Algorithms, Design & Analysis

Lecture 15: **Topological Sort, MST, UF, Kruskal's and Prim's Algorithm**

Hamza Raza & Umer Nadeem

Information Technology University

June 19, 2025

About Your Fellows

- Hi there! We are **Hamza** and **Umer**.
- We are Associate Students at ITU.

Topological Sorting Algorithm

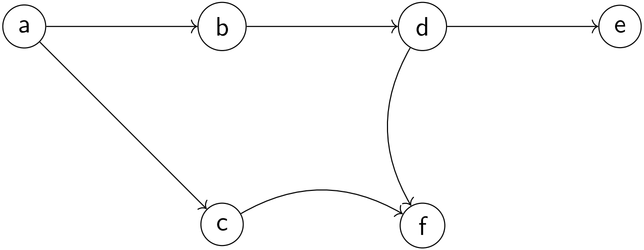
Step 1: Given a Directed Acyclic Graph (DAG) $G = (V, E)$.
Step 2: Compute the in-degree $d(v)$ for each vertex $v \in V$.
Step 3: Initialize an empty list topological_order and a queue W.
Step 4: Add all vertices v with $d(v) = 0$ to W.
Step 5: While W is not empty, repeat the following:

- a) Remove a vertex u from W.
- b) Append u to topological_order.
- c) For each vertex v such that $(u, v) \in E$:
  - i) Remove edge (u, v).
  - ii) Decrease in-degree $d(v)$ by 1.
  - iii) If $d(v) = 0$, append v to W.

Step 6: The list topological_order contains the topological ordering of the graph.

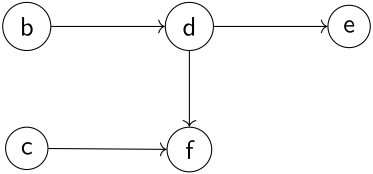
Example: Step 1 - Initial Graph

Solving an Example: Let's apply the topological sorting algorithm to this Directed Acyclic Graph.



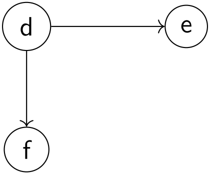
Explanation: This is the given directed acyclic graph (DAG). The goal is to perform topological sorting by iteratively removing nodes with an in-degree of 0.

Step 2: Remove *a* (in-degree 0)



Explanation: Node *a* had no incoming edges (in-degree 0), so we removed it. The edges coming out of *a* are also removed.

Step 3: Remove b, c (in-degree 0)



Explanation: Nodes b and c now have an in-degree of 0 and are removed. Their outgoing edges are also removed.

Step 4: Remove *d* (in-degree 0)

Explanation: Node *d* is now free of incoming edges, so it is removed, along with its outgoing edges.

Final Topological Orders

Two Valid Topological Orders:

Order 1: a, b, c, d, e, f
Order 2: a, c, b, d, f, e

Explanation:

- Both orders start with a since it has no incoming edges.
- After removing a, either b or c can be chosen first.
- The rest of the ordering follows the dependency constraints of the DAG.

Conclusion: There can be multiple valid topological sorts in a DAG as long as dependencies are maintained.

Minimum Spanning Tree (MST)

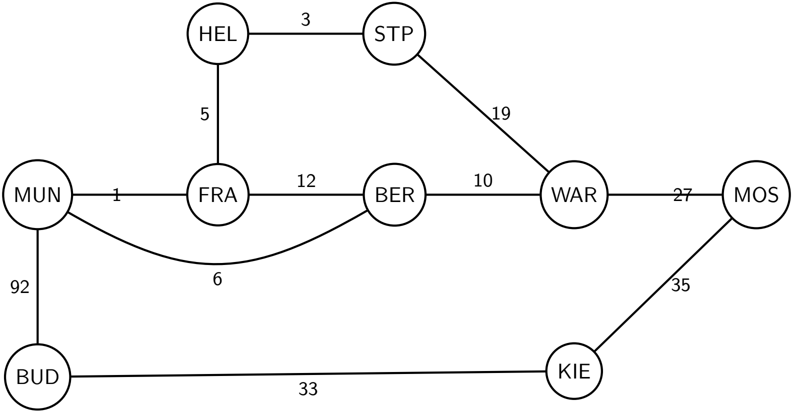
Definition: A **Minimum Spanning Tree (MST)** is a subset of the edges in a connected, weighted graph that:

- **Spans the entire graph:** It includes all the vertices of the original graph.
- **Ensures connectivity:** The MST forms a connected subgraph, meaning there are no isolated vertices.
- **Minimizes the total edge cost:** Among all spanning trees, the MST has the smallest possible sum of edge weights.
- **Avoids cycles:** Since it is a tree, it does not contain cycles.

