# CO 353: Final Exam Proofs

## 1. Shortest Paths

In this problem, we are given a directed graph $G = (V, E)$ and edge lengths $\ell_e \geq 0$ for all $e \in E$. Given a start vertex $s \in V$, the goal is to find the shortest path from $s$ to $v$ for all vertices $v \in V$.

**Dijkstra's algorithm.** Let $d(v)$ denote the length of a shortest path from $s$ to $v$. Let $d'(v)$ be upper bounds on the length of a shortest path from $s$ to $v$, and let $A$ be the set of visited vertices.

1. Initialize $A = \{s\}$ and $d(s) = 0$. For all $v \in V \setminus A$, let $d'(v) = \infty$.
2. While $A \neq V$, do the following:
   (a) For all $v \in V \setminus A$, set
   $$d'(v) = \min_{\substack{u \in A \\ (u,v) \in E}} \left\{ d'(v), d(u) + \ell_{(u,v)} \right\}.$$
   (b) Let $w = \operatorname{argmin}_{v \in V \setminus A} d'(v)$. Set $A = A \cup \{w\}$ and $d(w) = d'(w)$.

**Proof of correctness.** It is enough to show that the length of a shortest path from $s$ to $v$ is correctly computed for all $v \in V$. We proceed by induction on $|A|$. For $|A| = 1$, we have $d(s) = 0$ as expected.

Suppose that the result holds for $|A|$ at a given point in the algorithm, and that $w$ is the vertex that is being added to $A$. Let $u \in A$ be the vertex that is determining the upper bound for $w$; that is,

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

We know that $d(u)$ is correctly computed by induction since $u \in A$. Suppose towards a contradiction that $d'(w)$ is not correctly computed. Let $P_u$ be a shortest path from $s$ to $u$, and let $P'$ be a shortest path from $s$ to $w$. Then we have that

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

Note that a shortest path from $s$ to $w$ contains an edge $(x, y)$ such that $x \in A$ and $y \in V \setminus A$. Then $d(x)$ is correctly computed, and we obtain

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

where the second inequality uses the fact that a shortest path from $s$ to $w$ containing $(x, y)$ must use a shortest path from $s$ to $x$ followed by $(x, y)$. In particular, since $d'(y) < d'(w)$, this means that Dijkstra's algorithm should have chosen $y$ instead of $w$, which is a contradiction. □

## 2. Minimum Spanning Trees

A **tree** is a connected acyclic graph. A **spanning tree** of a graph $G = (V, E)$ is a subgraph $T = (V, F)$ of $G$ such that $F \subseteq E$ and $T$ is a tree.

In the minimum spanning tree problem, we are given a connected graph $G = (V, E)$ and costs $c_e$ for all $e \in E$. We want to find a spanning tree in $G$ of minimum cost.

First, we state the fundamental theorem of trees from MATH 239.

**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 $|F| = |V| - 1$.

Consider the following three properties: (1) $T$ is connected; (2) $T$ is acyclic; (3) $|F| = |V| - 1$. Then by the fundamental theorem of trees, knowing that two of the three properties hold guarantees that the remaining one holds as well.

The following are two useful properties for determining if an edge is contained in a minimum spanning tree or not. We will make extensive use of the cut property in our algorithms.

**Cut property.** Suppose that the costs $c_e$ are distinct. Let $S \subseteq V$ such that $S \neq \varnothing$ and $S \neq V$, and let

$$e = \operatorname*{argmin}_{f \in \delta(S)} c_f.$$

Then $e$ is contained in every minimum spanning tree of $G$.

**Proof.** We proceed by contradiction. Suppose that $T = (V, F)$ is a minimum spanning tree of $G$ such that $e \notin F$. Consider the graph $(V, F \cup \{e\})$. Then $|F \cup \{e\}| = |V|$ and this graph is connected, so it cannot be a tree by the fundamental theorem of trees. In particular, it must contain some cycle $C$, and we have $e \in C$ since $T = (V, F)$ was acyclic.

Note that $|C \cap \delta(S)|$ must be even because for every edge leaving the cut $\delta(S)$, there must be some edge coming back into the cut. Moreover, since $e \in C \cap \delta(S)$, this implies that $|C \cap \delta(S)| \geq 2$. Then there must be some other edge $e' \in C \cap \delta(S)$ distinct from $e$.

Now, consider $T' = (V, (F \cup \{e\}) \setminus \{e'\})$. We see that from the fundamental theorem of trees that $T'$ is a tree. Indeed, we have that $|(F \cup \{e\}) \setminus \{e'\}| = |V| - 1$, and $T'$ is still connected because if a path between two vertices used the edge $e'$, then we can go along the edges in $C \setminus \{e'\}$ instead. Finally, notice that

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

since the edge costs are distinct with $e = \operatorname{argmin}_{f \in \delta(S)} c_f$ and $e' \in \delta(S)$. This implies that $T$ is not a minimum spanning tree, which is a contradiction. □

**Cycle property.** Suppose that the costs $c_e$ are distinct. Let $C$ be a cycle in $G$ and let

$$e = \operatorname*{argmax}_{f \in C} c_f.$$

Then $e$ is not contained in any minimum spanning tree of $G$.

**Proof.** Let $T = (V, F)$ be a spanning tree containing $e$, and suppose that the endpoints of $e$ are $v$ and $w$. We show that $T$ does not have the smallest possible cost. Similar to the proof of the cut property, we do this by exchanging $e$ with an edge $e'$ of cheaper cost in a way that we still obtain a spanning tree.

By deleting $e$ from $T$, we obtain a graph with two components: $S$ containing $v$ and $V \setminus S$ containing $w$. The edge we are replacing $e$ with should have one end in $S$ and the other in $V \setminus S$ to connect the components back together.

To find such an edge, we follow the cycle $C$. Note that the edges in $C \setminus \{e\}$ together form a path $P$ from $v$ to $w$. If we follow $P$ from $v$ to $w$, we will begin at $S$ and end in $V \setminus S$, so there is some edge $e'$ in $P$ that crosses from $S$ to $V \setminus S$.

