of MSteinerT (G, c, T), where $G = (V, E)$. Consider (G', c', T'), where

- $T' = T$;
- $G' = (V, E')$ is the complete graph on the vertices of G; and
- c'_{uv} is the length of a shortest u, v-path in G with respect to edge costs c_e.

The following lemma tells us that (G', c', T') is in fact an instance of MMSteinerT.

Lemma 5.1

(a) For every Steiner tree F for the instance (G, c, T), $F' = F$ is also a Steiner tree for the instance (G', c', T') of cost $c'(F') \leq c(F)$.

(b) For any Steiner tree F' for the instance (G', c', T'), there is a Steiner tree F for the instance (G, c, T) such that $c(F) \leq c'(F')$.

(c) The new edge costs c' satisfy the triangle inequality. In particular, (G', c', T') is an instance of MMSteinerT.

Proof of Lemma 5.1.

(a) Since $E' \supseteq E$, we have that $F \subseteq E'$. Moreover, F is still a tree and connects all vertices in $T = T'$. Note that for every edge $e \in E$, we have $c'_e \leq c_e$ because taking the edge $e = uv$ is among one possibility for a u, v-path and c'_e corresponds to the length of the shortest one. This gives us $c'(F) \leq c(F)$.

(b) For each $uv \in F'$, consider a shortest u, v-path P_{uv} in the original instance (G, c, T). Let

$$K = \bigcup_{uv \in F'} P_{uv}$$

be the union of all the edges in these shortest paths. (Note that K is not necessarily a tree.) Since $c(P_{uv}) \leq c'_{uv}$ by construction, we have

$$c(K) \leq c'(F') = \sum_{uv \in F'} c'_{uv}.$$

Also, K consists only of edges in the original graph G and connects all terminals in T. Consider the subgraph with K as the edges and look at the connected component containing the terminals in T. Construct a spanning tree F on this connected component. Then $c(F) \leq c(K) \leq c'(F')$ and F connects the terminals in T, so it is a Steiner tree for (G, c, T).

(c) Let $u, v, w \in V$ be distinct. Consider a shortest u, v-path P_1, a shortest v, w-path P_2, and a shortest u, w-path P_3 in G with respect to the original costs c. By definition, we have $c'_{uv} = c(P_1)$, $c'_{vw} = c(P_2)$, and $c'_{uw} = c(P_3)$. Note that P_1 together with P_2 yields a u, w-path. Since P_3 is a shortest u, w-path, it cannot be more expensive than the aforementioned u, w-path, so we obtain

$$c'_{uv} + c'_{vw} = c(P_1) + c(P_2) \geq c(P_3) = c'_{uw},$$

so the triangle inequality holds. □

Therefore, given an MSteinerT instance, we can efficiently construct an MMSteinerT instance. Moreover, if OPT is an optimal solution for (G, c, T) and OPT' is an optimal solution for (G', c', T'), then $c(\mathsf{OPT}) \geq c'(\mathsf{OPT}')$ by part (a) of Lemma 5.1, and $c'(\mathsf{OPT}') \geq c(\mathsf{OPT})$ by part (b). This gives us $c(\mathsf{OPT}) = c'(\mathsf{OPT}')$, so given an oracle to solve MMSteinerT, we can also solve MSteinerT instances efficiently by creating a new instance of MMSteinerT and feeding it as input to the oracle.

5.3 An Approximation Algorithm for the Metric Problem

Let's now try to solve MMSteinerT for a given instance (G', c', T'); that is, G' is a complete graph and the costs c' satisfy the triangle inequality. Unfortunately, since MSteinerT is **NP**-hard, so is MMSteinerT, because these problems are equivalent! As such, we don't know how to solve it efficiently. If there is an efficient algorithm to solve it, this shows that $\mathbf{P} = \mathbf{NP}$, resolving the major million dollar problem!

