LECTURE 6 – DISJOINT SET UNION (UNION-FIND) AND MINIMUM SPANNING TREES (KRUSKAL / PRIM)

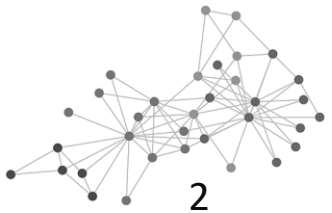
Olzhas Aimukhambetov
olzhas.aimukhambetov@astanait.edu.kz

ASTANA IT UNIVERSITY

ASTANA IT UNIVERSITY

CONTENT

1. Graph Connectivity
2. Disjoint Set Union (DSU): Concept
3. DSU Operations and Efficiency
4. Union by Rank & Path Compression
5. Java Implementation
6. Minimum Spanning Tree: Definition & Intuition
7. Kruskal's Algorithm (with DSU)
8. Prim's Algorithm (Priority Queue)
9. Comparing Kruskal vs Prim
10. Engineering Notes & Applications



ASTANA IT UNIVERSITY

GRAPH CONNECTIVITY PROBLEMS

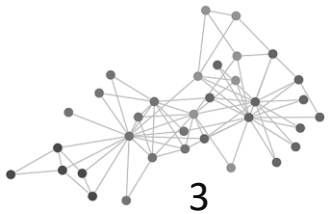
Many graph algorithms rely on quickly answering: “Are these two vertices connected?”

Examples:

- Detecting cycles in undirected graphs
- Building Minimum Spanning Trees (MST)
- Tracking connected components in dynamic networks
- Clustering, percolation, union operations in simulations

Naive approach: recompute connectivity by BFS/DFS after each change → O(V+E) per query.

Better: maintain dynamic partition of vertices (DSU) → near O(1) per query.



DISJOINT SET UNION: CONCEPT AND OPERATIONS

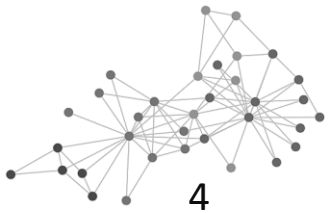
Disjoint Set Union (DSU) / Union-Find manages a collection of disjoint sets under two operations:

- find(x) — returns representative (root) of x's set
- union(x, y) — merges two sets if distinct

Representation:

- Each set represented by a tree; root is the leader
- Initially each vertex is its own parent
- With heuristics (union by rank, path compression) operations are extremely fast

Typical usage: dynamic connectivity queries, cycle detection, Kruskal's algorithm.



UNION BY RANK AND PATH COMPRESSION

Two heuristics ensure almost-constant performance:

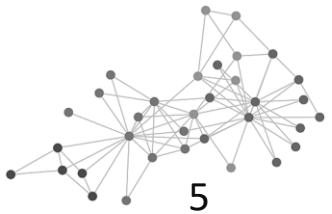
Union by Rank (or Size):

• Attach smaller tree under root of larger tree to keep depth small.

Path Compression:

• During find(x), make each node on path point directly to root → flattens tree.

Complexity: each operation ≈ **O(α(n))** where **α** is inverse Ackermann (≤ 5 for all practical n).



