Minimum Cost Spanning Trees

Varun Kumar

July 11, 2025

Contents

Python & C++ Programs

List of Algorithms

List of Tables

1 Minimum Cost Spanning Tree (MST)

A **Spanning Tree** of a connected, undirected graph is a subgraph that includes all the vertices with the minimum number of edges (i.e., $V - 1$ edges). A **Minimum Cost Spanning Tree** is a spanning tree with the minimum total edge weight.

It was first proposed by Otakar Borůvka[1] in 1926 on a paper with title "*Onacertainminimalproblem*"

Applications

- Network design (LAN, telecommunication)[3]
- Circuit design
- Clustering and approximation algorithms

1.1 Minimum Cost Spanning Tree

A Minimum Cost Spanning Tree (MCST) connects all vertices of a graph with the smallest possible total edge weight and no cycles.

1.2 General Approach (Greedy Strategy)

Algorithm 1 General MCST Algorithm

Input: Connected undirected weighted graph $G = (V, E)$
Output: Set of edges forming the MCST
function MCST(G)
    Initialize an empty set `MST`
    Initialize totalCost $\leftarrow 0$
    while `MST` does not have $V - 1$ edges **do**
        Pick the minimum weight edge (u, v) that doesn't form a cycle in `MST`
        Add (u, v) to `MST`
        Update totalCost $\leftarrow$ totalCost + weight of edge
    end while
    return `MST`, totalCost
end function

Note:

- Step 5 uses cycle detection via DSU (Kruskal) or visited set (Prim).
- The greedy approach ensures local optimality leads to global optimality.

Explanation of Pseudocode

The general idea behind the greedy strategy for Minimum Cost Spanning Tree (MCST) is simple and efficient. The algorithm starts with an empty tree and keeps adding the smallest weight edge that does not create a cycle. This process continues until the tree spans all the vertices (i.e., has exactly $V - 1$ edges).

Steps Explained in Plain English:

- Start with an empty set to store the MST edges.
- Keep track of the total cost (initially zero).
- Until we have exactly $V - 1$ edges:
  - Look for the smallest edge that connects two different parts of the graph and does not create a cycle.
  - Add this edge to the MST.
  - Add its weight to the total cost.
- Once the MST has $V - 1$ edges, the algorithm is complete.
- Return the final MST and its total cost.

This approach is the foundation of well-known algorithms like **Prim's** and **Kruskal's**, which implement the greedy strategy in different ways.

1.3 DP Approach to MCST (Theoretical)

The Minimum Cost Spanning Tree (MCST) problem seeks to connect all vertices of a connected, undirected, and weighted graph with the minimum total edge cost, forming a tree (i.e., no cycles). Traditional approaches like **Prim's** and **Kruskal's** algorithms efficiently solve this problem using greedy strategies.

However, from a theoretical perspective, one can formulate MCST as a **dynamic programming (DP)** problem using *bitmasking* to represent subsets of vertices. Although this approach is not practical for large graphs due to its exponential time complexity $O(V^2 \cdot 2^V)$, it provides insight into the relationship between combinatorial optimization and DP paradigms.

The DP approach serves as a useful conceptual tool in algorithm theory, NP-completeness discussions, and for solving small graph instances in academic contexts.

Algorithm 2 DP approach to MCST (Exponential Time)

Input: Graph $G = (V, E)$
Let: $dp[S][u]$ = minimum cost to connect subset $S \subseteq V$ ending at vertex u
Initialize all $dp[S][u] \leftarrow \infty$
for each vertex $u \in V$ **do**
    $dp[2^u][u] \leftarrow 0$ ▷ Cost to start at each node
end for
for each subset $S \subseteq 2^V$ **do**
    for each vertex $u \in V$ where S includes u **do**
        for each neighbor v of u **do**
            if $v \notin S$ **then**
                $newS \leftarrow S \cup \{v\}$
                $dp[newS][v] \leftarrow \min(dp[newS][v], dp[S][u] + \text{weight}(u, v))$
            end if
        end for
    end for
end for
return $\min_u dp[2^V - 1][u]$

Note

This algorithm is exponential and mainly of theoretical interest. For practical purposes, Prim's or Kruskal's algorithm is always preferred.

DP Approach to MCST (Exponential Time)

This algorithm uses a Dynamic Programming (DP) approach to find the Minimum Cost Spanning Tree (MCST). It is based on the idea of exploring all subsets of vertices and building the solution incrementally. Although not efficient for large graphs (exponential time), it is conceptually useful and similar in spirit to the Held-Karp algorithm for TSP.

Plain English Explanation of the Algorithm:

- The graph is represented as $G = (V, E)$, where V is the set of vertices and E is the set of edges.
- Define a DP table: `dp[S][u]` represents the minimum cost to connect all vertices in subset $S \subseteq V$, ending at vertex u.
- Initially, set all `dp[S][u]` to ∞ (unknown or unreachable).
- For each vertex u, set `dp[{u}][u] = 0`, meaning starting at node u with only that node in the subset has zero cost.
- For all subsets S of vertices:
  - For each vertex u in subset S:
    - For each neighbor v of u that is not already in S:
      - Let `newS = S \cup {{v}}`, which means $S \cup \{v\}$.
      - Update the DP value for `dp[newS][v]` as the minimum of its current value or the cost of extending `dp[S][u]` by the edge (u, v).
- After all subsets have been processed, the final answer is the minimum of `dp[2^V - 1][u]` over all u, which represents the cost to connect all nodes, ending at any node u.

Note: This algorithm runs in exponential time, $O(V \cdot 2^V)$, and is used mainly for theoretical or very small graphs.

