Minimum spanning tree
Prima
Kruskal
Dijikstra
Floyd
Ford Bellman

Lecturer: Madyar Turgenbayev

Graph vs Spanning Tree

Graph

- A graph is a **collection of vertices (nodes)** and **edges** connecting them.
- It can have **cycles**, **multiple paths**, **any shape**, directed or undirected.
- It can be **connected or disconnected**.
- Example: a road network with many possible routes.

Spanning Tree

- A spanning tree is a **subgraph** of a graph.
- It **includes all the vertices** of the graph.
- It is **connected** and has **no cycles**.
- It has exactly **V – 1 edges** (where V is the number of nodes).
- It exists **only if the graph is connected**.
- Example: a minimal set of roads connecting all cities without any loops

Minimum spanning tree

is a **weighted**, **connected**, **undirected** graph is:

- A **spanning tree** (uses all vertices, no cycles, connected)
- With the **minimum possible total edge weight** among all spanning trees of that graph.

Prim’s algorithm

- Prim's algorithm is a greedy algorithm that finds a minimum spanning tree (MST).
  - Builds an MST for a weighted, undirected graph.
  - Includes all vertices with minimum total edge weight.
  - Starts from any vertex and grows the tree one vertex at a time.
  - At each step: add the cheapest edge connecting the tree to a new vertex.

Algo visualization:

https://www.w3schools.com/dsa/dsa_algo_mst_prim.php

Prim's algorithm

key[v]: minimum edge weight to connect v.

parent[v]: parent of each vertex.

vis[v]: whether v is visited.

key[source] = 0

parent[source] = -1; others = ∞

Prim’s algorithm

- Insert the source vertex into the min-heap.
- While the min-heap is not empty:
  - Select vertex u with the smallest key; mark vis[u] = true.
  - For each vertex v adjacent to u:
    - If v not visited and weight(u, v) < key[v], update key[v].
- Continue until heap is empty → MST constructed.

Prim’s algorithm implementation demo

Prim’s algorithm (Priority Queue Complexity)

- Time & Space Complexity

  Using adjacency list + priority queue (min-heap).
  - Time Complexity: O(E log V)
    - \- Each edge may update the priority queue.
    - \- Extract-min is O(log V), repeated V times.
  - Space Complexity: O(V + E)
    - \- Graph storage: adjacency list O(V + E)
    - \- Priority queue and auxiliary arrays: O(V)

- Using adjacency matrix:
  - Time Complexity: O(V^2)
  - Space Complexity: O(V^2)

Kruskal algorithm

is a classic greedy algorithm used to find the
Minimum Spanning Tree (MST) of a connected, undirected, weighted graph

Key Ideas:

- Sort all edges by weight (ascending)
- Add the smallest edge that **does NOT form a cycle**
- Continue until exactly **V – 1 edges** are selected

Why It Works:

- Always picks the cheapest edge available (Greedy strategy)
- Uses a special data structure (DSU) to detect cycles efficiently

Visualization: https://www.w3schools.com/dsa/dsa_algo_mst_kruskal.php

Disjoint Set Union (DSU)

Disjoint Set Union (DSU) is a data structure used to track connected components.

DSU Supports Two Operations:

- **find(x)**
  Returns the representative (root) of the set containing x.
  Includes **path compression** to flatten the tree.
- **unite(x, y)**
  Merges the sets of x and y.
  Uses **union by rank** to keep trees shallow.

Disjoint Set Union (DSU)

