1. **In the lecture notes, we argued that the "minimum cost" occurs when the subgraph is a spanning tree. What happens if zero or negative costs are allowed? What goes wrong with our argument that the optimal answer must be a spanning tree?**

Own attempt:

If zero costs are allowed, then it's possible for the optimal answer to have a cycle. For example, consider a graph with a cycle of three nodes, all with weight 0. It would have the same minimum cost as a minimum spanning tree. This violates our assumption that the optimal answer must be a spanning tree.

If negative costs are allowed, then, for a graph with negative cost edges, the optimal answer would not be a minimum spanning tree, but a graph that includes as many negative cost edges as possible. Such optimal answers could include cycles, therefore also violating our assumption that the optimal answer must be a spanning tree.

AI-assisted:

If zero or negative costs are allowed, the minimum-cost connected spanning subgraph is not necessarily a spanning tree.

- For zero costs: Consider a triangle with all edge weights 0. The whole graph, a cycle, has the same total cost as any spanning tree: 0.
- For negative costs: Consider a triangle with each edge weight as -1. The whole graph, a cycle, has cost -3, while any spanning tree has only two edges, cost -2.

In both cases, the optimal solution can contain a cycle, violating the claim that the optimal answer must be a spanning tree.

2. **Prove that Prim's algorithm is correct, by proving the following statement by induction on k: "After k steps, Prim's algorithm produces a tree connecting k+1 nodes, and this tree is a subset of some MST."**

AI-assisted:

I claim that: "After k steps, Prim's algorithm produces a tree connecting k + 1 nodes, and this tree is a subset of some MST.

We prove by induction on k.

Let V_k be the set of nodes already in the tree and E_k be the set of edges chosen by Prim's algorithm after k steps.

By the property of trees, we have $|V_k| = k + 1$ and $|E_k| = k$.

Initialization. Base case (k = 0): Prim's algorithm starts with an arbitrary node x and no edges. Thus $V_0 = \{x\}$,
$E_0 = \varnothing$.

This single-node graph is trivially a tree, and because the graph is connected, there exists an MST. Any MST contains node x, by the spanning property, so the empty set of edges E_0 is certainly a subset of that MST. Hence, the statement holds for k = 0.

Maintenance. By the inductive hypothesis, we assume the statement holds for some k with $0 <= k < n - 1$. Thus, there exists an MST **T** such that E_k is a subset of **T** and (V_k, E_k) is a tree with k + 1 nodes.

In step k + 1, Prim's algorithm chooses a minimum-cost edge a = (u, v) where u is a part of the set V_k while v is not. Such an edge must exist, otherwise the graph would be disconnected, contradicting its connectivity.

We first show that Prim's algorithm creates a tree with k + 1 nodes.

Add a to E_k and v to V_k:

$$E_{k+1} = E_k \cup \{a\}, V_{k+1} = V_k \cup \{v\}.$$

Since (V_k, E_k) is a tree and a is the only edge incident to v in E_{k+1}, the new subgraph (V_{k+1}, E_{k+1}) remains acyclic and connected, hence a tree with $(k + 1) + 1 = k + 2$ nodes. So the tree property is maintained.

Next, we show that this tree is a subset of some MST, containing E_{k+1}.

[Exchange Argument] Because T is a spanning tree, there is a unique path in T from u to v. Let a' be the first edge along this path that leaves V_k (with one endpoint in V_k, the other outside V_k). Such an edge exists because u is a part of V_k and v is not.

Consider the tree $T \setminus \{a'\}$. Removing a' splits T into two subtrees: one containing u (and all of V_k) and the other containing v. Now form

$$T' = (T \setminus \{a'\}) \cup \{a\}.$$

Since **a** connected a node in the first subtree to a node in the second, T' is again a spanning tree. By Prim's choice, **a** is a minimum-cost edge among all edges that connect a vertex in V_k to a vertex outside V_k.

Edge a' also has exactly one endpoint in V_k and one outside V_k. So,

$$\text{cost}(\mathbf{a}) <= \text{cost}(\mathbf{a'}).$$

Therefore,

$$w(\mathbf{T'}) = w(\mathbf{T}) - w(\mathbf{a'}) + w(\mathbf{a}) <= w(\mathbf{T}).$$

But **T** is an MST, so $w(\mathbf{T'}) = w(\mathbf{T})$ and **T'** is also a minimum spanning tree.

By the inductive hypothesis, $E_k \subseteq T$.