1.4 When to Use Dynamic Programming for Minimum Cost Spanning Tree (MCST)

Dynamic Programming (DP) is generally not the most efficient method for solving MCST problems in practice. However, it becomes relevant in:

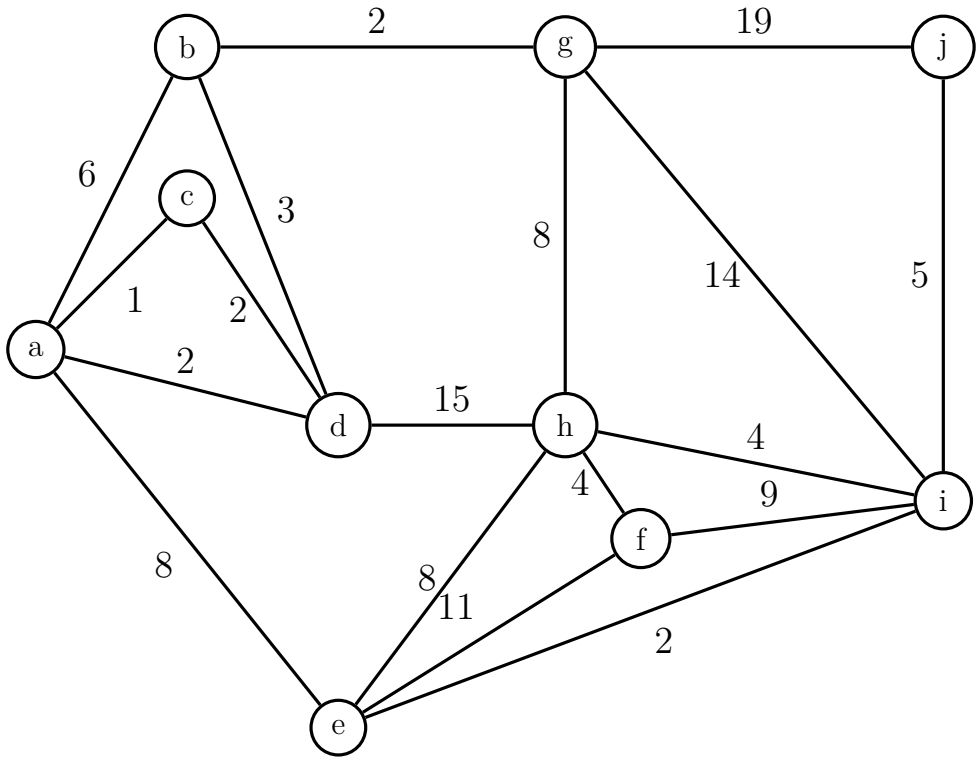
- **Theoretical analysis:** DP helps in understanding the structure and properties of spanning trees and optimal substructure.
- **Special versions:** Problems like the *Travelling Salesman Problem (TSP)* or *Steiner Tree*, which are extensions or variants of MCST, often require DP.
- **Exhaustive optimization:** When all possible spanning trees need to be analyzed (e.g., for robustness or reliability), DP can be used to cache results and reduce recomputation.
- **Dynamic Graphs:** If the graph changes (edge insertions/deletions), certain DP-based or memoization strategies can help update the MCST incrementally.

In general, greedy algorithms[2] like **Prim's** and **Kruskal's** are preferred due to their simplicity and efficiency, but DP is valuable when the problem cannot be solved optimally using greedy methods alone.

1.5 GATE CSE 2003

What is the weight of a minimum spanning tree of the following graph?

Weighted Graph



(a) 29 (b) 31

(c) 38 (d) 41

1.6 GATE CSE 2007

Let w be the minimum weight among all edge weight in an undirected commected graph of weight w. Which of the following is false?

(a) There is a minimum spanning tree containing e.

(b) If e is not in a minimum spanning tree T, then in the cycle formed by adding e to T, all edges ahve the same weight w

(c) Every minimum spanning tree has an edge of weight w

(d) e is present in every minimum spanning tree

2 Prim's Algorithm

Prim's[6] algorithm grows the MST one edge at a time, starting from an arbitrary node. It always adds the minimum weight edge that connects a visited node to an unvisited node.

2.1 Algorithm Steps

- Initialize a priority queue (min-heap) with the starting vertex.
- Repeat until all vertices are included:
  - Extract the edge with minimum weight.
  - If the adjacent node is unvisited, add it to the MST.

2.2 Core Idea

Prim's Algorithm constructs a Minimum Spanning Tree (MST) by:

- Starting from an arbitrary node.
- Maintaining a priority queue to track minimum weight edges.
- At each step, selecting the smallest edge that connects the MST to a new vertex.
- Repeating until all vertices are included.

Pseudocode

Algorithm 3 Prim's Algorithm

```
1:  Input: Weighted undirected graph G = (V, E)
2:  Output: Minimum Cost of Spanning Tree
3:  function PRIM(Graph G, Start Vertex s)
4:      Initialize a min-heap Q
5:      visited ← empty set
6:      cost ← 0
7:      Insert (0, s) into Q                          ▷ Start with source node
8:      while Q is not empty do
9:          (w, u) ← Extract-Min from Q
10:         if u ∉ visited then
11:             Add u to visited
12:             cost ← cost + w
13:             for each neighbor (v, weight) of u do
14:                 if v ∉ visited then
15:                     Insert (weight, v) into Q
16:                 end if
17:             end for
18:         end if
19:     end while
20:     return cost
21: end function
```

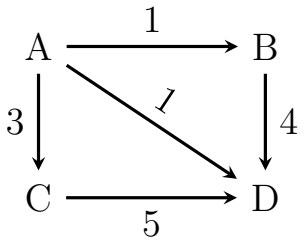
Time and Space Complexity

- Time: $O(E \log V)$ with min-heap and adjacency list
- Space: $O(V)$

2.3 Prim's Algorithm: Step-by-Step Dry Run Example

Graph:

Directed Weighted Graph



Step-by-Step Execution (Start from A)

Step	Action and Explanation
1	Start at vertex A. Add A to visited set. Insert its neighbors (B, 1), (C, 3), (D, 1) into priority queue.
2	Extract min edge (A–B, 1). B is unvisited. Add B to MST and visited set. Push B's neighbor (D, 4). Queue now: (D, 1), (C, 3), (D, 4)
3	Extract min edge (A–D, 1). D is unvisited. Add D to MST. Push its neighbor (C, 5). Queue now: (C, 3), (D, 4), (C, 5)
4	Extract min edge (A–C, 3). C is unvisited. Add C to MST. All vertices visited.