+ Quickly checks whether adding an edge creates a **cycle**

+ If find(u) == find(v) → u and v are already connected → cycle

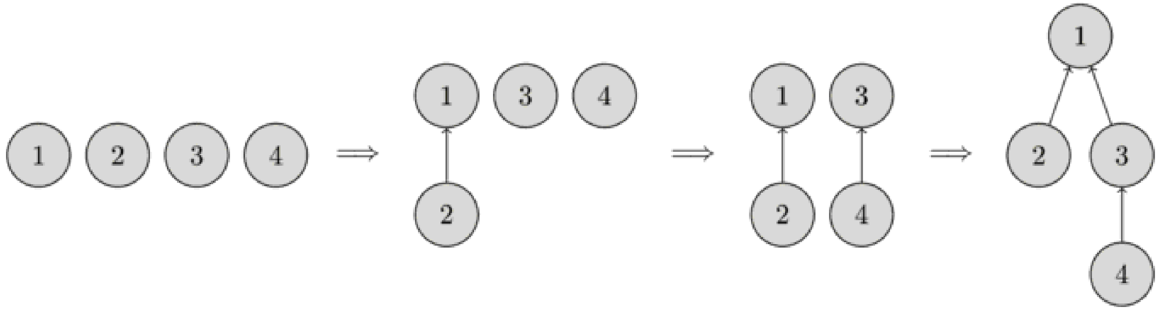
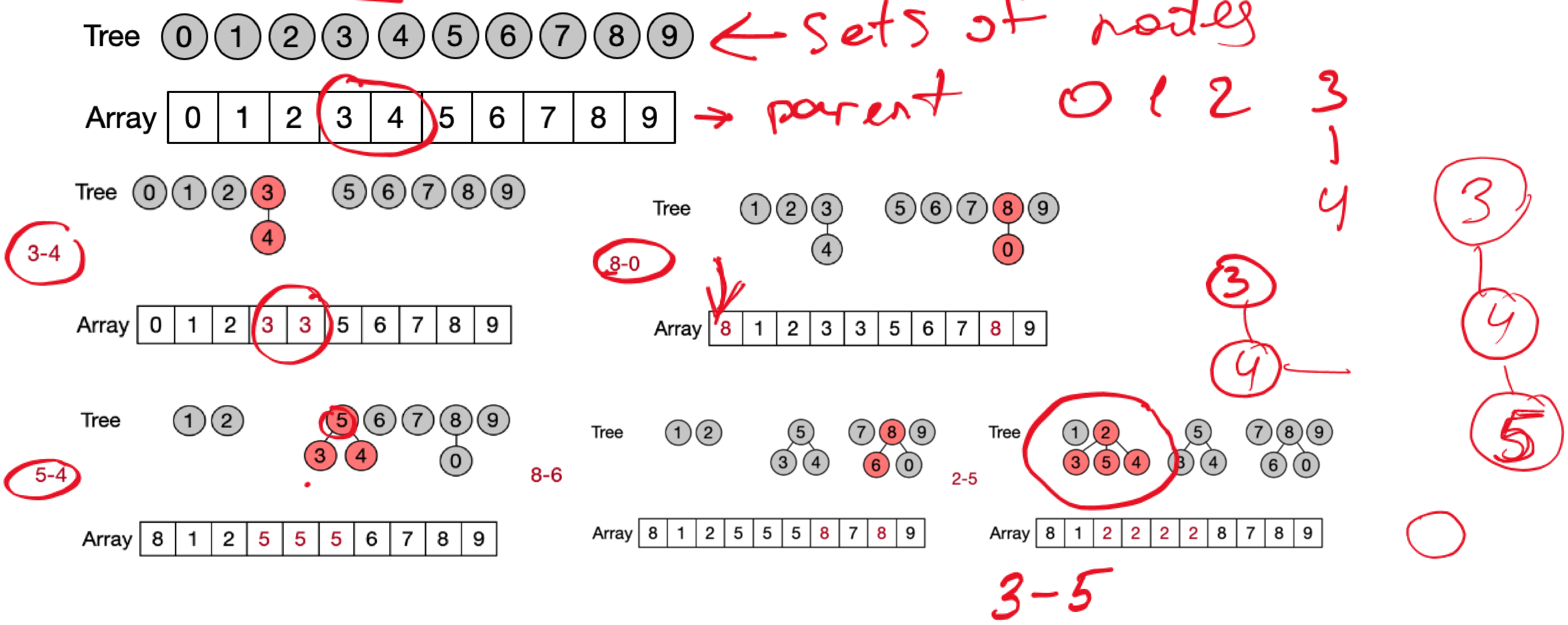


Fig 1. DSU unite representation *

* – [https://cp-algorithms.com/data_structures/disjoint_set_union.html]

← data Structure Kruskal

DSU find(x) and unit(x, y)



[https://www.francofernando.com/blog/data-structures/data%20structures/2021-06-29-disjoint-set-union]

DSU rank

Rank is an estimate of the *tree height* in a DSU set.
It helps keep the structure shallow when merging sets

When combining two sets, always attach the **smaller tree** under the **larger tree**.

```
if (rank[s1] < rank[s2]) parent[s1] = s2;
else if (rank[s1] > rank[s2]) parent[s2] = s1;
else {
    parent[s2] = s1;
    rank[s1]++;
}
```

Kruskal algorithm implementation demo

Dijkstra algorithm

Dijkstra's algorithm is a **shortest path algorithm** for graphs with **non-negative edge weights**