Moreover, edge **a'** cannot belong to E_k, since every edge of E_k has both endpoints in V_k, while **a'** has one endpoint outside V_k.

Therefore, removing **a'** does not remove any edge of E_k, and adding **a** gives:

$$E_{k+1} = E_k \cup \{a\} \subseteq T'.$$

Thus, after k + 1 steps, the tree built by Prim's algorithm is contained in some MST, completing the induction. The induction proves the statement for all k up to n - 1.

Termination. After n - 1 steps, we have a spanning tree (V_{n-1}, E_{n-1}) that is contained in some MST. Since it already spans all n nodes, it must be that MST itself. Therefore, Prim's algorithm correctly computes an MST.

3. The **reverse-delete algorithm** is an MST algorithm appearing in the same paper where Kruskal introduced the algorithm now named after him. It goes as follows.

```
sort the edges in decreasing cost
for each edge in this order
    if removing the edge doesn't disconnect the graph
        remove it
```

Prove that the reverse-delete algorithm is correct.

3. **Prove that the reverse-delete algorithm is correct.** (Proof methods to use: Induction, exchange argument).

Let the edges be sorted in **decreasing** order of cost: $a_1, a_2, \ldots, a_e$. We define after k steps:

- S_k = edges kept so far,
- E_k = edges removed so far.

We have $S_k \cup E_k = \{a_1, \ldots, a_k\}$ because each step decides on the next edge.

I claim that: “For every k with 0 <= k <= e, there exists an MST containing all edges in S_k and no edges in E_k.”

Proof by induction on k.

Base Case (k = 0): $S_0 = E_0 = \varnothing$. Any MST trivially contains all edges of S_0 and no edges of E_0.

Inductive Step: We assume that the lemma holds for some k ($0 <= k < e$). So there is an MST **T** with $S_k \subseteq \mathbf{T}$ and $E_k \cap \mathbf{T} = \varnothing$.

Let $\mathbf{C} = S_k \cup \{a_{k+2}, \ldots, a_e\}$. This is the set of edges still present **after** removing a_{k+1} from the original sorted list (but before deciding whether to keep a_{k+1}).

Note $\mathbf{T} \subseteq \mathbf{C} \cup \{a_{k+1}\}$ because **T** contains no edges from E_k and all edges of S_k, and any other edge of **T** must come from $\{a_{k+2}, \ldots, a_e\}$.

We consider step k + 1:

Case 1: C is disconnected.
Then the algorithm **keeps** a_{k+1} since removing it would disconnect the graph. Thus,

$$S_{k+1} = S_k \cup \{a_{k+1}\}, E_{k+1} = E_k.$$

Because **T** is connected but **C** is not, and $T \subseteq \mathbf{C} \cup \{a_{k+1}\}$, **T** must contain a_{k+1}. Hence $S_{k+1} \subseteq \mathbf{T}$ and $E_k \cap \mathbf{T} = \varnothing$ since $E_{k+1} = E_k$.
So **T** itself satisfies the lemma for k + 1.

Case 2: C is connected.
Then the algorithm **removes** a_{k+1}. Thus,

$$S_{k+1} = S_k, E_{k+1} = E_k \cup \{a_{k+1}\}.$$

If a_{k+1} is not part of **T**, then **T** already satisfies the requirements for k + 1.
If a_{k+1} is part of **T**, then we must find another MST that excludes a_{k+1} but still contains S_k.

Let x, y be the endpoints of a_{k+1}. Remove a_{k+1} from T: $T' = T \setminus \{a_{k+1}\}$ is a forest with two trees, one containing x, and the other containing y.

Color nodes in the tree containing x **black**, nodes in the other tree **white**. Since **C** is connected, there is a path in **C** from x to y.

That path must contain an edge **a** whose endpoints have different colors because x is black, and y is white.

Such an edge **a** connects the two trees of **T'** and is **not** in **T',** as all edges of **T'** connect same-colored nodes.

Also **a** $\notin S_k$ since $S_k \subseteq$ **T'** and **a** would connect different components of **T'**, so **a** $\in \{a_{k+2}, \ldots, a_e\}$.

Because edges are sorted in **decreasing** order,

$$\text{cost}(a) <= \text{cost}(a_{k+1}).$$

Now form

$$T'' = T' \cup \{a\} = (T \setminus \{a_{k+1}\}) \cup \{a\}.$$

Adding **a** reconnects the two trees, so T'' is a spanning tree.
We have

$$\text{cost}(T'') = \text{cost}(T) - \text{cost}(a_{k+1}) + \text{cost}(a) <= \text{cost}(T).$$

Since **T** is an MST, cost(T'') >= cost(T). Hence equality holds, so T'' is also an MST.