Although we can't find the optimal solution efficiently, we can get an approximate solution that isn't too far from optimal in polynomial time. Consider the **induced subgraph** of G' on the vertices T', given by

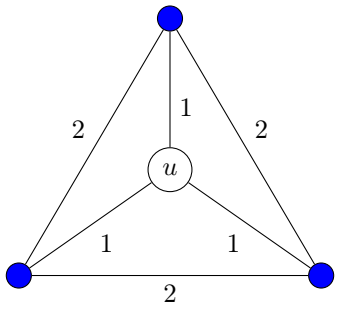
$$G'[T'] = (T', \{uv : u, v \in T' \text{ with } u \neq v\}),$$

which is the complete graph on the vertices in T'. The idea is to find a minimum spanning tree F'' in $G'[T']$ with respect to the costs c'. We have already seen that this can be done in polynomial time by using Prim's algorithm or Kruskal's algorithm. Note that F'' is a Steiner tree of (G', c', T') of cost $c'(F'')$.

We cannot hope that F'' is an optimal solution in general. For example, consider the complete graph $G' = (V', E')$, and let $u \in V'$. Set $T' = V' \setminus \{u\}$, and define edge costs by

$$c'_{vw} = \begin{cases} 1, & \text{if } v = u \text{ or } w = u, \\ 2, & \text{otherwise.} \end{cases}$$

It can be checked that these satisfy the triangle inequality. We illustrate this graph below when $|T'| = 3$.



For this graph, we see that the optimal solution has value $c(\mathsf{OPT}') = |T'|$ by taking all the edges with u as an endpoint with cost 1. On the other hand, the induced subgraph $G'[T']$ does not contain u, so a minimum spanning tree takes only the edges of cost 2, and there are $|T'| - 1$ of them. Therefore, we have

$$c'(F'') = 2(|T'| - 1) > |T'| = c(\mathsf{OPT}')$$

when $|T'| \geq 3$, and as $|T'| \to \infty$, we see that

$$\frac{c'(F'')}{c'(\mathsf{OPT}')} = \frac{2(|T'| - 1)}{|T'|} \to 2.$$

Fortunately, when using this strategy, this error is the worst it could get.

Lemma 5.2

Let F'' be a minimum spanning tree for $G'[T']$, and let OPT' be an optimal solution for (G', c', T'). Then $c'(F'') \leq 2c'(\mathsf{OPT}')$.

Proof of Lemma 5.2.

Let P be an Euler tour on the optimal solution OPT', which is a Steiner tree. That is, P is a path where the starting vertex is the same as the finish vertex, and it visits all the terminals in T'. Note that P uses every edge in OPT' exactly twice, so $c'(P) = 2c'(\mathsf{OPT}')$. Let

$$F^* := \{uv : u, v \in T' \text{ where } u \text{ and } v \text{ are consecutive terminals on the tour } P\}.$$

Then F^* is a set of edges in $G'[T']$. For every edge $uv \in F^*$, we have by the triangle inequality that

$$c'_{uv} \leq c'(P_{uv}),$$

where P_{uv} is the part of the Euler tour P between u and v. This gives

$$\sum_{e \in F^*} c'_e \leq c'(P) = 2c'(\mathsf{OPT}').$$

Note that F^* connects all the vertices in $G'[T']$ since P connects all the vertices in T'. Thus, $G'[T']$ has a spanning tree $\overline{F}$ such that $\overline{F} \subseteq F^*$, so $c'(\overline{F}) \leq 2c'(\mathsf{OPT}')$. But F'' is a minimum spanning tree of $G'[T']$, so we deduce that $c'(F'') \leq c'(\overline{F}) \leq 2c'(\mathsf{OPT}')$. □

Let $\alpha \geq 1$. An **α-approximation algorithm** for a minimization problem is an algorithm that outputs a feasible solution with value at most $\alpha \cdot z(\mathsf{OPT})$, where z denotes the objective function and OPT is an optimal solution for the minimization problem.

Similarly, for $\alpha \leq 1$, an α-approximation algorithm for a maximization problem outputs a feasible solution with value at least $\alpha \cdot z(\mathsf{OPT})$, where z is the objective function and OPT is an optimal solution for the maximization problem.

From Lemma 5.2, we see that the following is an efficient 2-approximation algorithm for MMSteinerT.

Input. A complete graph $G = (V, E)$, costs $c_e \geq 0$ for each $e \in E$ satisfying the triangle inequality, and terminals $T \subseteq V$.

Output. A minimum cost Steiner tree connecting the terminals in T.