Key Properties:

- If a graph has V vertices, the MST will have exactly $V - 1$ edges.
- There can be multiple valid MSTs for a graph.
- MSTs are commonly found using algorithms like **Kruskal's** and **Prim's**.

Cities Graph



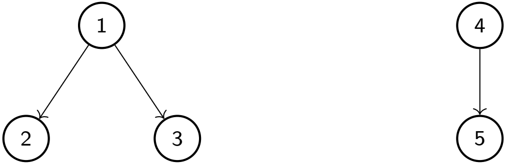
- Cities are connected through roads.
- There is a cost in kilometers between the edges, representing their distance.

Introduction to Union-Find

- Union-Find (Disjoint Set) is a data structure that helps in managing a partition of elements into disjoint sets.
- Supports two main operations:
    - **Find**: Determine which set an element belongs to.
    - **Union**: Merge two sets.
- Used in Kruskal's algorithm, network connectivity problems, etc.

Set Representation (Tree Structure)

- **Example:** Two sets represented as trees:
- **Parent nodes** point to their children.

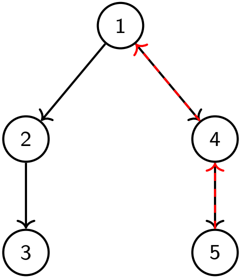


Find Algorithm

- The Find(a) function determines the representative (root) of the set containing element a.
- If a is not the root, we recursively call Find((a)).
- Using rank helps keep the tree balanced.

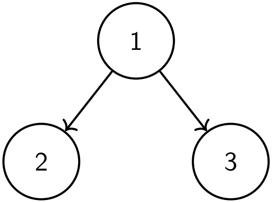
Pseudocode:

- **Find(a):**
  1. If $\pi(a) == a$, return a.
  2. Else, recursively call Find($\pi(a)$).



Union by Rank

- We use rank to keep the tree balanced.
- The node with the highest rank becomes the parent.



Root Node:
Parent = 1, Rank = 1

Left Child (2):
Parent = 1, Rank = 0

Right Child (3):
Parent = 1, Rank = 0

Union-Find Data Structure

- A **Union-Find** (Disjoint Set) data structure helps efficiently manage dynamic connectivity.
- Supports two main operations:
  - **Find(x)**: Finds the representative (root) of the set containing ‘x‘.
  - **Union(a, b)**: Merges the sets containing ‘a‘ and ‘b‘ based on **rank**.
- Uses **Union by Rank** and **Path Compression** for efficiency.

Union by Rank

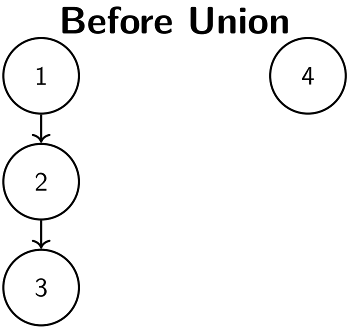
Algorithm:

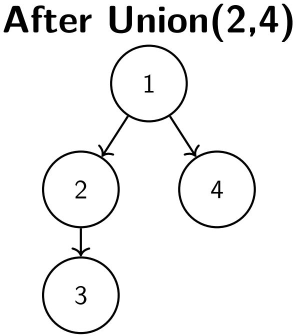
- Find the parents of ‘a‘ and ‘b‘:

$$x = \text{find}(a), \quad y = \text{find}(b)$$

- Compare their ranks:
  - If **rank(x)** $>$ **rank(y)**, set **parent(y) = x**
  - If **rank(y)** $>$ **rank(x)**, set **parent(x) = y**
  - If **rank(x) == rank(y)**, set **parent(x) = y** and increment **rank(y)++**

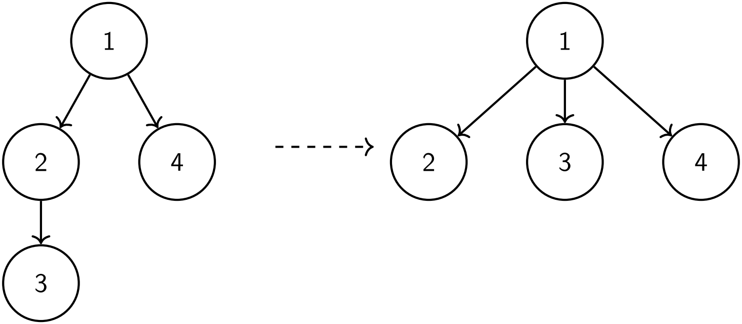
Example: Before and After Union(2,4)





Path Compression

- After applying Union, we can optimize **find(x)** using **Path Compression**.
- Instead of traversing the tree every time, we directly point all nodes to the root.
- This reduces the time complexity to **nearly O(1)** for each operation.