We now check if the lemma holds:

- $S_{k+1} = S_k \subseteq T''$ (we only removed a_{k+1} which was not in S_k and added a, which is not in S_k)
- $E_{k+1} = E_k \cup \{a_{k+1}\}$: T'' contains no edge from E_k (since T didn't and we only added a $\notin E_k$) and T'' does not contain a_{k+1}.

So, T'' satisfies the lemma for k + 1.

Thus, in all cases, the lemma holds for k + 1, completing the induction.

We now apply the lemma to k = e. When the algorithm finishes, S_e is the set of kept edges, E_e is the set of removed edges, and $S_e \cup E_e = \{a_1, \ldots, a_e\}$ (all edges). By the lemma, there exists an MST T* such that $S_e \subseteq T^*$ and $E_e \cap T^* = \varnothing$.

But since $S_e \cup E_e$ is **all** edges, this means $T^* = S_e$. Therefore, S_e itself is an MST, so the reverse-delete algorithm correctly returns an MST.

4. **Provide an $O(e^2)$ implementation of the reverse-delete algorithm.**

Pseudocode:

```
def MST(V, E)
    for each a in E in decreasing order of cost (via merge sort)
        E' := E \ {a} # try removing the edge a from the list of edges
        Run BFS starting from node 0, using only edges in E'
        if all nodes have been visited
            E := E' # replace the adjacency list with one not containing a
    return E
```

The sorting step runs in O(e log e). "BFS" is used to check the graph connectivity, done using the standard routine; one could construct the adjacency list and then perform the traversal using a queue. This runs in O(n + e). Since we assume the graph is connected, e >= n - 1, hence n = O(e) and so the above is just O(e + e) = O(e). Constructing E' also takes O(e) time. Therefore, because the loop's body is run e times, this takes e * O(e) = $O(e^2)$. The overall running time is

$O(e \log e) + O(e^2) = O(e^2)$, dominated by the main loop.

5. The **minimum spanning forest** (MSF) is the union of the MSTs of all connected components.Which among the following four algorithms compute the MSF out of the box, without modification, and why? Also, for those that don't compute the MSF out of the box, explain how to modify them so that they do.

- **Prim's algorithm (No)**
    - Prim starts from a single root and grows one tree. It only builds an MST for the connected component containing that start node, ignoring the other components.
    - **Modification:** Repeat Prim's algorithm from an unvisited node until all nodes are visited, building an MST for each connected component separately.
- **Kruskal's algorithm (Yes)**
    - Kruskal builds a maximal acyclic subgraph (a forest) by adding edges in increasing weight order, skipping edges that create cycles.
    - It doesn't require the graph to be connected.
    - If the graph is disconnected, it simply yields a spanning forest where each connected component is a tree (an MSF by definition).
- **Boruvka's algorithm (Yes, only need to tweak stopping condition)**
    - Boruvka works in phases, each component picks its cheapest outgoing edge.
    - If the graph is disconnected, each connected component will independently converge to its own MST.
    - The algorithm naturally handles multiple components as long as the stopping condition is "no new edges are added in a phase" rather than "only one component remains."
- **Reverse-delete algorithm (No)**

- Reverse-delete removes edges in decreasing weight order only if removal **does not disconnect the graph**.
- If the original graph is disconnected, removing any edge will never disconnect it further as it's already disconnected, so the algorithm will remove all edges and produce an empty forest, which is wrong.
- **Modification:** Change the condition from "removing the edge does not disconnect the graph" to "removing the edge does not increase the number of connected components."

6. **Let *G* be a weighted undirected graph. Prove or disprove: Every MSF of *G* can be the output of Kruskal's algorithm.**

Let **F** be an arbitrary MSF of **G**. We must show that there exists a run of Kruskal's algorithm that outputs **F**.

Step 1: Handle ties
Kruskal's algorithm sorts edges by weight, but when weights are equal, the order is not specified. We can choose a tie-breaking rule that prioritizes edges in F over edges not in F within each equal-weight group.

Step 2: Inductive Invariant

Let $w_1 < w_2 < \ldots < w_k$ be the distinct weights in increasing order. Define E_i = {edges with weight w_j, $j <= i$}.