Step 1. Construct the induced subgraph $G[T]$.

Step 2. Compute a minimum spanning tree F'' in $G[T]$ and output F''.

We will say more on approximation algorithms later. Before that, we will discuss some complexity theory in the next section.

6 Complexity Theory

6.1 The Complexity Class P

The complexity class **P** consists of all problems that can be solved efficiently (in polynomial time).

- The problem MST of finding a minimum spanning tree is in **P** using Prim's or Kruskal's algorithm.
- The problem ShortestPaths is in **P** using Dijkstra's algorithm.
- The problem MCA of finding a minimum cost arborescence is in **P** by using Edmonds' algorithm.
- However, we are unsure whether or not MSteinerT and MMSteinerT are in **P**. We'll discuss this further when we introduce the class **NP**.

Let X be a minimization problem. The **decision problem** corresponding to X takes as input an instance of X and a number k, and answers if there is a feasible solution for the given instance of value at most k.

For example, the decision version of the MSteinerT problem, which we'll call DecisionMSteinerT, takes as input a graph $G = (V, E)$, costs $c_e \geq 0$ for $e \in E$, terminals $T \subseteq V$, and a number k. To solve the problem, we should output "yes" if there exists a Steiner tree of cost at most k, and output "no" otherwise.

We'll be working with linear programs a lot later in this course. Therefore, we should note that solving a linear program is in **P**.

- It is still not known whether the simplex algorithm can be implemented to give a polynomial running time in the worst case. The hard part is the choice of the pivoting rule; it is known that Bland's rule, which ensures that the simplex algorithm never cycles, is quite inefficient.
- There are other algorithms that can solve linear programs in polynomial time. The first one to be discovered was the ellipsoid method, but it is also inefficient in practice.

6.2 Polynomial Time Reductions

We have seen a glimpse of reductions when we discussed MSteinerT and MMSteinerT in Section 5.2. Given two problems X and Y, we say that X **reduces** in polynomial time to Y, denoted by $X \leq_P Y$, if one can solve X by using a polynomial number of basic operations and a polynomial number of calls to an oracle that solves Y. Intuitively, this means that X is "not harder" than Y, because if we know how to solve Y, then we have a way to solve X. We look at some examples of reductions.

- We can show that MST $\leq_P$ MSteinerT. Suppose we are given an instance of MST, namely a graph $G = (V, E)$ and edge costs c_e for each $e \in E$. Then we can construct an instance of MSteinerT by simply setting $T = V$. We can pass (G, c, T) to an oracle that solves MSteinerT and output that result. Overall, this process involved a polynomial number of operations (assigning $T = V$), and then a polynomial number of calls to the oracle (only one call was needed here).
- Next, let's show that MST $\leq_P$ MCA. As before, suppose we are given an instance of MST, so a graph $G = (V, E)$ and edge costs c_e for each $e \in E$. We now construct an instance of MCA as follows. Let r be an arbitrary node in V. Let $G' = (V', E')$ where $V' = V$ and

$$E' = \{(u, v) : uv \in E\} \cup \{(v, u) : uv \in E\}.$$

  Set edge costs $c'_{(u,v)} = c'_{(v,u)} = c_{uv}$ for all $uv \in E$. Feed this instance (G', c', r) of MCA to the oracle that solves it and output the obtained arborescence, ignoring directions.

However, these reductions were a bit silly. We already know that we can solve MST in polynomial time; we didn't even need the oracles for the other problems! In particular, any problem in **P** reduces to any other problem simply by using the polynomial time algorithm that solves it.

Suppose that X and Y are problems such that $X \leq_P Y$. We make two easy observations:

- If $Y \in \mathbf{P}$, then $X \in \mathbf{P}$ because adding a polynomial to a product of polynomials is still a polynomial.
- If $X \notin \mathbf{P}$, then $Y \notin \mathbf{P}$. This is just the contrapositive of the previous point, but this statement is slightly more interesting.

Moreover, if $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$ since the composition of polynomials is a polynomial.

6.3 The Complexity Class NP

Let X be a decision problem. An **efficient certifier** B for the problem X is an algorithm with polynomial running time which takes two inputs s and t, where s is an instance of X and t is a **certificate**.