Let $T' = (V, (F \cup \{e'\}) \setminus \{e\})$. Note that $T'$ has $|V| - 1$ edges and is connected by our above argument, so it is a tree by the fundamental theorem of trees. Moreover, we have $c_{e'} < c_e$ since $e$ is the edge of maximum cost in $C$ with $e' \in C$ so that

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

Then $c(T) > c(T')$, so $T'$ is a spanning tree of $G$ with smaller cost than $T$. It follows that $T$ is not a minimum spanning tree. □

Now, we discuss two efficient algorithms that solve the minimum spanning tree problem.

**Prim's algorithm.** The main idea is to use the cut property to construct a minimum spanning tree starting from an arbitrary vertex $s \in V$. At each iteration, we keep track of a partial tree construction $T$ and a set of vertices $A \subseteq V$ that are connected to $s$ in $T$. We stop the algorithm once we have reached all the vertices in $G$.

1. Let $s \in V$ be an arbitrary vertex. Set $A = \{s\}$ and $T = \varnothing$.
2. While $A \neq V$, do the following:
   (a) Set $e = \operatorname{argmin}_{f \in \delta(A)} c_f$ where $e = uv$ with $u \in A$ and $v \notin A$.
   (b) Set $A = A \cup \{v\}$ and $T = T \cup \{e\}$.

**Proof of correctness.** We assume that the edge costs $c_e$ are distinct. At each iteration of Step 2, we add one edge to $T$, so $|T| = |V| - 1$ since there are $|V| - 1$ iterations. Moreover, $T$ is connected because an invariant of the algorithm is that all vertices are connected to $s$. It follows by the fundamental theorem of trees that $T$ is a tree. Finally, by the cut property and our assumption that the edge costs are distinct, we see that $T$ contains only the edges that are contained in every minimum spanning tree, so $T$ itself is a minimum spanning tree. □

A consequence is that a graph with distinct edge costs $c_e$ has a unique minimum spanning tree.

**Kruskal's algorithm.** This is a greedy algorithm that 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.

1. Sort the edges such that $c_{e_1} \leq c_{e_2} \leq \cdots \leq c_{e_m}$. Set $T = \varnothing$ and $j = 1$.
2. While $j < m + 1$, do the following:
   (a) If $T \cup \{e_j\}$ is acyclic, then set $T = T \cup \{e_j\}$.
   (b) Increment $j = j + 1$.

**Proof of correctness.** We again assume that the edge costs are distinct. Let $T$ be the output set consisting of edges. We know that $T$ is acyclic by construction as Step 2 of the algorithm rules out all edges that create a cycle. We show that $(V, T)$ is connected by way of contradiction so that it is a spanning tree by the fundamental theorem of trees.

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

Next, we show that we have a minimum spanning tree. Consider the moment an arbitrary edge $e = uv$ was added to $T$, and let $T'$ be the set of edges in $T$ just before the addition of $e$ to $T$. Note that $T'$ has no $u, v$-path, for otherwise such a path together with $e = uv$ would form a cycle. Hence, 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 $e = \text{argmin}_{f \in \delta(S)} c_f$. By the cut property, $e$ is contained a minimum spanning tree. Then this holds for all edges $e \in T$, implying that $T$ is itself a minimum spanning tree. □

**Clusterings of maximum spacing.** We are given a set $U = \{p_1, \ldots, p_n\}$ of $n$ objects and a distance $d(p_i, p_j)$ between points. We require that $d(p_i, p_i) = 0$ and $d(p_i, p_j) = d(p_j, p_i) > 0$ for distinct $p_i$ and $p_j$. A **$k$-clustering** of $U$ is a partition of $U$ into $k$ nonempty sets $C_1, \ldots, C_k$. The **spacing** of a $k$-clustering is the minimum distance between any pair of points lying in different clusters. We want points in different clusters to be far apart from each other, so a natural goal is to seek the $k$-clustering with the maximum possible spacing. That is, over all $k$-clusterings $C_1, \ldots, C_k$ of $U$, we want to maximize

$$\min_{1 \le i < j \le k} \left\{ \min_{p \in C_i, q \in C_j} d(p, q) \right\}.$$

**Single linkage clustering algorithm.** Consider the complete graph $K_n$ on the vertex set $U$ with the edge costs given by the distances. The connected components will be the clusters. We want to bring nearby points together into the same cluster as rapidly as possible so that they don't end up in different clusters that are close together. We see that if we sort by ascending distances and skip over edges where both endpoints are already in the same cluster, then we avoid cycles and end up with a union of trees.

Note that this procedure is precisely Kruskal's algorithm. The only difference is that we are looking for a $k$-clustering, so we stop once we hit $k$ connected components. That is, we are running Kruskal's algorithm but stopping it before it adds the last $k-1$ edges. This is equivalent to taking the full minimum spanning tree and deleting the $k-1$ most expensive edges and defining the $k$-clustering to be the resulting connected components $C_1, \ldots, C_k$. We show that this is a $k$-clustering of maximum spacing.

**Proof of correctness.** Let $\mathcal{C}$ denote the clustering $C_1, \ldots, C_k$. The spacing of $\mathcal{C}$ is the length $d^*$ of the $(k-1)$-th most expensive edge in the minimum spanning tree; this is the length of the edge that Kruskal's algorithm would have added next at the moment we stopped it.

Let $\mathcal{C}'$ be another $k$-clustering which partitions $U$ into nonempty sets $C'_1, \ldots, C'_k$. We show that the spacing of $\mathcal{C}'$ is at most $d^*$. Since $\mathcal{C}$ and $\mathcal{C}'$ are not equal, it must be that one of the clusters $C_r$ is not a subset of any of the $k$ sets $C'_s$ in $\mathcal{C}'$. Hence, there are points $p_i, p_j \in C_r$ that belong to different clusters in $\mathcal{C}'$, say $p_i \in C'_s$ and $p_j \in C'_t$ with $C'_s \neq C'_t$.