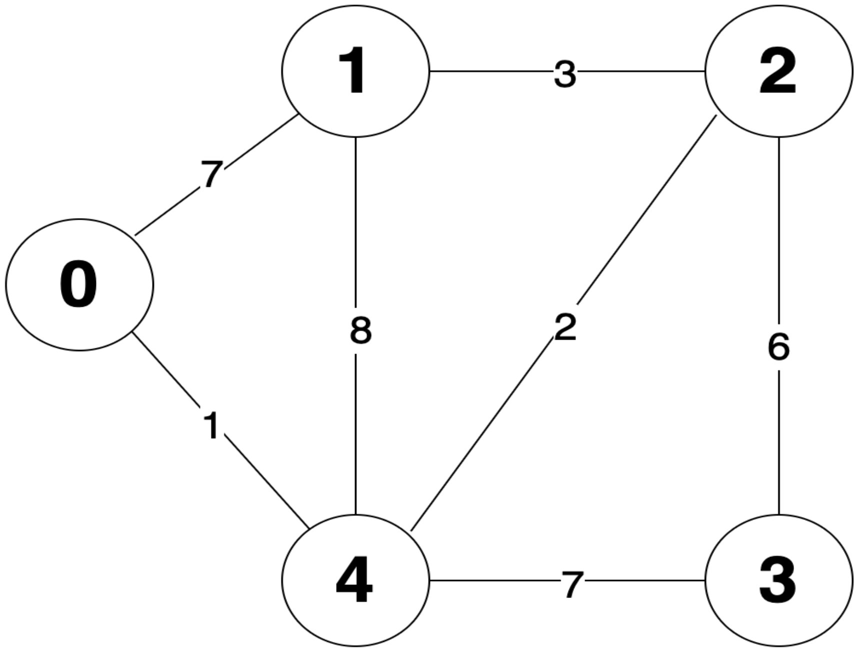
It finds the **minimum distance** from a **single source** vertex to all other vertices

Works on:

- Directed and undirected graphs
- Weighted graphs (no negative weights)

Common use cases:

- GPS navigation and route planning
- Network routing
- Pathfinding in games



Algorithm

```
d[s] ← 0
for each v ∈ V − {s}
   do d[v] ← ∞
visited ← ∅
pq ← V         //pq is a priority queue maintaining V − S
while pq is not empty:
   do u ← EXTRACT-MIN(pq)
      visited.append(u)
      for each v ∈ Adj[u]     // all neighbours of u
         do if d[v] > d[u] + w(u, v)
              then d[v] ← d[u] + w(u, v)   // relaxation step
```

Complexity

Time complexity:

- Using array / adjacency matrix: $\mathbf{O(V^2)}$
- Using min-heap + adjacency list: $\mathbf{O((V + E) \log V)}$

Space complexity: $O(V + E)$

Limitations:

- **No negative edge weights**

Implementation

Ford Bellman Algorithm

Solves **single-source shortest paths** in a weighted graph

- Works even if there are **negative edge weights**
- Input: graph $G = (V, E)$,source vertex s
- Output: shortest distance $d[v]$from s to every vertex v

(or detection of a **negative cycle** reachable from s

Idea: **Dynamic Programming / relaxation** – repeatedly improve distances over all edges

Algorithm

```
for v ∈ V
    do d[v] ← +∞
d[s] ← 0

for i ← 1 to |V| − 1
    do for (u, v) ∈ E
        if d[v] > d[u] + w(u, v)
            then d[v] ← d[u] + w(u, v)

return d
```

Complexity

Time Complexity (When graph is connected):

- Best Case: O(E) (when distance array after 1st and 2nd relaxation are the same)
- Average Case: O(V*E)
- Worst Case: O(V*E)

Time Complexity (When graph is disconnected): O(E*(V^2))

Auxiliary Space: O(V)

Implementation

Floyd–Warshall algorithm

helps determine the **shortest path distance** between all pair of nodes **i** and **j** in the graph

The algorithm relies on the **principle of optimal substructure**, meaning:

- If the shortest path from i to j passes through some vertex k, then the path from i to k and the path from k to j must also be shortest paths
- The iterative approach ensures that by the time vertex k is considered, all shortest paths using only vertices 0 to k-1 have already been computed

Complexity

Time Complexity: $O(V^3)$, where V is the number of vertices in the graph and we run three nested loops each of size V

Auxiliary Space: $O(1)$