Final MST Edges and Cost

- Edges in MST: (A–B, 1), (A–D, 1), (A–C, 3)
- Total Cost $= 1 + 1 + 3 = 5$

2.4 Python Code

```python
import heapq

def prim(graph, start):
    visited = set()
    min_heap = [(0, start)]
    mst_cost = 0

    while min_heap:
        weight, u = heapq.heappop(min_heap)
        if u not in visited:
            visited.add(u)
            mst_cost += weight
            for v, w in graph[u]:
                if v not in visited:
                    heapq.heappush(min_heap, (w, v))

    return mst_cost
```

Listing 1: Prim's Algorithm in Python

2.5 C++ Code

```cpp
#include <bits/stdc++.h>
using namespace std;

int prim(vector<vector<pair<int,int>>> &graph, int V) {
    vector<bool> visited(V, false);
    priority_queue<pair<int,int>, vector<pair<int,int>>,
       greater<>> pq;
    pq.push({0, 0});
    int cost = 0;

    while (!pq.empty()) {
        auto [w, u] = pq.top(); pq.pop();
        if (!visited[u]) {
            visited[u] = true;
            cost += w;
            for (auto [v, wt] : graph[u])
                if (!visited[v])
                    pq.push({wt, v});
        }
    }
    return cost;
}
```

Listing 2: Prim's Algorithm in C++

2.6 Boundary Conditions and Limitations of Prim's Algorithm

- **Disconnected Graph:** Prim's algorithm assumes the graph is connected. If the graph has disconnected components, it cannot produce a valid MST.
- **Negative Weight Edges:** Although Prim's can technically work with negative weights, it may behave unexpectedly or be confused with shortest path problems.
- **Multiple Components:** If the graph has multiple disconnected subgraphs, Prim's will only cover one. To handle this, run Prim's separately on each component to form a minimum spanning forest.
- **Directed Graphs:** Prim's algorithm only works on undirected graphs. Using it on directed graphs results in invalid or incomplete solutions.
- **Dense Graph + Naïve Implementation:** If a dense graph is used without an efficient priority queue (e.g., min-heap), the time complexity becomes $O(V^2)$, which is inefficient.
- **Tie-breaking in Equal Weights:** If multiple edges have the same minimum weight, the MST might not be unique. The final result depends on edge ordering or the heap behavior.
- **Integer Overflow:** Large edge weights may exceed variable limits like `int`, causing incorrect results or crashes in implementation.
- **Floating Point Precision:** When using decimal weights, rounding errors can lead to incorrect edge selection due to precision limits.

2.7 Where Prim's Algorithm is Not Applicable

- Finding shortest paths from a source to all vertices (use Dijkstra or Bellman-Ford instead).
- Working on directed graphs or graphs with cycles.
- Online/dynamic edge insertion/removal problems.

2.8 Correct Use Case for Prim's Algorithm

Prim's Algorithm is ideal when working with:

- A **connected**, **undirected**, **weighted** graph
- No isolated vertices
- Static graph (no live updates)
- Need to minimize total connection cost (not shortest paths)

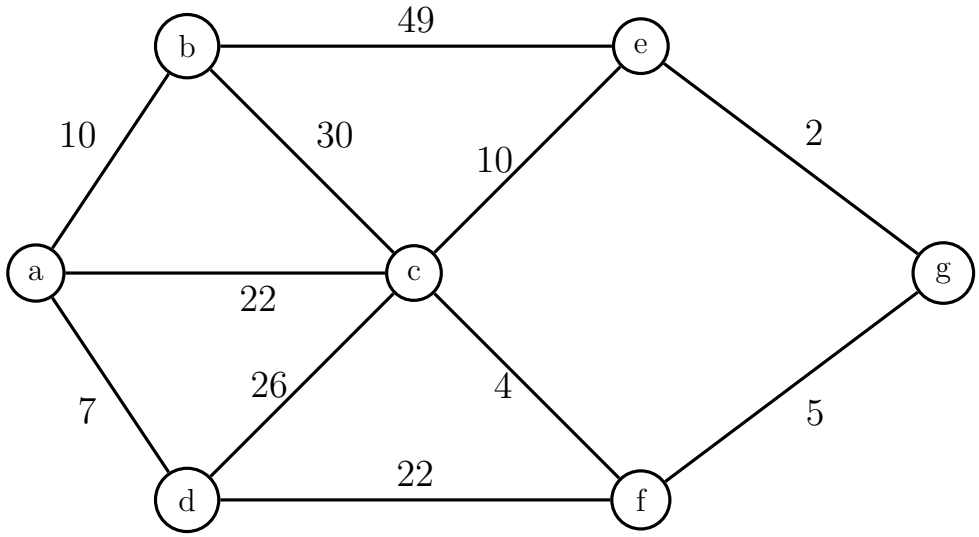
Clarification on Point 4 (Total Connection Cost vs. Shortest Path):

- Suppose you want to connect all houses in a village with electrical wires at minimum cost — **Prim's** is the correct choice.
- But if you want to go from your house to the hospital in the least time — use **Dijkstra's** or **BFS/DFS**, depending on the case.