ASTANA IT UNIVERSITY

JAVA IMPLEMENTATION

```java
class DSU {
    private int[] parent;
    private int[] rank;

    public DSU(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    public int find(int x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]); // path compression
        return parent[x];
    }

    public void union(int x, int y) {
        int rootX = find(x), rootY = find(y);
        if (rootX == rootY) return;

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

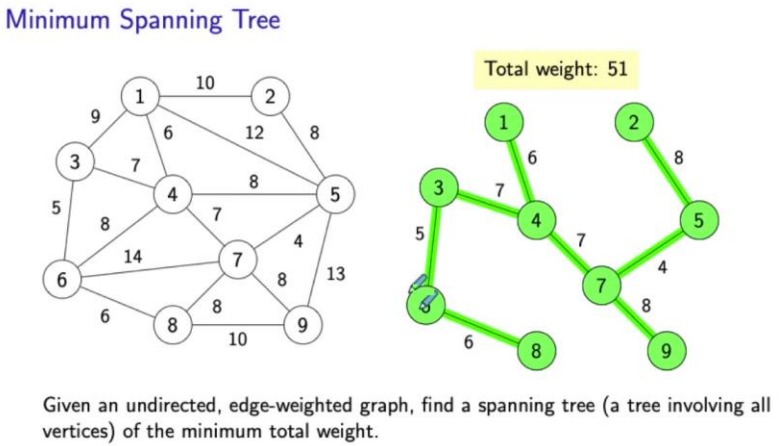
MINIMUM SPANNING TREE: DEFINITION AND INTUITION

Minimum Spanning Tree (MST): a subset of edges connecting all vertices with minimal total weight.

Formal: For a connected weighted undirected graph G(V,E), MST $\subseteq$ E connects all vertices with minimum $\sum$ w(e).

Properties:

- A tree (no cycles) that spans all vertices
- Unique if all edge weights are distinct
- Fundamental in network design, clustering, approximation algorithms



Key insight: repeatedly add the minimum safe edge that does not form a cycle.

KRUSKAL'S ALGORITHM

Kruskal's algorithm — greedy, edge-based:

1. Sort all edges by weight (ascending).

2. Initialize DSU with each vertex separate.

3. For each edge (u,v,w) in sorted order:

   • if find(u) != find(v): add edge to MST; union(u,v)

   • else skip (would form cycle)

Correctness: by cut and cycle properties — smallest edge across any cut is safe.

Complexity:

• Sorting: O(E log E)

• DSU ops: O(E α(V)) ≈ O(E)

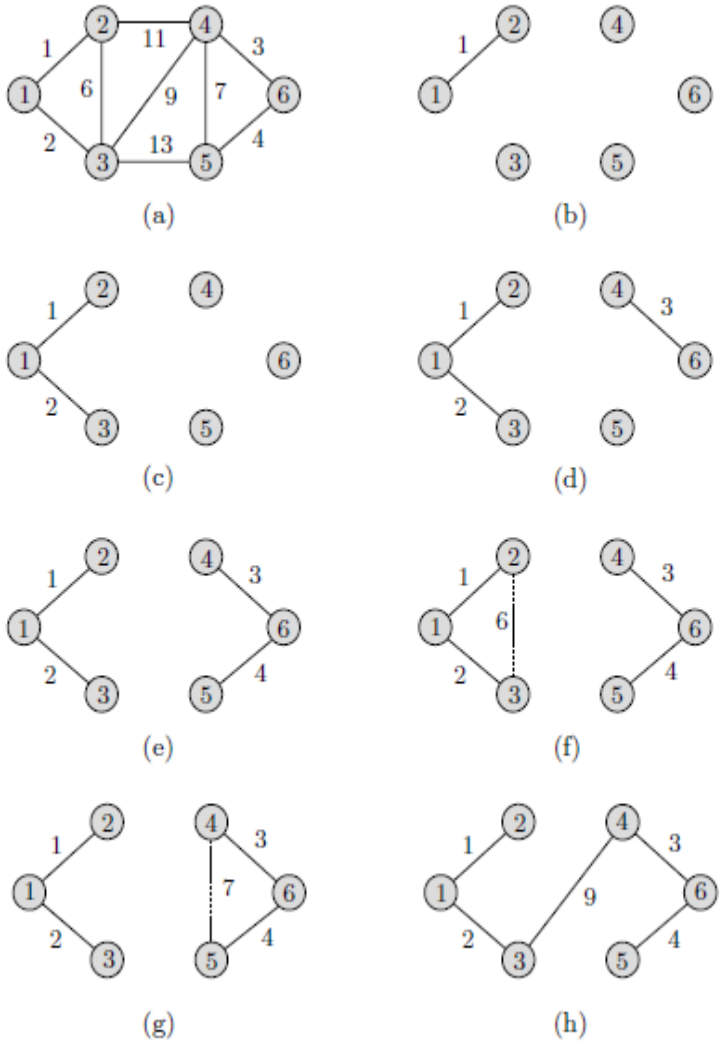


Fig. 7.4. An example of Kruskal Algorithm.

KRUSKAL'S ALGORITHM (WITH DSU)

Input: A weighted connected undirected graph $G = (V,E)$ with n vertices.

Output: The set of edges T of a minimum cost spanning tree for G.

- 1. Sort the edges in E by nondecreasing weight.
- 2. **for** each vertex $v \in V$
- 3. makeset({v})
- 4. **end for**
- 5. $T = \{\}$
- 6. **while** $|T| < n - 1$
- 7. Let (x, y) be the next edge in E.
- 8. **if** find(x) = find(y) **then**
- 9. Add (x, y) to T
- 10. union(x, y)
- 11. **end if**
- 12. **end while**

PRIM'S ALGORITHM (PRIORITY QUEUE)

Prim's algorithm — grow-from-vertex approach:

- Start from any vertex.
- Maintain min-priority queue of edges that cross from current MST to non-tree vertices.