Since $p_i$ and $p_j$ are in the same component $C_r$, it must be that Kruskal's algorithm added all the edges of a $p_i, p_j$-path $P$ before we stopped it. In particular, each edge in $P$ has length at most $d^*$. But $p_i \in C'_s$ and $p_j \notin C'_t$, so let $p'$ be the first node on $P$ that does not belong to $C'_s$ and let $p$ be the node on $P$ that comes just before $p'$. Then $d(p, p') \le d^*$ since the edge from $p$ to $p'$ was added by Kruskal's algorithm. But $p$ and $p'$ belong to different clusters in $\mathcal{C}'$, so the spacing of $\mathcal{C}'$ is at most $d^*$, as desired. □

## 3. Minimum Cost Arborescences

Let $G = (V, E)$ be a directed graph and let $r \in V$ be a distinguished node which we call a **root**. An **arborescence** rooted at $r$ is a subgraph $T = (V, F)$ such that:

(i) $T$ is a spanning tree of $G$ if we ignore the direction of edges;

(ii) there is a path in $T$ from $r$ to every other node $v \in V$ if we take the direction of edges into account.

In other words, an arborescence rooted at $r$ is essentially a directed spanning tree rooted at $r$.

**Characterization of arborescences.** Let $G = (V, E)$ be a directed graph and let $r \in V$. A subgraph $T = (V, F)$ of $G$ is an arborescence rooted at $r$ if and only if

(1) $T$ has no cycles; and

(2) every node $v \neq r$ has exactly one edge entering it.

**Proof.** ($\Rightarrow$) Let $T$ be an arborescence rooted at $r$. Then after ignoring all directions, $T$ is a spanning tree. In particular, $T$ has no cycles. Moreover, there is a unique path from $r$ to $v$, and the last edge on this path must be incoming for $v$.

($\Leftarrow$) Suppose that $T$ has no cycles and every node $v \neq r$ has exactly one entering edge. To show that $T$ is an arborescence rooted at $r$, we first verify that for all $v \neq r$, there is a directed path from $r$ to $v$ in $T$. Consider the unique edge $(v_1, v)$ incoming to $v$, and repeatedly follow edges backwards in this way.

Note that $r$ is the only vertex that does not have any incoming edges. Indeed, if it did have an incoming edge, say $(v, r)$, then we could continually follow the edges backwards as above and eventually obtain a cycle, which is a contradiction. Therefore, the above process must terminate at $r$ since $T$ has no cycles and $r$ is the only vertex without an incoming edge. This gives us a directed path from $r$ to $v$.

We now verify the other condition that $T$ is a spanning tree of $G$ ignoring directions. Since we can get from $r$ to any other vertex $v \neq r$, we see that $T$ is connected when ignoring directions. Moreover, $r$ has no incoming edges. Since every edge is incoming for exactly one of its endpoints, the total number of edges in $T$ is $|V| - 1$, implying that $T$ is a spanning tree of $G$ after ignoring directions by the fundamental theorem of trees. □

**Characterization of graphs with arborescences.** A directed graph $G = (V, E)$ has an arborescence rooted at $r \in V$ if and only if there is a directed path from $r$ to every other node.

**Proof.** ($\Rightarrow$) Suppose that for some $v \neq r$, there is no directed path from $r$ to $v$ in $G$. Then no subgraph could possibly have a directed path from $r$ to $v$; in particular, no arborescence rooted at $r$ exists.

($\Leftarrow$) For each $v \neq r$, suppose that there is a directed path $P_v$ from $r$ to $v$. Then a breadth-first search tree rooted at $r$ forms an arborescence rooted at $r$. □

In the minimum cost arborescence problem, we are given a directed graph $G = (V, E)$, a distinguished root node $r \in V$, and edge costs $c_e \geq 0$ for all $e \in E$. The goal is to find an arborescence rooted at $r$ so that the total cost is minimized. We assume throughout that $G = (V, E)$ has an arborescence rooted at $r$ as this can be easily verified using the above characterization.

We make the observation that every arborescence contains exactly one edge entering $v \neq r$, so if we pick some node $v$ and subtract a uniform quantity from the cost of every edge entering $v$, then the total cost of every arborescence changes by the exact same amount. Therefore, the actual cost of the cheapest edge entering $v$ is not important; what matters is the cost of all other edges entering $v$ *relative* to this.

Due to this reasoning, we will define $y_v$ to be the minimum cost of an edge entering $v$. Let $f_v$ be an arbitrary edge achieving this minimum, and let $F^*$ be this set of $n - 1$ edges. For each edge $(u, v) \in E$, we define its modified cost to be $c'_{(u,v)} = c_{(u,v)} - y_v$. Since $c_{(u,v)} \geq y_v$ for all $(u, v) \in E$, it follows that all the modified costs are still nonnegative.

It is possible that $(V, F^*)$ is not an arborescence. However, since $F^*$ has an edge incoming to every node $v \neq r$, the arborescence characterization implies that $(V, F^*)$ contains a cycle $C$. The above discussion motivates the following crucial fact.

**Relationship between original and modified costs.** An arborescence $T$ rooted at $r$ is of minimum cost subject to edge costs $c_e$ if and only if it is of minimum cost subject to the modified edge costs $c'_e$.

**Proof.** Let $T$ be an arbitrary arborescence rooted at $r$. Note that the difference between its cost with respect to $c_e$ and its cost with respect to $c'_e$ is exactly

$$\sum_{e \in T} c_e - \sum_{e \in T} c'_e = \sum_{v \neq r} y_v$$

since an arborescence has exactly one edge entering $v \neq r$ in the sum. But this value is independent of the choice of arborescence $T$, implying that $T$ has minimum cost with respect to $c_e$ if and only if it has minimum cost with respect to $c'_e$. □

We now consider the problem with respect to the modified costs $c'_e$. All the edges in the set $F^*$ now have cost 0 under the modified costs, so if $(V, F^*)$ contains a cycle $C$, then we know that all the edges in $C$ have cost 0. Then we can afford to use as many edges from $C$ as we want since including edges from $C$ doesn't raise the cost.