2.9 GATE CSE 2004

Consider the following graph

Weighted Graph



Using Prim's Algorithm to construct a minimmum spanning tree starting with node A, Which one of the following sequences of edges represents a possibel order in which the edges would be added to construct the minimum spanning tree?

(a) (e, g), (c, f), (f, g), (a, d), (a, b), (a, c)

(b) (a, d), (a, b), (a, c), (c, f), (g, e), (f, g)

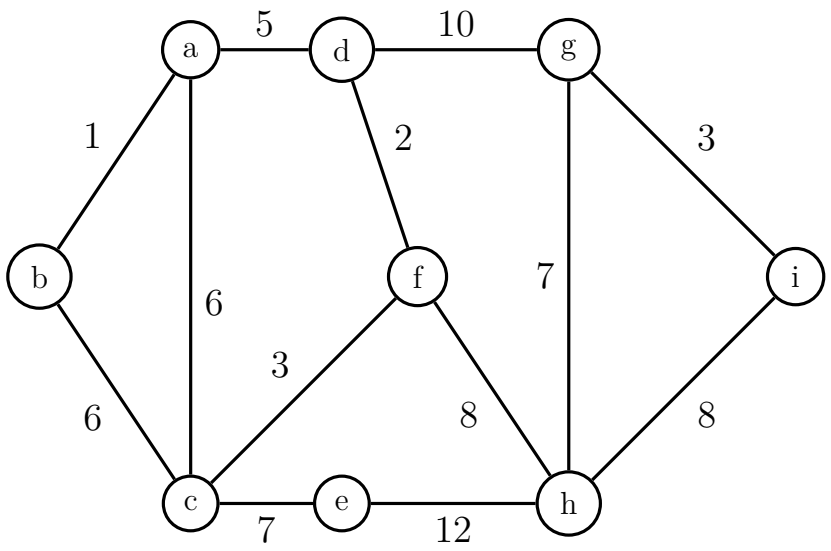
(c) (a, b), (a, d), (d, f), (f, g), (g, e), (f, c)

(d) (a, d), (a, b), (d, f), (f, c), (f, g), (g, e)

2.10 GATE CSE 2008

Apply Prim's Algorithm and provide the edge in order which they were added

Weighted Graph



Ans:- (d,f), (f,c), (d,a), (a,b), (c,e), (f,h), (g,h), (g,i) when started from vertex d

3 Kruskal's Algorithm

Kruskal's[5] algorithm sorts all edges in increasing order and adds them one by one to the MST if they do not form a cycle. It uses the Disjoint Set Union (DSU[7]) data structure.

3.1 Algorithm Steps

- Sort all edges in ascending order.
- Initialize each node as its own set.
- Iterate through the edges and add them to the MST if they connect different sets.

3.2 Core Idea

- Sort all edges in increasing order of weight.
- Use the **Disjoint Set Union (DSU)** data structure to detect cycles.
- Keep adding the next lightest edge that doesn't cause a cycle.
- Stop when $V - 1$ edges have been added (where V is the number of vertices).

Time and Space Complexity

- **Time Complexity:** $O(E \log E)$, dominated by sorting and union-find operations.
- **Space Complexity:** $O(V)$ for DSU.

3.3 Pseudocode

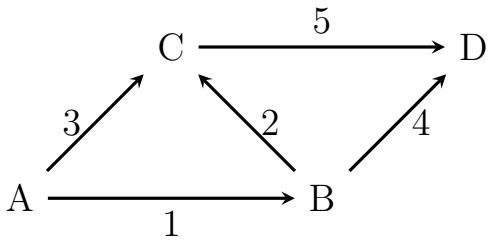
Algorithm 4 Kruskal's Algorithm

```
1: Input: Graph G = (V, E)
2: Output: MST set of edges
3: function KRUSKAL(V, E)
4:     Sort edges E in ascending order by weight
5:     Initialize DSU with each vertex in its own set
6:     MSTEdges ← empty set
7:     for each edge (u, v) in sorted E do
8:         if FIND(u) ≠ FIND(v) then
9:             Add (u, v) to MSTEdges
10:            UNION(u, v)
11:        end if
12:    end for
13:    return MSTEdges
14: end function
15: function FIND(x)
16:    if parent[x] ≠ x then
17:        parent[x] ← FIND(parent[x])                ▷ Path compression
18:    end if
19:    return parent[x]
20: end function
21: function UNION(x, y)
22:    xRoot ← FIND(x)
23:    yRoot ← FIND(y)
24:    if rank[xRoot] < rank[yRoot] then
25:        parent[xRoot] ← yRoot
26:    else if rank[yRoot] < rank[xRoot] then
27:        parent[yRoot] ← xRoot
28:    else
29:        parent[yRoot] ← xRoot
30:        rank[xRoot] ← rank[xRoot] + 1
31:    end if
32: end function
```

3.4 Dry Run Example: Kruskal's Algorithm

Graph:

Directed Weighted Graph



Vertices: A, B, C, D **Edges:** 5

Step 1: Sort edges by weight

- (A–B, 1)
- (B–C, 2)
- (A–C, 3)
- (B–D, 4)
- (C–D, 5)

Step 2: Initialize DSU

Vertex	Parent
A	A
B	B
C	C
D	D

Step 3: Process edges one by one

- **(A–B, 1)**: Find(A) = A, Find(B) = B ⇒ Different sets ⇒ Add edge to MST ⇒ Union(A, B) ⇒ Parent[B] = A
- **(B–C, 2)**: Find(B) = A, Find(C) = C ⇒ Different ⇒ Add to MST ⇒ Union(B, C) ⇒ Parent[C] = A

- **(A–C, 3)**: Find(A) = A, Find(C) = A ⇒ Same set ⇒ Ignore (would form cycle)
- **(B–D, 4)**: Find(B) = A, Find(D) = D ⇒ Different ⇒ Add to MST ⇒ Union(B, D) ⇒ Parent[D] = A
- **(C–D, 5)**: Find(C) = A, Find(D) = A ⇒ Same set ⇒ Ignore