- Repeatedly extract min edge and add the adjacent vertex if it's not in MST.

Implementation notes:

- Use adjacency list + PriorityQueue<Edge>
- Track visited[] to avoid adding same vertex twice

Complexity: O(E log V) with binary heap (improvable with Fibonacci heap).

When to use: practical for dense graphs or when adjacency is given.

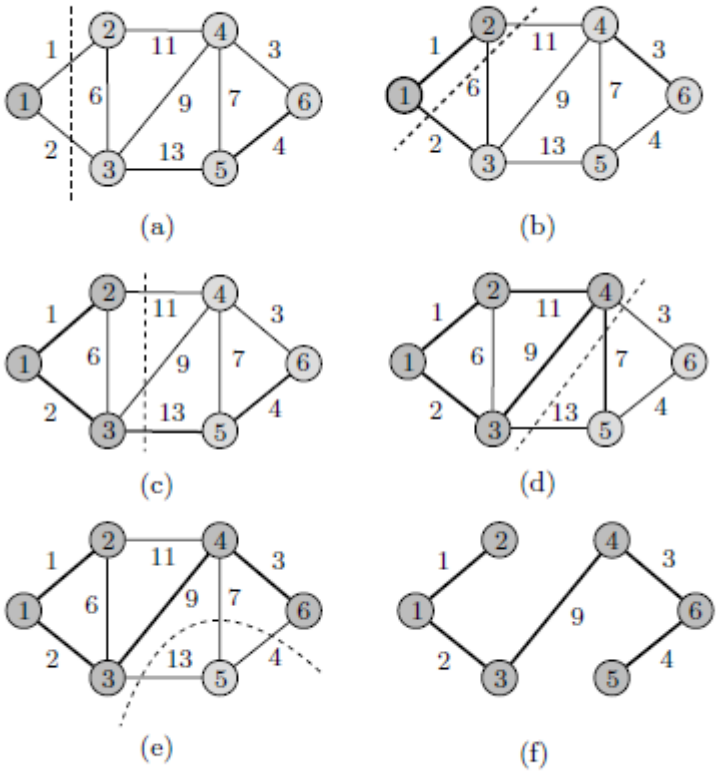


Fig. 7.5. An example of Prim's algorithm.

PRIM'S ALGORITHM (PRIORITY QUEUE)

Input: A weighted connected undirected graph $G = (V,E)$, where $V = \{1, 2, \ldots, n\}$.

Output: The set of edges T of a minimum cost spanning tree for G.

- 1. $T \leftarrow \{\}$; $X \leftarrow \{1\}$; $Y \leftarrow V - \{1\}$
- 2. **for** $y \leftarrow 2$ **to** n
- 3. **if** y adjacent to 1 **then**
- 4. $N[y] \leftarrow 1$
- 5. $C[y] \leftarrow c[1, y]$
- 6. **else** $C[y] \leftarrow \infty$
- 7. **end if**
- 8. **end for**
- 9. **for** $j \leftarrow 1$ **to** $n - 1$ {find $n - 1$ edges}
- 10. Let $y \in Y$ be such that $C[y]$ is minimum
- 11. $T \leftarrow T \cup \{(y,N[y])\}$ {add edge $(y,N[y])$ to T}
- 12. $X \leftarrow X \cup \{y\}$ {add vertex y to X}
- 13. $Y \leftarrow Y - \{y\}$ {delete vertex y from Y }
- 14. **for** each vertex $w \in Y$ that is adjacent to y
- 15. **if** $c[y,w] < C[w]$ **then**
- 16. $N[w] \leftarrow y$
- 17. $C[w] \leftarrow c[y,w]$
- 18. **end if**
- 19. **end for**

COMPARISON & PRACTICAL CONSIDERATIONS

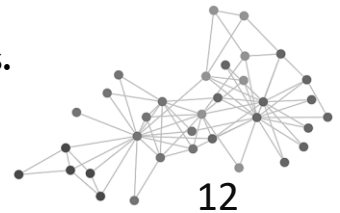
Comparison:

- Kruskal: edge-based (global), uses DSU. Complexity O(E log E). Best for sparse graphs.
- Prim: vertex-based (local), uses priority queue. Complexity O(E log V). Best for dense graphs.

Other notes:

- Both produce same total weight for MST if graph connected.
- Use Kruskal when edge-list is available or graph is sparse.
- Use Prim when adjacency lists/matrix are convenient (dense graph).
- For very large graphs, consider external sorting / parallelization.

Applications: network design, clustering (single-linkage), approximation algorithms.



COMPARISON (KRUSKAL VS. PRIM)

Aspect	Kruskal	Prim
Strategy	Edge-based (sort + union)	Vertex-based (grow MST)
Data structure	DSU	Priority Queue
Best for	Sparse graphs	Dense graphs
Complexity	O(E log E)	O(E log V)
Edge addition	Global order	Local minimum edge

ASTANA IT UNIVERSITY

THANK YOU!