- If s is a "yes" instance of X, then there exists a certificate t whose size is polynomial in the size of s such that $B(s,t)$ returns "yes".
- If s is a "no" instance of X, then $B(s,t)$ returns "no" for all certificates t.

Recall that the DecisionMSteinerT problem takes as input a graph $G = (V, E)$, costs $c_e \geq 0$ for $e \in E$, terminals $T \subseteq V$, and a number k. This is a "yes" instance if there exists a Steiner tree of cost at most k, and a "no" instance otherwise.

For this problem, a certificate t is an edge set F, and the efficient certifier outputs "yes" if F is a Steiner tree with $c(F) \leq k$, and outputs "no" otherwise. For checking that F is a Steiner tree, it needs to verify that the edges form a tree and the terminals are all connected, both of which can be done in polynomial time.

The complexity class **NP** is the class of all decision problems that admit an efficient certifier. In other words, **NP** consists of problems where "yes" instances can be verified efficiently with a "short" certificate.

Clearly, if a decision problem is in **P**, then it is in **NP**. The verifier can use the polynomial time algorithm that solves the decision problem and completely ignore the certificate. Conversely, it is not known whether every decision problem in **NP** also lies in **P**. This is the famous $\mathbf{P} = \mathbf{NP}$ problem.

6.4 NP-hardness and NP-completeness

A problem X is said to be **NP-hard** if for every problem $Y \in \mathbf{NP}$, we have $Y \leq_P X$. Furthermore, we say that X is **NP-complete** if $X \in \mathbf{NP}$ and X is **NP**-hard. We can think of **NP**-complete problems as the "hardest problems in **NP**".

In the 3-SAT problem, we are given boolean variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$ that are disjunctions of the form $t_1 \vee t_2 \vee t_3$ where $t_1, t_2, t_3 \in \{x_1, \ldots, x_n\} \cup \{\overline{x_1}, \ldots, \overline{x_n}\}$. The goal is to determine if there is a truth assignment to the boolean variables $x_1, \ldots, x_n$ such that $C_1 \wedge C_2 \wedge \cdots \wedge C_k$ is satisfied.

For example, given the variables x_1, x_2, x_3, x_4 and the clauses

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_3} \vee \overline{x_4}),$$

we see that the assignment $x_1 = x_2 = \mathsf{False}$ and $x_3 = x_4 = \mathsf{True}$ does the trick.

The 3-SAT problem is one of the most famous in computer science because it is one of the first problems shown to be **NP**-complete, due to Cook and Levin in 1973.

Theorem 6.1: Cook-Levin

3-SAT is **NP**-complete.

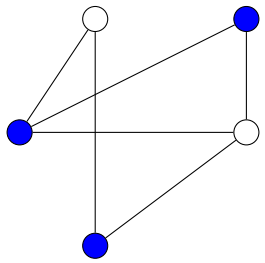
There has been a lot of growth in computational complexity theory in the past few decades, so we now have a large variety of **NP**-complete problems to work with. To prove that a given problem X is **NP**-complete, it suffices to do the following:

(i) Show that $X \in \mathbf{NP}$. This part is typically easy to prove.

(ii) Find another **NP**-complete problem Y and show that $Y \leq_P X$. This will imply that X is **NP**-hard because we have $Z \leq_P Y$ for all $Z \in \mathbf{NP}$ and hence $Z \leq_P X$ by transitivity.

We now take a look at the decision vertex cover problem, which we denote by DecVC. Here, we are given a graph $G = (V, E)$ and a number ℓ. The goal is to determine whether if there is a vertex set $U \subseteq V$ such that $|U| \leq \ell$ and every edge $e \in E$ has at least one endpoint in U.

For example, the following graph has the vertex cover in blue with $\ell = 3$.



On the other hand, the complete graph K_5 has no vertex cover with size at most $\ell = 3$.

Proposition 6.2

DecVC is **NP**-complete.

Proof of Proposition 6.2.

We use the recipe that we supplied above. First, we show that DecVC $\in$ **NP**. Consider a subset of vertices $U \subseteq V$ to be our certificate. It is easy to verify that $|U| \leq \ell$. We can iterate through the edge set and determine if one of the endpoints is in U. Overall, the certifier can be implemented in linear time.