Final MST:

- (A–B, 1)
- (B–C, 2)
- (B–D, 4)

Total Cost: $1 + 2 + 4 = 7$

Final DSU Table:

Vertex	Parent
A	A
B	A
C	A
D	A

3.5 Python Code

```python
def find(parent, x):
    if parent[x] != x:
        parent[x] = find(parent, parent[x])  # Path compression
    return parent[x]

def union(parent, rank, x, y):
    xroot = find(parent, x)
    yroot = find(parent, y)
    if xroot != yroot:
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        else:
            parent[yroot] = xroot
            if rank[xroot] == rank[yroot]:
                rank[xroot] += 1

def kruskal(V, edges):
    edges.sort(key=lambda x: x[2])  # Sort by weight
    parent = list(range(V))
    rank = [0] * V
    cost = 0

    for u, v, w in edges:
        if find(parent, u) != find(parent, v):
            union(parent, rank, u, v)
            cost += w

    return cost

# ============================
# User Input Section
# ============================

V, E = map(int, input("Enter number of vertices and edges: ").split())
edges = []

print("Enter edges in format: u v w (0-indexed)")
for _ in range(E):
    u, v, w = map(int, input().split())
    edges.append((u, v, w))

# Run Kruskal's Algorithm
mst_cost = kruskal(V, edges)
print("Minimum Cost of Spanning Tree:", mst_cost)
```

Listing 3: Kruskal's Algorithm in Python

```
# Number of vertices
V = 4
# List of edges (u, v, weight)
edges = [
    (0, 1, 10),
    (0, 2, 6),
    (0, 3, 5),
    (1, 3, 15),
    (2, 3, 4)
]
# Run Kruskal's algorithm and print MST cost
print("Minimum Cost of Spanning Tree:", kruskal(V, edges))
```

3.6 Working of Kruskal's Algorithm (Python)

Dry Run Example

Graph Details

Vertices: 4 (0, 1, 2, 3)
Edges:
(0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4)

4 Vertices with lebels 0, 1, 2, 3 and edges with weights as follows: between 0 and 1 = 10, between 0 and 2 = 6, between 0 and 3 = 5, between 1 and 3 = 15, between 2 and 3 = 4.

Step 1: Sort Edges by Weight

- (2, 3, 4)
- (0, 3, 5)
- (0, 2, 6)
- (0, 1, 10)
- (1, 3, 15)

Step 2: Initialize

- parent = [0, 1, 2, 3]

- rank = [0, 0, 0, 0]
- cost = 0

Step 3: Process Each Edge

- **(2, 3, 4)**: Find(2)=2, Find(3)=3 → Different sets → Add to MST
  Union(2, 3) → parent[3] = 2 → cost = 4
- **(0, 3, 5)**: Find(0)=0, Find(3)=Find(2)=2 → Different sets → Add to MST
  Union(0, 2) → parent[2] = 0 → cost = 9
- **(0, 2, 6)**: Find(0)=0, Find(2)=Find(0)=0 → Same set → Ignore
- **(0, 1, 10)**: Find(0)=0, Find(1)=1 → Different sets → Add to MST
  Union(0, 1) → parent[1] = 0 → cost = 19
- **(1, 3, 15)**: Find(1)=Find(0)=0, Find(3)=Find(2)=Find(0)=0 → Same set → Ignore

Final MST

Edges included in MST:

- (2, 3, 4)
- (0, 3, 5)
- (0, 1, 10)

Total cost of MST = **19**

Edge Inclusion Summary

Edge	Included in MST?	Reason
(2, 3, 4)	Yes	Different sets
(0, 3, 5)	Yes	Different sets
(0, 2, 6)	No	Same set (would form cycle)
(0, 1, 10)	Yes	Different sets
(1, 3, 15)	No	Same set (would form cycle)

3.7 C++ Code

```cpp
struct Edge {
    int u, v, w;
    bool operator<(const Edge& e) const { return w < e.w; }
};

int find(int parent[], int x) {
    if (parent[x] != x)
        parent[x] = find(parent, parent[x]);
    return parent[x];
}

void unite(int parent[], int rank[], int x, int y) {
    int xroot = find(parent, x);
    int yroot = find(parent, y);
    if (rank[xroot] < rank[yroot])
        parent[xroot] = yroot;
    else {
        parent[yroot] = xroot;
        if (rank[xroot] == rank[yroot]) rank[xroot]++;
    }
}

int kruskal(int V, vector<Edge>& edges) {
    sort(edges.begin(), edges.end());
    int parent[V], rank[V] = {};
    iota(parent, parent + V, 0);
    int cost = 0;

    for (auto& e : edges) {
        if (find(parent, e.u) != find(parent, e.v)) {
            unite(parent, rank, e.u, e.v);
            cost += e.w;
        }
    }
    return cost;
}
```

Listing 4: Kruskal's Algorithm in C++

3.8 Boundary Conditions and Limitations

- **Graph Type:** Works only on **connected**, **undirected**, and **weighted** graphs.
- **Disconnected Graph:** If the input graph is disconnected, Kruskal's algorithm will compute a **Minimum Spanning Forest**, not a single tree.
- **Sorting Bottleneck:** Requires sorting of edges by weight — incurs $O(E \log E)$ time complexity, which may be slow for very large edge sets.
- **Cycle Detection:** Needs a Disjoint Set Union (DSU) or Union-Find data structure with path compression and union by rank for efficiency.
- **Sparse Graphs:** More efficient than Prim's on sparse graphs, but less efficient than Prim's on dense graphs due to edge sorting.
- **Does Not Handle Negative Cycles:** Kruskal's algorithm does not detect or manage negative weight cycles (though for MCSTs, negative cycles are not typical).
- **No Real-Time Updates:** Kruskal's algorithm is not suitable for dynamic graphs where edges are frequently added/removed, since re-sorting or rebuilding DSU is needed.