Claim: After Kruskal processes all edges in E_i, it has chosen exactly $\mathbf{F} \cap \mathbf{E_i}$.

We prove by induction on i.

Base i = 0: $E_0 = \varnothing$, $\mathbf{F} \cap \mathbf{E_0} = \varnothing$ holds.
Inductive Step: Assume after processing E_{i-1}, chosen edges = $\mathbf{F} \cap \mathbf{E_{i-1}}$. Now, process edges of weight w_i.

1. Edges of **F** with weight w_i are processed first by our specific tie-breaking rule.
   - Since **F** is acyclic, adding these edges won't create a cycle in the current forest. Thus, Kruskal includes them.
2. Edges not in **F** with weight w_i:
   - Take such an edge $\mathbf{a} \notin \mathbf{F}$.
   - Adding a to F creates a cycle **C**, since **F** is maximal acyclic.

- Since **a** $\notin$ **F**, **C** contains at least one edge **b** with cost(b) > cost(a) = w_i. Otherwise, all edges in C have weight <= w_i, and swapping a for b would give a cheaper or equal-cost forest, contradicting minimality of **F.** Thus, **b** $\notin$ $\mathbf{E_i}$**.**
- When Kruskal considers a, the forest so far contains **F** $\cap$ $\mathbf{E_i}$**,** which includes all edges of **C** except possibly **b,** since **b** has weight > w_i and isn't in E_i. So C \ {b} is already present. Adding **a** would complete cycle **C**, so Kruskal rejects **a**.

Therefore, after processing weight w_i, chosen edges = **F** $\cap$ $\mathbf{E_i}$**.**

After processing all weights (i = k), chosen edges = **F** $\cap$ $\mathbf{E_k}$ **= F**. Hence, Kruskal outputs F. Since F was an arbitrary MSF, every MSF can be produced by Kruskal's algorithm with suitable tie-breaking.

7. Let *G* be a weighted undirected graph where all edge weights are distinct. Prove that *G* has a unique MSF.

Since all edge weights are distinct, G has exactly one MSF because there is exactly one way to sort the edges by increasing weight. Hence, Kruskal has a unique forest output **F** for **G.** By the result of Item 6, every MSF can be obtained from Kruskal's algorithm, but since **F** is the unique output of **G,** then every output of Kruskal's algorithm must be equal to **F.**

8. (gawa-gawa ko lang) Assuming that the given graph is connected, prove the correctness of Boruvka's algorithm.

We hold the invariant: "After each round i,

1. E_i is a forest; and
2. There exists an MST such that $E_i \subseteq T$."do

Initialization. Initially, $E_0 = \{\varnothing\}$, which is a forest of 0 nodes and is contained in every MST. So the invariant holds.

Maintenance. We assume that the invariant holds after round i. We show that it holds after round i + 1.

Let the connected components of the forest be $C_1, \ldots, C_k$.

For each component C_j, Boruvka selects the minimum-weight edge e_j, leaving C_j, and sets

$$E_{i+1} = E_i \cup \{e_1, \ldots, e_k\}.$$

[Forest Property] Each e_j connects two different components, so no cycle is created. Thus, E_{i+1} is a forest.

[MST Containment] Let T be an MST with $E_i \subseteq T$. We fix a component C_j and its chosen edge $e_j = (u,v)$.

In T, there is a unique path from u to v, which must contain an edge e_j' leaving C_j. Consider removing e'_j and adding e_j to obtain

$$T' = (T \setminus \{e'_j\}) \cup \{e_j\}.$$

Then, T' is a spanning tree.

Since e_j is the minimum-weight edge leaving C_j and e'_j also leaves C_j,

$$w(e_j) <= w(e'_j), \text{ so } w(T') <= w(T),$$

demonstrating the minimality of T', thus an MST.

Moreover, since all previously chosen edges lie within the components, and e'_j leaves C_j, the removed edge was not previously chosen by Boruvka's algorithm. Thus, none of the earlier edges are lost, and after adding e_j, all chosen edges remain contained in the new MST.

If we apply this argument to all components being processed, then there exists an MST containing all edges of E_{i+1}.

Hence, the invariant holds after round i + 1.

Termination. Each round, every component merges with at least one other, so the number of components decreases. Eventually, only one component T remains, which is a spanning tree. Since T is contained in an MST, it must itself be an MST. Therefore, Boruvka's algorithm is correct.