Therefore, our approach will be to contract $C$ into a *supernode* to obtain a smaller graph $G' = (V', E')$. Note that $V'$ contains the nodes of $V \setminus C$ together with a single node $c^*$ representing $C$. We transform each edge $e \in E$ to an edge $e' \in E'$ by replacing each end of $e$ that belongs to $C$ with the new node $c^*$. This can result in $G'$ having parallel edges (i.e. edges with the same endpoints), but we delete all self-loops, namely edges that have $c^*$ as both its endpoints. We recursively find an arborescence of minimum cost in this smaller graph $G'$ subject to the costs $c'_e$. The arborescence returned by this recursive call can be converted to an arborescence of $G$ by including all but one edge in the cycle $C$.

**Edmonds' algorithm.** The above procedure is precisely Edmonds' algorithm. We summarize it below.

1. For all $v \neq r$:
   (a) Let $y_v$ be the minimum cost of an edge entering node $v$.
   (b) Modify the costs of all edges $(u, v) \in E$ to be $c'_{(u,v)} = c_{(u,v)} - y_v$.
   (c) Let $f_v$ be an arbitrary edge entering $v$ of modified cost 0.

   Let $F^* = \{f_v : v \neq r\}$ be the set of chosen edges of modified cost 0.
2. If $F^*$ is an arborescence rooted at $r$, then return it.
3. Otherwise, there is a directed cycle $C \subseteq F^*$.
   (a) Contract $C$ to a single supernode to obtain a graph $G' = (V', E')$.
   (b) Recursively find an arborescence $(V', F')$ of minimum cost in $G'$ with respect to costs $c'_e$.
   (c) Extend $(V', F')$ to an arborescence $(V, F)$ in $G$ by adding all but one edge of $C$.

Does this lead to an arborescence of minimum cost? One thing we need to worry about is that not every arborescence in $G$ corresponds to an arborescence in the contracted graph $G'$. Could we "miss" the arborescence of minimum cost in $G$ by focusing on $G'$?

What we do know is that the arborescences of $G'$ are in one-to-one correspondence with the arborescences of $G$ that have exactly one edge entering the cycle $C$. (By an edge entering the cycle, we mean an edge $e = (u, v)$ such that $v \in C$ but $u \notin C$.) These corresponding arborescences have the same cost with respect to $c'_e$ since $C$ consists of edges of cost 0. Therefore, to prove the correctness of the algorithm, we need to show that $G$ has an arborescence of minimum cost with exactly one edge entering $C$. This is what we'll do next.

**Existence of minimum cost arborescence with exactly one edge entering the 0-cost cycle.** Let $C$ be a cycle in $G$ consisting of edges of cost 0 such that $r \notin C$. Then there is an arborescence rooted at $r$ of minimum cost that has exactly one edge entering $C$.

**Proof.** Consider an arborescence $T$ of minimum cost in $G$. Since $r$ has a path in $T$ to every node, there is at least one edge that enters $C$. If $T$ enters exactly once, we are done.

Suppose now that there are at least two edges entering $C$. We will construct another minimum cost arborescence $T'$ that has 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 $C$, namely the last one. 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. In our construction, we are only adding edges with cost $c'_e = 0$, so $c'(T) \geq c'(T')$. Moreover, $T'$ has no cycles because $T$ previously had no cycles and now only $(u, v)$ enters $C$. Moreover, there is exactly one edge entering each vertex $v \neq r$ because for each vertex on $C$, we either did nothing or added and removed an edge entering $v$. By the characterization of arborescences, it follows that $T'$ is a minimum cost arborescence with exactly one edge in $C'$. □

Putting everything together, we now argue the correctness of Edmonds' algorithm.

**Proof of correctness of Edmonds' algorithm.** We proceed by induction on the number of nodes in $G$. If the edges of $F^*$ form an arborescence, then the algorithm returns a minimum cost arborescence since $F^*$ represents the cheapest way of having one edge enter each node $v \neq r$. Otherwise, we consider the problem with the modified costs $c'_e$, which is equivalent by the relationship we discovered between the original costs $c_e$ and the modified costs $c'_e$. After contracting the 0-cost cycle $C$ to obtain the smaller graph $G'$, the algorithm produces a minimum cost arborescence for $G'$ by the inductive hypothesis. Then by the previous result, there is an minimum cost arborescence in $G$ that corresponds to the minimum cost arborescence computed for $G'$. □

## 4. Matroids

A **matroid** $M$ is a pair $(S, \mathcal{I})$ where $S$ is a ground set and $\mathcal{I} \subseteq 2^S$ is a family of sets over the ground set 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 an element $e \in A \setminus B$ such that $B \cup \{e\} \in \mathcal{I}$.

We call the elements of $\mathcal{I}$ the independent sets.

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

**Proof.** We verify that the three properties hold.

(1) We see that $(V, \varnothing)$ is acyclic, so $\varnothing \in \mathcal{I}$.
(2) Let $A \in \mathcal{I}$ and $B \subseteq A$. Then $(V, A)$ is acyclic, so the subgraph $(V, B)$ is acyclic and hence $B \in \mathcal{I}$.
(3) Let $A, B \in \mathcal{I}$ with $|A| > |B|$. Then $(V, A)$ has $|V| - |A|$ connected components and $(V, B)$ has $|V| - |B|$ connected components with $|V| - |B| > |V| - |A|$. Therefore, there is a connected component $C$ of $(V, A)$ which 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 there must be an edge crossing the boundary of the connected component to get to $C_2'$. We have that $B \cup \{e\} \in \mathcal{I}$ since including $e$ does not introduce a cycle. □

A **basis** for a matroid $M = (S, \mathcal{I})$ is a maximal independent set $A \in \mathcal{I}$. By maximal, we mean that $A \cup \{e\}$ is not independent for any $e \in S \setminus A$.