3.9 Where Kruskal's Algorithm is Not Applicable

- When the graph is **directed**: Kruskal's algorithm only works on undirected graphs.
- For **disconnected graphs**, it returns a **Minimum Spanning Forest**, not a single spanning tree.
- Not suitable for **dynamic graphs** where edges are added or removed frequently.
- Not designed for computing **shortest paths** between two nodes.
- Less efficient for **dense graphs** due to edge sorting overhead ($O(E \log E)$).

3.10 Correct Use Case for Kruskal's Algorithm

- Graph is **connected**, **undirected**, and **weighted**.
- Especially efficient when the graph is **sparse** (fewer edges).
- You want to **minimize the total edge cost** without creating cycles.
- Graph is **static**, i.e., edges don't change over time.
- You need a **global view of all edges** (edge list is available).

3.11 Why Kruskal's Algorithm is Not for Shortest Path Problems

- **Kruskal's Algorithm** is designed to find a **Minimum Cost Spanning Tree (MCST)**.
- It connects **all nodes** in the graph while minimizing the **total edge weight** and **avoiding cycles**.
- However, it does **not compute shortest paths** between any two specific vertices.

Use the Right Algorithm for the Right Task:

- If you want to connect all cities (nodes) with minimum wiring cost, use **Prim's** or **Kruskal's**.
- If you want to travel from your house to the hospital in the shortest time, use:
  - **Dijkstra's Algorithm** – for graphs with non-negative weights.
  - **BFS** – for unweighted graphs (or equal weights).
  - **A* Search** – if you have heuristics (e.g., map coordinates).

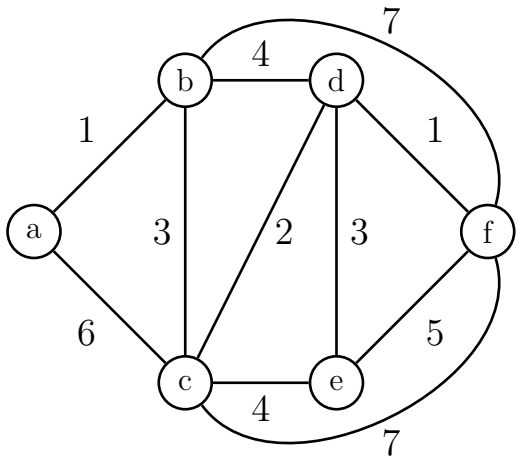
Goal	**Best Algorithm**
Connect all cities with minimum cost	Kruskal / Prim
Go from A to B in least time	Dijkstra / BFS / A*

Table 2: Choosing the Right Graph Algorithm

3.12 GATE CSE 2006

Consider the following graph:

Weighted Graph



Which one of the following cannot be the sequence fo edges added, in that order, to a minimum spanning tree using Kruskal's algorithm?

(a) $(a-b)$, $(d-f)$, $(b-f)$, $(d-c)$, $(d-e)$

(b) $(a-b)$, $(d-f)$, $(d-c)$, $(b-f)$, $(d-e)$

(c) $(d-f)$, $(a-b)$, $(d-c)$, $(b-f)$, $(d-e)$

(d) $(d-f)$, $(a-b)$, $(b-f)$, $(d-e)$, $(d-c)$

4 Dijkstra's Algorithm (Shortest Path, Not MST)

Note: Dijkstra's Algorithm is not used to compute MST. It is used for finding the shortest path from a source node to all other nodes in a graph with non-negative weights.

Steps

- Use a priority queue to pick the node with the least distance.
- Update distances of adjacent vertices if a shorter path is found.

Time and Space Complexity

- Time: $O(E \log V)$
- Space: $O(V)$

4.1 Dijkstra's Algorithm: Logic

Purpose

Dijkstra's algorithm is used to **find the shortest path** from a **source node** to all other nodes in a **weighted graph** with **non-negative edge weights**.

Core Idea (Greedy Approach)

- Start from the source vertex.
- At each step, **select the node with the smallest known distance** from the source (greedy choice).
- **Update distances** of its neighbors if shorter paths are found.
- Repeat until all vertices are processed.

Working Steps

1. Initialize all distances to ∞ except the source (set to 0).
2. Use a min-priority queue (or min-heap) to pick the vertex with the **minimum distance**.
3. For each neighbor v of current node u, if

$$\text{dist}[u] + \text{weight}(u, v) < \text{dist}[v]$$

   then update dist$[v]$.
4. Continue until all vertices are visited.

4.2 Dijkstra's Algorithm Pseudocode

Algorithm 5 Dijkstra's Algorithm

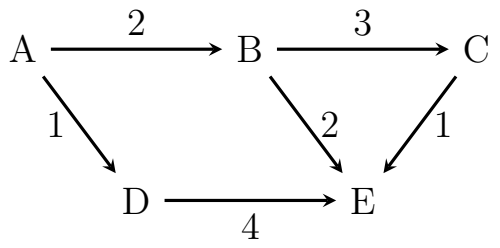
```
1:  procedure DIJKSTRA(G, source)
2:      Initialize distance array dist[v] ← ∞ for all v in G
3:      dist[source] ← 0
4:      Create a min-priority queue Q
5:      Q.insert(source, 0)
6:      while Q is not empty do
7:          u ← Q.extract_min()
8:          for all neighbors v of u do
9:              if dist[u] + weight(u, v) < dist[v] then
10:                 dist[v] ← dist[u] + weight(u, v)
11:                 Q.insert_or_update(v, dist[v])
12:             end if
13:         end for
14:     end while
15:     return dist
16: end procedure
```

4.3 Dijkstra's Algorithm: Worked Example

Sample Graph

Consider the following weighted, undirected graph with 5 nodes:

Directed Weighted Graph



Goal: Find the shortest distances from source node A to all other nodes.