Now, we reduce a known **NP**-complete problem to DecVC. Here, we only have 3-SAT at our disposal, so let's show that 3-SAT $\leq_p$ DecVC.

Suppose that are are given an instance of 3-SAT. In particular, we have variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$ of the form $t_1 \vee t_2 \vee t_3$ where $t_1, t_2, t_3 \in \{x_1, \ldots, x_n\} \cup \{\overline{x_1}, \ldots, \overline{x_n}\}$. We construct an instance of DecVC as follows:

- Let's start with the graph $G = (V, E)$. For every clause $C_i = t_1 \vee t_2 \vee t_3$ where $i = 1, \ldots, k$, introduce the vertices (t_1, i), (t_2, i), and (t_3, i). Add edges between each pair of vertices to form a triangle.

  Next, for each variable x_j appearing in clause C_{i_1} as x_j and in clause C_{i_2} as $\overline{x_j}$ where $i_1 \neq i_2$, include an edge between (x_j, i_1) and $(\overline{x_j}, i_2)$.

- Let $\ell = 2k$ be twice the number of clauses.

Next, we show that the original 3-SAT instance is a "yes" instance if and only if the constructed DecVC instance is a "yes" instance. In particular, if we have an oracle that solves DecVC, then we can also solve 3-SAT efficiently.

($\Rightarrow$) Suppose that the original 3-SAT instance is a "yes" instance. Then there is some truth assignment v to the variables $x_1, \ldots, x_n$ such that $C_1 \wedge \cdots \wedge C_k$ is satisfied. For each clause $C_i = t_1 \vee t_2 \vee t_3$, there exists $t \in \{t_1, t_2, t_3\}$ such that $t = \mathsf{True}$ for the assignment v. Let y_i denote this t for all $i = 1, \ldots, k$.

Let $U = V \setminus \{(y_i, i) : i = 1, \dots, k\}$ and observe that $|U| = |V| - k = 3k - k = 2k = \ell$. Moreover, we claim that U is a vertex cover for $G = (V, E)$. Suppose otherwise, so there is some $e \in E$ with both endpoints not in U. By construction, these endpoints are (y_i, i) and (y_j, j) for some $i \neq j$. It is clear that this edge is not one from the triangle between (t_1, i), (t_2, i) and (t_3, i) for a clause $C_i = t_1 \vee t_2 \vee t_3$ since $i \neq j$. Moreover, it cannot be an edge between (x_t, i) and $(\overline{x_t}, j)$ for some variable x_t since v would not be a valid truth assignment. Therefore, both cases are impossible, so U must be a vertex cover. That is, our constructed DecVC instance is a "yes" instance.

($\Leftarrow$) Suppose now that our constructed DecVC instance is a "yes" instance. Let U be a vertex cover in G such that $|U| \leq \ell = 2k$. By construction of the graph $G = (V, E)$, it must be the case that $|U| = 2k$ because for each clause $C_i = t_1 \vee t_2 \vee t_3$, the vertex cover must take two out of three of the vertices (t_1, i), (t_2, i), and (t_3, i).

Let (y_i, i) be the vertex that is not in U from $y_i \in \{t_1, t_2, t_3\}$. Define an truth assignment v to $x_1, \dots, x_n$ such that $y_i = \mathsf{True}$ for all $i = 1, \dots, k$. Suppose that v is not well-defined. That is, there is a variable x_t which needs to be assigned both True and False. Then we have $y_i = x_t$ and $y_j = \overline{x_t}$ for some $i \neq j$. By construction of $G = (V, E)$, there is an edge between (y_i, i) and (y_j, j). Neither of these endpoints are in U, which contradicts the fact that U is a vertex cover.

Therefore, the truth assignment v is well-defined, and it corresponds to satisfying each clause C_i. This means that the original 3-SAT instance is a "yes" instance, as desired. □

In Section 5, we claimed that the Steiner tree problem was **NP**-hard. Let's now prove this. First, we state the decision Steiner tree problem, denoted DecSteinerTree. We are given a graph $G = (V, E)$, terminals $T \subseteq V$, edge costs $c_e \geq 0$ for each $e \in E$, and a number k. The goal is to determine whether there is a Steiner tree $F \subseteq E$ such that $c(F) \leq k$.