**Size of a basis.** All bases of a matroid have the same size.

**Proof.** Let $A, B \in \mathcal{I}$ be bases for the matroid. Towards a contradiction, suppose that $|A| \neq |B|$. Without loss of generality, we may assume that $|A| > |B|$ since we can switch the roles of $A$ and $B$ otherwise. By the exchange property, there exists $e \in A \setminus B$ such that $B \cup \{e\} \in \mathcal{I}$, contradicting the fact that $B$ is a maximal independent set. □

In the maximum weight independent set problem, we are given a matroid $M = (S, \mathcal{I})$ and a weight $w_e$ for all $e \in S$. The goal is to find an independent set of maximum total weight.

We may assume that $w_e > 0$ for all $e \in S$. Indeed, if $A \in \mathcal{I}$ were independent, then $A \setminus \{e \in A : w_e \leq 0\}$ is also independent by the downward closed property with $w(A \setminus \{e \in A : w_e \leq 0\}) \geq w(A)$ since we are throwing away elements that are decreasing the weight.

**Greedy algorithm for the maximum weight independent set problem.** This algorithm is similar to Kruskal's algorithm in that it sorts the element by weight and chooses an element if the set remains independent.

1. Sort the elements in $S$ such that $w_{e_1} \geq \cdots \geq w_{e_{|S|}}$. Set $A = \varnothing$ and $i = 1$.
2. While $i \neq |S| + 1$, do the following:
   (a) If $A \cup \{e_i\} \in \mathcal{I}$, then set $A = A \cup \{e_i\}$.
   (b) Increment $i = i + 1$.

Note that the assumption that $w_e > 0$ is crucial here, as otherwise the algorithm might choose elements that decrease the weight of the independent set. To prove correctness, we first need to introduce a few useful lemmas.

**Lemma 4.1.** (Matroids exhibit the greedy choice property.) Suppose that $M = (S, \mathcal{I})$ is a matroid with weight function $w$ and $S$ is sorted into monotonically decreasing order by weight. Let $x$ be the first element of $S$ such that $\{x\}$ is independent (if any such $x$ exists). If $x$ exists, then there is an maximum weight independent set $A$ that contains $x$.

**Proof.** If no such $x$ exists, then the only independent set is the empty set and the lemma holds vacuously. Otherwise, let $B$ be any maximum weight independent set. We assume that $x \notin B$ for otherwise setting $A = B$ gives us a maximum weight independent set containing $x$.

No element of $B$ has weight greater than $w_x$. To see this, observe that $y \in B$ implies that $\{y\} \in \mathcal{I}$ by the downward closed property. Our choice of $x$ ensures that $w_x \geq w_y$ for any $y \in B$.

Construct the set $A$ as follows. Start with $A = \{x\}$. Note that $A$ is independent by assumption. By the exchange property, we can repeatedly find a new element of $B$ that we can add to $A$ until $|A| = |B|$, while preserving the independence of $A$. In particular, $A$ and $B$ are the same except that $A$ has $x$ and $B$ has some other element $y$. Then $A = (B \cup \{x\}) \setminus \{y\}$ for some $y \in B$, implying that

$$w(A) - w(B) = w_x - w_y \geq 0.$$

Since $B$ is a maximum weight independent set, it follows that $A$ is also a maximum weight independent set containing $x$. □

The next lemma says that if an element is not an option initially, then it cannot be an option later.

**Lemma 4.2.** Let $M = (S, \mathcal{I})$ be a matroid. Let $A \in \mathcal{I}$, and suppose that $x$ is an element of $S$ such that $A \cup \{x\} \in \mathcal{I}$. Then $\{x\} \in \mathcal{I}$.

**Proof.** This follows immediately from the downward closed property. □

The following is the contrapositive of Lemma 4.2.

**Lemma 4.3.** Let $M = (S, \mathcal{I})$ be a matroid. If $\{x\} \notin \mathcal{I}$, then $A \cup \{x\} \notin \mathcal{I}$ for any $A \in \mathcal{I}$.

In particular, the greedy algorithm cannot make an error by passing over initial elements $x$ in $S$ such that $\{x\} \notin \mathcal{I}$, since they can never be used.

**Lemma 4.4.** (Matroids exhibit the optimal substructure property.) Let $x$ be the first element chosen by the greedy algorithm for the matroid $M = (S, \mathcal{I})$ with weight function $w$. Then the problem of finding a maximum weight independent set containing $x$ reduces to finding a maximum weight independent set of the weighted matroid $M' = (S', \mathcal{I}')$, where

$$S' = \{y \in S : \{x, y\} \in \mathcal{I}\},$$
$$\mathcal{I}' = \{B \subseteq S \setminus \{x\} : B \cup \{x\} \in \mathcal{I}\}.$$

The weight function for $M'$ is the weight function for $M$ restricted to $S'$. (We call $M'$ the **contraction** of $M$ by the element $x$.)

**Proof.** If $A$ is a maximum weight independent set containing $x$, then $A' = A \setminus \{x\}$ is independent for $M'$. Conversely, any independent set $A'$ of $M'$ yields an independent set $A = A' \cup \{x\}$ for $M$. Since we have in both cases that $w(A) = w(A') + w_x$, a maximum weight independent set in $M$ containing $x$ yields a maximum weight independent set in $M'$, and vice versa. □

Finally, we prove the correctness of the greedy algorithm.

**Proof of correctness of the greedy algorithm.** By Lemma 4.3, any elements $x$ in $S$ that the greedy algorithm passes over initially because $\{x\} \notin \mathcal{I}$ can be safely ignored. Once the greedy algorithm selects the first element $x$, Lemma 4.1 implies that the algorithm does not make an error by adding $x$ to $A$ as there is a maximum weight independent set containing $x$.

Finally, Lemma 4.4 implies that the problem reduces to finding an maximum weight independent set in the matroid $M'$ that is the contraction of $M$ by $x$. After the greedy algorithm sets $A$ to $\{x\}$, we can interpret the remaining steps as acting in the matroid $M' = (S', \mathcal{I}')$ because $B$ is independent in $M'$ if and only if $B \cup \{x\}$ in $M$ for all $B \in \mathcal{I}'$. Then the subsequent application of the greedy algorithm finds a maximum weight independent set for $M'$, and so overall, the greedy algorithm finds a maximum weight independent set for $M$. □

## 5. Minimum Cost Steiner Trees

In the minimum cost Steiner tree problem, we are given a graph $G = (V, E)$, edge costs $c_e \geq 0$ for all $e \in E$, and a set of terminals $T \subseteq V$. The goal is to find a tree of minimum cost that connects all the terminals in $T$. Unfortunately, this problem is **NP**-hard.