Initialization

- Set distance of A = 0, all others = ∞
- Distance array: `dist = {A:0, B:∞, C:∞, D:∞, E:∞}`
- Min-heap queue: `Q = [(0, A)]`

Step-by-Step Execution

1. **Extract A (0):**
    - Neighbors: B (2), D (1)
    - Update dist[B] = 2, dist[D] = 1
    - Q = [(1, D), (2, B)]
2. **Extract D (1):**
    - Neighbors: A (already visited), E (1+4=5)
    - Update dist[E] = 5
    - Q = [(2, B), (5, E)]
3. **Extract B (2):**
    - Neighbors: A, C (2+3=5), E (2+2=4)

- Update dist[C] = 5, dist[E] = **min(5,4)=4**
- Q = [(4, E), (5, C)]

4. **Extract E (4):**
   - Neighbors: D, B, C (4+1=5) → No change
   - Q = [(5, C)]
5. **Extract C (5):** All neighbors visited. Done.

Final Shortest Distances from A

Node	Distance from A
A	0
B	2
C	5
D	1
E	4

Path Summary

- A → D = 1
- A → B = 2
- A → B → E = 4
- A → B → C = 5

Note: The greedy nature of Dijkstra ensures the shortest path is computed correctly as long as edge weights are non-negative.

Python Code

```python
import heapq

def dijkstra(graph, start):
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    pq = [(0, start)]

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]: continue
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return dist
```

Listing 5: Dijkstra's Algorithm in Python

C++ Code

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> dijkstra(int V, vector<pair<int,int>> adj[]) {
    vector<int> dist(V, INT_MAX);
    dist[0] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>,
       greater<>> pq;
    pq.push({0, 0});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        for (auto [v, w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

Listing 6: Dijkstra's Algorithm in C++

4.4 Boundary Conditions and Limitations

Where Dijkstra's Algorithm Fails or Is Not Applicable:

- **Negative Edge Weights:** Dijkstra's algorithm assumes that once a node is visited with the shortest path, it cannot be improved further — this fails with negative weights.
- **Dynamic or Real-Time Updates:** Dijkstra is not optimal for rapidly changing edge weights (like in live GPS traffic).
- **Directed Cycles with Negative Weights:** It may get stuck or give incorrect results.
- **Memory Usage:** Requires $O(V)$ space for distances and priority queue — inefficient for very large graphs unless optimized.

4.5 Correct Use Cases for Dijkstra's Algorithm:

- Graphs with **non-negative edge weights**.
- Finding the **shortest path from a single source** to all other vertices.
- Works well with **dense graphs** using Min Heap (e.g., via Fibonacci or Binary Heap).
- Suitable when you need **guaranteed shortest distance** (not just any path).

4.6 Where Dijkstra Is Ideal:

- Navigation apps like Google Maps (when all weights are time or distance, and non-negative).
- Packet routing in static, weighted networks.
- Robotics path planning (without heuristic).

Table: Applicability Summary

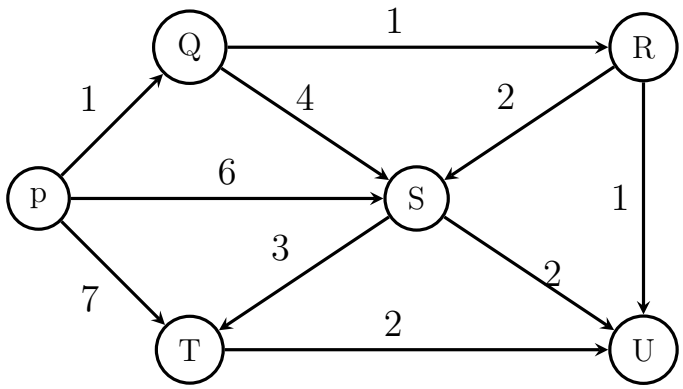
Scenario	Use Dijkstra?
Non-negative weights	Yes
Negative weights (e.g., Bellman-Ford case)	No
Single Source Shortest Path	Yes
All Pairs Shortest Path	Use Dijkstra multiple times or Floyd-Warshall
Real-time edge updates	No (prefer A* or dynamic algorithms)

Table 3: When to Use Dijkstra's Algorithm

4.7 GATE CSE 2004

Suppose we run Dijkstra's single source shortest path algorithm on the following edge-weighted directed graph with vertex P as the source.

Directed Weighted Graph



(a) P, Q, R, S, T, U

(b) P, Q, R, U, S, T

(c) P, Q, R, U, T, S

(d) P, Q, T, R, U, S

https://gateoverflow.in/1041/gate-cse-2004-question-44

4.8 GATE CSE 2005

Let $G(V, e)$ an undirected graph with positive edge weights. Dijkstra's Single Source Shortest Path algorithm can be implemented using the binary heap data structure with time complexity of?