Proposition 6.3

DecSteinerTree is **NP**-complete.

Proof of Proposition 6.3.

We see that DecSteinerTree $\in$ **NP** by taking a subset $F \subseteq E$ as a certificate. It is easy to check that $c(F) \leq k$ and that there is a u, v-path in F for every $u, v \in T$ in polynomial time.

Next, we show that DecVC $\leq_p$ DecSteinerTree, where we know that DecVC is **NP**-complete from Proposition 6.2. Suppose that we are given a graph $G = (V, E)$ and a number ℓ from DecVC. We construct a DecSteinerTree instance as follows:

- Let $G' = (V', E')$ where $V' = \{r\} \cup V \cup \{t_e : e \in E\}$ and

$$E' = \{rv : v \in V\} \cup \{vt_e : e \in E \text{ and } v \text{ is an endpoint of } e\}.$$

  In particular, we can view the graph G' as one consisting of three "layers": one with the new vertex r at the top, the original vertices V in the middle, and new vertices t_e corresponding to each edge $e \in E$ at the bottom. There is an edge from r to each original vertex $v \in V$, and an edge from each $v \in V$ to t_e if v is an endpoint of e.

- The terminals will be $T = \{r\} \cup \{t_e : e \in E\}$; namely, the edges in the top and bottom layers.

- We assign edge costs via

$$c_e = \begin{cases} 100|V|, & \text{if } e \in \{vt_e : e \in E \text{ and } v \text{ is an endpoint of } e\}, \\ 1, & \text{otherwise.} \end{cases}$$

- Finally, we set $k = 100|V||E| + \ell$.

We claim that the original DecVC instance is a "yes" instance if and only if the DecSteinerTree instance is a "yes" instance.

($\Rightarrow$) Suppose that the original DecVC instance is a "yes" instance. Then there is a vertex cover $U \subseteq V$ of $G = (V, E)$ such that $|U| \leq \ell$. For each $e \in E$, let $u_e \in U$ be a vertex such that $e \in \delta(u_e)$ (when both endpoints are in U, we can arbitrarily pick one).

Consider the set of edges $F := \{ru : u \in U\} \cup \{t_e u_e : e \in E\} \subseteq E'$. Note that this has cost $c(F) = |U| + 100|V||E| \leq \ell + 100|V||E| = k$ and that F is a Steiner tree of $G' = (V', E')$ with respect to the terminals $T = \{r\} \cup \{t_e : e \in E\}$, so the DecSteinerTree instance is a "yes" instance.

($\Leftarrow$) Suppose that the DecSteinerTree instance is a "yes" instance. Moreover, towards a contradiction, assume that the original DecVC instance is a "no" instance. There exists a Steiner tree $F \subseteq E'$ for $G' = (V', E')$ with respect to T such that $c(F) \leq k = \ell + 100|V||E|$.

Let $\overline{E} = \{vt_e : e \in E \text{ and } v \text{ is an endpoint of } e\}$ and observe that

$$c(F \cap \overline{E}) \geq 100|V||E|$$

since $100|V|$ is the cost of each edge incident to t_e for $e \in E$, and the number of terminals is $|E|$. This bound together with $c(F) \leq \ell + 100|V||E|$ implies that $c(F \setminus \overline{E}) \leq \ell$.

We may assume that $\ell \leq |V|$, for otherwise DecVC is always a "yes" instance. Every vertex t_e corresponding to $e \in E$ is incident to exactly one edge of F, because adding even one extra edge incident to t_e leads to

$$c(F) \geq 100|V||E| + 100|V| > 100|V||E| + \ell = k,$$

which is a contradiction. Construct the set

$$U = \{v \in V : v \text{ is an endpoint of an edge in } F \setminus \overline{E}\}.$$

Note that $|U| \leq c(F \setminus \overline{E}) \leq \ell$ since each edge from $F \setminus \overline{E}$ has cost 1. Moreover, U is a vertex cover for the original DecVC instance. Otherwise, there would exist some terminal t_e such that t_e is adjacent only to u_e in F. But u_e is only adjacent to terminals of the form t_g where $g \in E$. This implies that u_e is not connected to r, which is a contradiction. □