We also introduce a variant of this problem, called the minimum *metric* Steiner tree problem. In this case, are impose the additional conditions that $G = (V, E)$ is complete and $c_{uw} \leq c_{uv} + c_{vw}$ for all distinct $u, v, w \in V$.

We will call these problems MinSteinerTree and MinMetricSteinerTree respectively.

**Theorem 5.1.** There is an approximation factor preserving reduction from the MinSteinerTree problem to the MinMetricSteinerTree problem.

**Proof.** We will transform an instance $I$ of MinSteinerTree consisting the graph $G = (V, E)$ to an instance $I'$ of MinMetricSteinerTree as follows:

- Let $G' = (V, E')$ be the complete graph on the vertex set $V$.
- Define the cost $c'_{uv}$ of an edge $uv \in E'$ to be the cost of a shortest $u, v$-path in $G$.

We call $G'$ the **metric closure** of $G$. We will keep the terminal set $T$ the same.

We quickly verify that $I'$ is an instance of MinMetricSteinerTree. It is enough to show that the costs satisfy the triangle inequality. Given $u, v, w \in V$, let $P_1$ be a shortest $u, v$-path, let $P_2$ be a shortest $v, w$-path, and let $P_3$ be a shortest $u, w$-path. Then $P_1$ together with $P_2$ is a possible $u, w$-path, so we have

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

Note that for any edge $uv \in E$, its cost in $G'$ is no more than its cost in $G$. Therefore, the cost of an optimal solution in $I'$ does not exceed the cost of an optimal solution in $I$.

Next, given a Steiner tree $T'$ in $I'$, we show how to obtain a Steiner tree $T$ in $I$ of at most the same cost. The cost of an edge $uv \in E'$ corresponds to the cost of a shortest $u, v$-path in $G$. Replace each edge in $T'$ by the corresponding path to obtain a subgraph of $G$. Note that all the terminals are connected since they were connected in $T'$. However, this graph might contain cycles in general. If so, remove edges to obtain a tree $T$, which has cost at most that of $T'$. This completes the proof of the approximation factor preserving reduction. □

Therefore, any approximation factor established for the metric Steiner tree problem carries over to the entire Steiner tree problem.

**MST-based approximation algorithm for the metric Steiner tree problem.** It is clear that a minimum spanning tree on the terminals $T$ is a feasible solution for the problem. However, since we can find MSTs in polynomial time and the metric Steiner tree problem is **NP**-hard, we cannot hope for this MST on $T$ to give an optimal Steiner tree. Nonetheless, an MST on $T$ is not much more costly than an optimal Steiner tree.

**Theorem 5.2.** The cost of an MST on $T$ is at most $2 \cdot \text{OPT}$.

**Proof.** Consider a Steiner tree of cost OPT. By doubling the edges, we obtain an Eulerian graph connecting all the vertices in $T$ and potentially some other Steiner vertices. Find an Eulerian tour of this graph. The cost of this Eulerian tour is $2 \cdot \text{OPT}$.

Next, obtain a Hamiltonian cycle on the vertices of $T$ by traversing the Eulerian tour and "shortcutting" Steiner vertices and previously visited vertices of $T$. Note that by the triangle inequality, the shortcuts do not increase the cost of the tour. By deleting one edge of this Hamiltonian cycle, we obtain a path that connects all of $T$ and has cost at most $2 \cdot \text{OPT}$. This path is itself spanning tree on $T$. Then an MST on $T$ is at most the cost of this path, and thus has cost at most $2 \cdot \text{OPT}$. □

## 6. Computational Complexity

The complexity class **P** consists of all problems that can be solved efficiently (in polynomial time). Some examples of problems in **P** include shortest paths, minimum spanning trees, and minimum cost arborescences, which can be solved with Dijkstra's algorithm, Prim's or Kruskal's algorithm, and Edmonds' algorithm respectively. Since we'll be working with linear programs a lot later, we'll add that solving a linear program is in **P**.

For a minimization problem $X$, the **decision problem** corresponding to $X$ takes as input an instance of $X$ and a number $k$, and asks if there is a feasible solution for the given problem with cost at most $k$.

**Problem.** (DecisionSteinerTree) Given a graph $G = (V, E)$, edge costs $c_e \geq 0$ for all $e \in E$, terminals $T \subseteq V$, and a number $k$, determine if there exists a Steiner tree of cost at most $k$.

Given two problems $X$ and $Y$, we say that $X$ **reduces** in polynomial time to $Y$, denoted $X \leq_P Y$, if one can solve $X$ by using a polynomial number of elementary operations and a polynomial number of calls to an oracle that solves $Y$. Intuitively, this means that $X$ is not harder than $Y$, because given a way to solve $Y$, we also know how to solve $X$. We make some observations:

- Let $X$ and $Y$ be problems such that $X \leq_P Y$. If $Y \in \mathbf{P}$, then $X \in \mathbf{P}$. Equivalently, we know that if $X \notin \mathbf{P}$, then $Y \notin \mathbf{P}$.
- If $X$, $Y$, and $Z$ are problems such that $X \leq_P Y$ and $Y \leq_P Z$, then we have $X \leq_P Z$ because the composition of polynomials is a polynomial.

For a decision problem $X$, an **efficient certifier** $B$ for the problem $X$ is an algorithm with polynomial running time that 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 $B(s, t)$ outputs "yes".
- If $s$ is a "no" instance of $X$, then $B(s, t)$ outputs "no" for all certificates $t$.

For example, in the DecisionSteinerTree problem, a certificate $t$ can be an edge set $F \subseteq E$. The efficient certifier will output "yes" if $F$ is a Steiner tree such that $c(F) \leq k$, and "no" otherwise. It is easy to check that $c(F) \leq k$, and verifying whether $F$ is a Steiner tree can be done by checking if the edges form a tree and the terminals are all connected, both of which can be done in polynomial time.