Reason for path compression

Why apply Path Compression?

- Initially, the tree has nodes pointing indirectly to the root:

$$3 \to 2 \to 1$$

- This increases the time complexity of the `find` operation.

After Path Compression:

- Every node directly points to the root:

$$3 \to 1, \quad 2 \to 1, \quad 4 \to 1$$

- The tree becomes shallower, optimizing future queries.
- The time complexity of `find(x)` is reduced to nearly $O(1)$ (amortized).

Conclusion: Path compression improves the efficiency of DSU by minimizing the depth of the tree.

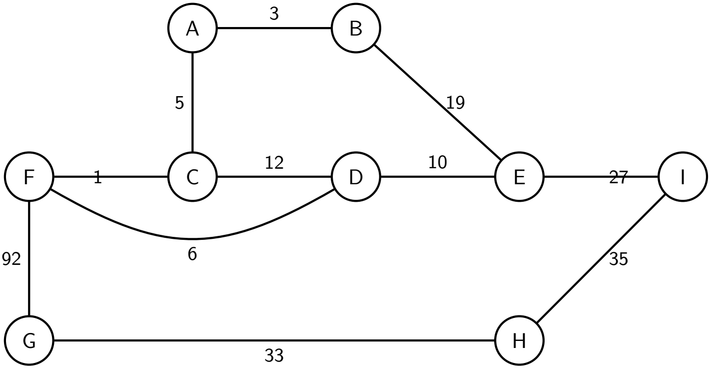
Kruskal's Minimum Spanning Tree (MST) Algorithm

Step 1: Sort all edges by weight in ascending order.
Step 2: Initialize an empty set MST to store the edges of the minimum spanning tree.
Step 3: Iterate through each edge e in sorted order:
a) If adding e to MST does not form a cycle:
i) Add e to MST.
b) If the number of edges in MST reaches $|V| - 1$, stop.
Step 4: Return MST, which contains the minimum spanning tree.

Key Concept: Cycle Detection in Kruskal's Algorithm

- To detect cycles, we use the **Union-Find (Disjoint Set)** data structure.
- Initially, each vertex is its own parent (makeset operation).
- For each edge (u, v), check if u and v belong to the same set:
  - If yes, adding the edge creates a cycle, so we discard it.
  - If no, perform a union operation to merge the sets.
- Using **path compression** and **union by rank**, we optimize the operations.

Kruskal's (MST) Algorithm Graph Example



Cycle Avoidance in Kruskal's Algorithm

- The current *MST* set: $\{1, 3, 5, 9, 12, 20, 33\}$.
- If we add edge 19, it forms a cycle.
- We skip all edges that form cycles and continue selecting the next smallest edge.
- To detect cycles efficiently, we use the **Union-Find (Disjoint Set)** data structure.

Time Complexity of Kruskal's Algorithm

Step 1: Sorting edges takes $O(E \log E)$.
Step 2: Union-Find operations (with path compression) take nearly $O(1)$.
Step 3: Overall, Kruskal's algorithm runs in $O(E \log E)$, where E is the number of edges.

Cycle Detection Approach:

- A naive approach could use Strongly Connected Components (SCC) with $O(E^2)$ complexity.
- However, using Union-Find reduces it to nearly $O(E \log V)$, making it much more efficient.

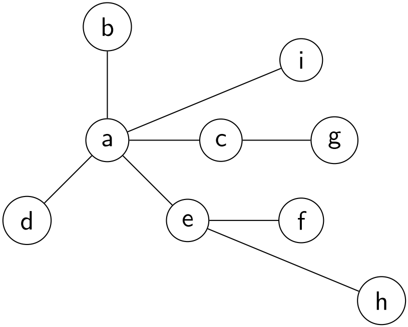
Union-Find in Kruskal's Algorithm

Step 1: Create a Union-Find (Disjoint Set) structure for all vertices.
Step 2: For each edge $e = (u, v)$:
a) If $\text{Find}(u) \neq \text{Find}(v)$ (i.e., they are in different sets):
i) Perform $\text{Union}(u, v)$ to merge the sets.
ii) Add (u, v) to MST.
Step 3: Repeat until MST contains $|V| - 1$ edges.