The complexity class **NP** consists of all decision problems that admit an efficient certifier. Clearly, if a decision problem is in **P**, then it is in **NP** by simply using the polynomial time algorithm as the certifier with an empty certificate. Conversely, it is not known if every problem in **NP** also lies in **P**; this is the famous **P** = **NP** problem.

A problem $X$ is **NP-hard** if for every $Y \in \mathbf{NP}$, we have $Y \leq_P X$. We say that $X$ is **NP-complete** if $X \in \mathbf{NP}$ and $X$ is **NP**-hard.

**Problem.** (3-SAT) 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 $t_1, t_2, t_3 \in \{x_1, \ldots, x_n\} \cup \{\bar{x}_1, \ldots, \bar{x}_n\}$, determine if there is a truth assignment to the boolean variables $x_1, \ldots, x_n$ such that $C_1 \wedge \cdots \wedge C_k$ is satisfied.

One of the earliest results in complexity theory is the following:

**Theorem 6.1.** (Cook-Levin) 3-SAT is **NP**-complete.

We give a recipe to prove that a problem $X$ is **NP**-complete.

(i) Show that $X \in \mathbf{NP}$.

(ii) Pick an **NP**-complete problem $Y$ and show that $Y \leq_P X$. Then $Z \leq_P Y$ and therefore $Z \leq_P X$ for all $Z \in \mathbf{NP}$ by transitivity, so $X$ is **NP**-hard.

**Problem.** (DecisionVertexCover) Given a graph $G = (V, E)$ and an integer $\ell$, determine 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 $U$.

**Proposition 6.2.** DecisionVertexCover is **NP**-complete.

**Proof.** Note that DecisionVertexCover $\in$ **NP** because by taking a subset $U \subseteq V$ as the certificate, it is easy to verify that $|U| \leq \ell$ and we can iterate through the edge set to determine if one of the endpoints is in $U$.

Next, we will reduce a known **NP**-complete problem to DecisionVertexCover. In this case, we will show that 3-SAT $\leq_P$ DecisionVertexCover.

Suppose we are given an instance of 3-SAT consisting of the variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$.

**Remark.** We can picture 3-SAT as follows: we must choose one term from each clause, and find a truth assignment that causes all these terms to evaluate to 1, therefore satisfying all clauses. We succeed if we can select a term from each clause in a way that no two terms conflict; we'll say that two terms **conflict** if one is equal to the variable $x_i$ and the other is its negation $\bar{x}_i$. If we can avoid conflicting terms, we can find a truth assignment that makes the selected terms from each clause evaluate to 1.

We construct a DecisionVertexCover instance using this perspective. Let $G = (V, E)$ be a graph consisting of $3k$ nodes grouped into $k$ triangles. That is, for $i = 1, \ldots, k$, we introduce three vertices $v_{i1}, v_{i2}, v_{i3}$ joined to each other by edges. We will label $v_{ij}$ with the $j$-th term from clause $C_i$ of the 3-SAT instance.

**Remark.** What does a vertex cover look like for this graph? We will need to pick at least two out of three vertices from each triangle in order to cover all of them. We want to prevent simply being able to pick everything, so we impose the condition that we can only pick two vertices from each triangle. This corresponds to taking $\ell = 2k$. However, this still does not prevent us from picking two conflicting terms.

We will encode conflicts by introducing more edges to the graph. For each pair of vertices whose labels correspond to conflicting terms, we add an edge between them.

**Claim.** The original 3-SAT instance is satisfiable if and only if the constructed graph $G$ has a vertex cover of size $\ell = 2k$.

**Proof of Claim.** ($\Rightarrow$) Suppose that the original 3-SAT instance is satisfiable. Then every triangle in our graph contains at least one node whose label evaluates to 1, say $w_i$ for all $i = 1, \ldots, k$. Let $U = V \setminus \{w_i : i = 1, \ldots, k\}$ and observe that $|U| = 3k - k = 2k$. We show that $U$ is a vertex cover for the constructed graph $G$. Suppose otherwise, so there is some $e \in E$ with both endpoints not in $U$. By construction, these endpoints are $w_i$ and $w_j$ for some $i \neq j$. Then these are in separate triangles, so this must be one of the edges between two conflicting terms. But the labels for $w_i$ and $w_j$ both evaluate to 1, meaning that we had an invalid truth assignment. This gives us a contradiction.

($\Leftarrow$) Suppose that $G$ has a vertex cover of size $\ell = 2k$. Note that it must consist of two nodes from each triangle in order to be a vertex cover. For all $i = 1, \ldots, k$, let $w_i$ be the vertex not chosen in $U$ from the triangle vertices $v_{i1}, v_{i2}, v_{i3}$. Define a truth assignment $v$ to $x_1, \ldots, x_n$ as follows:

- For each variable $x_i$, if neither $x_i$ nor $\bar{x}_i$ appears as a label in $w_1, \ldots, w_k$, then arbitrarily set $v(x_i) = 1$.
- The case where both $x_i$ and $\bar{x}_i$ appear as a label in $w_1, \ldots, w_k$ is impossible; there is an edge between these two points, contradicting the fact that $U$ is a vertex cover.
- For the remaining case where exactly one of $x_i$ or $\bar{x}_i$ appears as a label in $w_1, \ldots, w_k$, we set $v(x_i) = 1$ if $x_i$ appears and $v(x_i) = 0$ if $\bar{x}_i$ appears.

From this construction, all the clauses $C_i$ will evaluate to 1, so $v$ satisfies the original 3-SAT instance. $\square$

Previously, we claimed that the Steiner tree problem was **NP**-hard. We now prove this.

**Proposition 6.3.** DecisionSteinerTree is **NP**-complete.

**Proof.** We already showed earlier that DecisionSteinerTree is in **NP**. We will show that it is **NP**-hard via a reduction from DecisionVertexCover.

Suppose that we have an instance of DecisionVertexCover consisting of a graph $G = (V, E)$ and an integer $\ell$. We construct a DecisionSteinerTree instance as follows:

- Let $G' = (V', E')$, where $V' = \{r\} \cup V \cup \{t_e : e \in E\}$ and
$$E' = \{rv : v \in V\} \cup \{vt_e : e \in E \text{ and } v \text{ is an endpoint of } e\}.$$
We can think of $G'$ as 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 $v \in V$ to $t_e$ if $v$ is an endpoint of $e$.
- The terminals will be $T = \{r\} \cup \{t_e : e \in E\}$; these are the vertices in the top and bottom layers.
- We assign the edge costs to be
$$c_{e'} = \begin{cases} 100|V|, & \text{if } e' \in \{vt_e : e \in E \text{ and } v \text{ is an endpoint of } e\}, \\ 1, & \text{otherwise.} \end{cases}$$
- Finally, we set $k = 100|V||E| + \ell$.

**Claim.** The original DecisionVertexCover instance has a vertex cover of size at most $\ell$ if and only if the constructed DecisionSteinerTree instance has a Steiner tree of cost at most $k$.

**Proof of Claim.** ($\Rightarrow$) Suppose that $G = (V, E)$ has a vertex cover $U$ of size $|U| \leq \ell$. For all $e \in E$, we can find a vertex $u_e \in U$ such that $e \in \delta(u_e)$. Consider the set of edges
$$F := \{ru : u \in U\} \cup \{t_e u_e : e \in E\} \subseteq E'.$$
This has cost $c(F) = |U| + 100|V||E| \leq \ell + 100|V||E| = k$. Moreover, $F$ is a Steiner tree since it connects all the terminals $T = \{r\} \cup \{t_e : e \in E\}$.

($\Leftarrow$) Suppose that $F$ is a Steiner tree for $G' = (V', E')$ of cost $c(F) \leq k$ with respect to the terminals $T$. Moreover, suppose towards a contradiction that there is no vertex cover for $G = (V, E)$ of size at most $\ell$.

Let $\bar{E} = \{vt_e : e \in E \text{ and } v \text{ is an endpoint of } e\}$. Observe that
$$c(F \cap \bar{E}) \geq 100|V||E|$$
since $100|V|$ is the cost of each edge incident to $t_e$ and the number of terminals on the bottom layer of the graph is $|E|$. Together with the bound $c(F) \leq \ell + 100|V||E|$, this implies that $c(F \setminus \bar{E}) \leq \ell$.

We may assume that $\ell \leq |V|$, because otherwise the DecisionVertexCover instance has a vertex cover by simply choosing all the vertices. Then 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 $t_e$ gives
$$c(F) \geq 100|V||E| + 100|V| > 100|V||E| + \ell = k,$$
which is a contradiction. Construct the set
$$U = \{v \in V : v \text{ is an endpoint of an edge in } F \setminus \bar{E}\}.$$
Note that $|U| \leq c(F \setminus \bar{E}) \leq \ell$ since each edge from $F \setminus \bar{E}$ has cost 1. Suppose by way of contradiction that $U$ is not a vertex cover for $G$. Then there is some edge $e \in E$ such that both endpoints are not in $U$. The corresponding terminal $t_e$ is only adjacent to $u_e$ in $F$. But $u_e \notin U$, so it is not connected to $r$ and hence $t_e$ is not connected to $r$ either. This contradicts the fact that $F$ is a Steiner tree with respect to $T = \{r\} \cup \{t_e : e \in E\}$. Thus, $U$ is a vertex cover for $G$. □

## 7. Network Design Problem

In the network design problem, we are given a graph $G = (V, E)$, edge costs $c_e \geq 0$ for all $e \in E$, and a cut-requirement function $f : 2^V \to \{0, 1\}$. The goal is to find a subset $F \subseteq E$ which minimizes

$$c(F) = \sum_{e \in F} c_e$$

such that $|F \cap \delta(S)| \geq 1$ for all $S \subseteq V$ satisfying $f(S) = 1$. A function $f : 2^V \to \{0, 1\}$ is **proper** if:

(i) $f(V) = 0$;
(ii) $f(V \setminus S) = f(S)$ for all $S \subseteq V$;
(iii) $f(A \cup B) \leq \max\{f(A), f(B)\}$ for all nonempty subsets $A, B \subseteq V$ such that $A \cap B = \varnothing$.

We assume that the cut-requirement function is proper. The network design problem is **NP**-hard in general as the Steiner tree problem is a special case of it.

For ease of notation, we set $\mathcal{K} = \{S \subseteq V : f(S) = 1\}$. The primal LP for the problem is

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ \text{subject to} \quad & \sum_{e \in \delta(S)} x_e \geq 1 \quad \text{for all } S \in \mathcal{K} \\ & x_e \geq 0 \quad \text{for all } e \in E. \end{aligned}$$

The dual LP is then

$$\begin{aligned} \max \quad & \sum_{S \in \mathcal{K}} y_S \\ \text{subject to} \quad & \sum_{S \in \mathcal{K} \,:\, e \in \delta(S)} y_S \leq c_e \quad \text{for all } e \in E \\ & y_S \geq 0 \quad \text{for all } S \in \mathcal{K}. \end{aligned}$$

Given $F \subseteq E$, the collection of violated sets with respect to $F$ is $\mathcal{B} := \{S \in \mathcal{K} : F \cap \delta(S) = \varnothing\}$. Note that we ultimately need $|F \cap \delta(S)| \geq 1$ for all $S \in \mathcal{K}$, so these are the cuts that we still need to include. We will consider the collection $\mathcal{D}$ of (inclusion-wise) minimal sets in $\mathcal{B}$. In the primal-dual algorithm, the sets in $\mathcal{D}$ are the ones that we will uniformly increase the $y_S$ values for.

**Lemma 7.1.** For every $F \subseteq E$, we have that

$$\mathcal{D} = \{S \subseteq V : S \text{ is a connected component of } (V, F) \text{ such that } f(S) = 1\}.$$