Efficiency:

- $\text{Find}(u)$ and $\text{Find}(v)$ each take $O(\log V)$ with path compression.
- $\text{Union}(u, v)$ also runs in $O(\log V)$ using union by rank.
- Since we process E edges, the overall complexity remains $O(E \log E)$.

Kruskal's Algorithm: Example Graph & Complexity



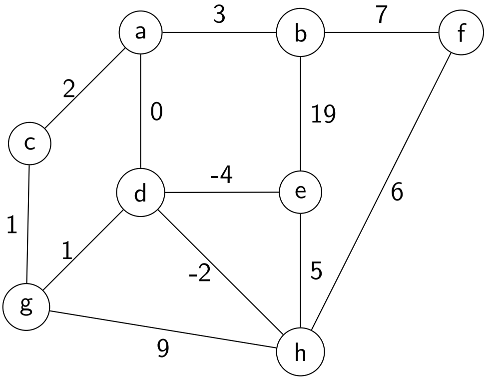
Complexity Analysis:

- Sorting edges takes $O(E \log E)$.
- Each Union-Find operation runs in $O(\log V)$.
- Overall time complexity: $O(E \log E + E \log V) = \theta(E \log V)$.
- Efficient for sparse graphs where $E \approx O(V)$.

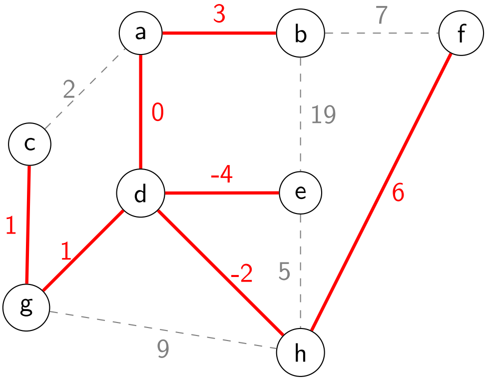
Prim's Algorithm for MST

Step 1: Start with an arbitrary vertex (e.g., $V[0]$).
Step 2: Pick the smallest outgoing edge that connects to an unvisited vertex.
Step 3: Add this edge to the Minimum Spanning Tree (MST).
Step 4: Repeat until all vertices are included in the MST.

Graph Representation



Prim's Algorithm - Minimum Spanning Tree



Prim's Algorithm - Path Explanation

Path Taken:

- Start at node D, selecting the edge $D \rightarrow E$ with the minimum weight -4.
- Select $D \rightarrow H$ (cost -2) as the next smallest edge.
- Pick $D \rightarrow A$ (cost 0), the next lowest cost edge.
- Choose $D \rightarrow G$ (cost 1) to expand the MST.
- Select $G \rightarrow C$ (cost 1), as it connects a new node with minimal cost.
- Add $A \rightarrow B$ (cost 3), as it's the cheapest edge connecting a new node.
- Finally, add $H \rightarrow F$ (cost 6), completing the MST.

Efficiency of Prim's Algorithm

- Uses a **priority queue** (typically a min-heap).
- Extracting the minimum edge takes $O(\log E)$.
- Processing all edges results in $O(E \log E)$.
- Overall complexity: $\theta(E \log E)$.

Graph Algorithms Overview

1. **Topological Sorting (DAG Only)**
   - Orders vertices such that for every edge (u, v), u appears before v.
   - **Kahn's Algorithm (BFS):** Uses in-degree and a queue.
   - **DFS-Based**: Uses a stack for reverse order.
2. **Minimum Spanning Tree (MST)**
   - Connects all vertices with the **minimum total edge weight**.
   - Two main algorithms:
     - **Kruskal's (Edge-based, uses Union-Find).**
     - **Prim's (Vertex-based, uses Priority Queue).**

Kruskal's, Prim's, and Union-Find

3. **Kruskal's Algorithm (Greedy)**
   - Sorts edges by weight and adds the smallest edge without forming a cycle.
   - Uses **Union-Find** to track connected components.
4. **Union-Find (Disjoint Set)**
   - **Find(x)**: Finds the root of the set.
   - **Union(x, y)**: Merges sets using **path compression**.
5. **Prim's Algorithm (Greedy)**
   - Starts from any vertex, picks the smallest edge using a **min-heap**.
   - Grows the MST by adding one vertex at a time.

Comparison:

- **Prim's:** Better for **dense** graphs.
- **Kruskal's:** Better for **sparse** graphs.