(a) $O\left(|V|^2\right)$
(b) $O\left(|E| + |V| \log |V|\right)$
(c) $O\left(|V| \log |V|\right)$
(d) $O\left((|E| + |V|) \log |V|\right)$

https://gateoverflow.in/1374/gate-cse-2005-question-38

4.9 GATE CSE 2006

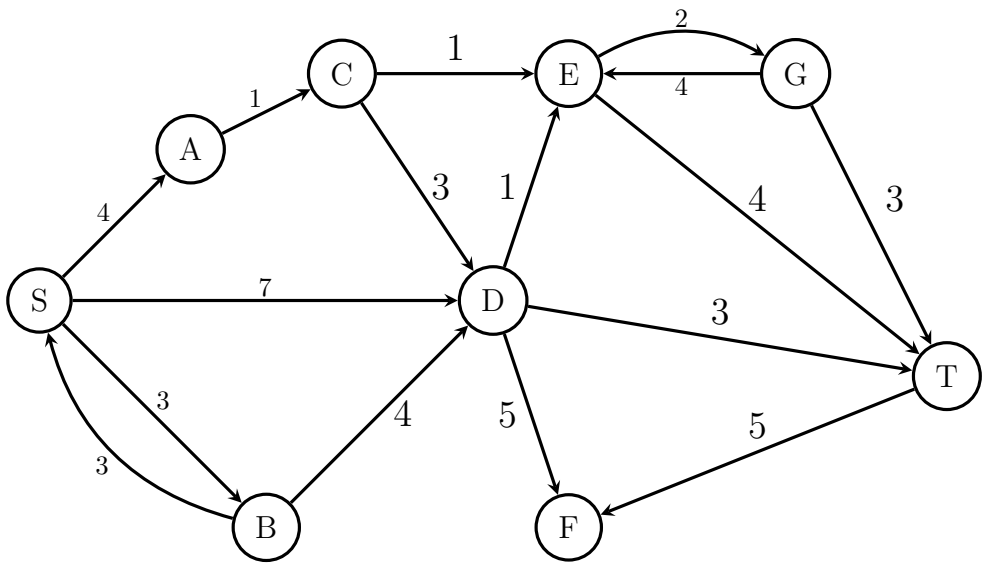
To implement Dijkstra's shortest path algorithm on unweighted graph so that it runs in linear time, then data structure to be used is

(a) Queue
(b) Stack
(c) Heap
(d) B-Tree

4.10 GATE CSE 2012

Consider the directed graph shown in the figure below. There are multiple shortest path between vertices S and T. Which one will be reported by Dijkstra's Shortest Path algorithm? Assume that, in any iteration, the shortest path to a vertex v is updated only when a strictly shorter path to v is discovered.

Directed Weighted Graph



(a) SDT

(b) SVDT

(c) SACDT

(d) SACET

5 Comparison Table

Algorithm	**Purpose**	**Time Complexity**	**Data Structure**
Prim's	Minimum Spanning Tree	$O(E \log V)$	Min Heap
Kruskal's	Minimum Spanning Tree	$O(E \log E)$	DSU + Sort
Dijkstra's	Single Source Shortest Path	$O(E \log V)$	Min Heap

Table 4: Comparison of Graph Algorithms

Algorithm	**Data Structure**	**Time Complexity**	**Remarks**
Prim's	Adjacency Matrix + Array	$O(V^2)$	For dense graphs
	Min Heap + Adjacency List	$O((V+E)\log V)$	Practical and commonly used
	Fibonacci Heap	$O(E + V \log V)$	Best theoretical bound
Kruskal's	Sorting + DSU (Naive)	$O(E \log E)$	Sorting dominates
	DSU (Path Compression + Rank)	$O(E \log V)$	Efficient in practice
Dijkstra's	Array	$O(V^2)$	For dense graphs
	Min Heap / Binary Heap	$O((V+E)\log V)$	Widely used
	Fibonacci Heap	$O(E + V \log V)$	Theoretical interest only

Table 5: Time Complexity Comparison of Graph Algorithms Based on Data Structures

6 Conclusion

Prim's and Kruskal's algorithms[4] are optimal solutions to find a minimum spanning tree. Prim's is suitable for dense graphs, while Kruskal's performs better on sparse graphs. Dijkstra's algorithm, on the other hand, is used to compute the shortest paths and not MSTs.

7 MCST-Based Competitive Programming Problems

1. Connecting Cities With Minimum Cost

Platform: LeetCode
Link: leetcode.com/problems/connecting-cities-with-minimum-cost
Concept: Apply Kruskal's algorithm to find MST over a set of city connections.
Tags: Graph, Union-Find, Kruskal, Greedy

2. Planet Connections

Platform: Codeforces (Round 101 Div. 2)
Link: codeforces.com/problemset/problem/1245/D
Concept: You are given coordinates and wire costs; build MST to minimize total cost.
Tags: Prim's Algorithm, Coordinate Geometry, Greedy

3. Fiber Network

Platform: CSES Problem Set
Link: cses.fi/problemset/task/1675
Concept: Standard MST problem with a very clean interface, ideal for beginners.
Tags: Graph, MST, Kruskal, Sorting

4. New Roads Queries

Platform: SPOJ
Link: spoj.com/problems/NEWROAD
Concept: Dynamic MST — handle queries involving MST edges and reweighting.
Tags: Offline Queries, Kruskal, DSU with rollback

5. City and Flood

Platform: HackerEarth
Link: hackerearth.com/problem/algorithm/city-and-flood-1
Concept: Simple DSU/MST-based component counting.
Tags: Disjoint Set, Flood Fill, MST

6. Dark Roads

Platform: CSES Problem Set
Link: cses.fi/problemset/task/1163
Concept: Given the total cost of all roads, compute the savings from the MST.
Tags: MST, Graph Optimization, Greedy

7. Minimum Spanning Tree

Platform: HackerRank
Link: hackerrank.com/challenges/kruskalmstrsub/problem
Concept: Classical MST implementation using Kruskal's algorithm.
Tags: Graph, Kruskal, Sorting, Union-Find

8. Road Construction

Platform: CSES Problem Set
Link: cses.fi/problemset/task/1676
Concept: Maintain number of connected components and minimum cost.
Tags: Kruskal, DSU, MST, Connected Components

8 Key Research Papers

1. Borůvka's Algorithm (1926)

Author: Otakar Borůvka
Paper: *O jistém problému minimálním (On a certain minimal problem)*
Published in: Práce Moravské Přírodovědecké Společnosti
Link: Digital Library of Czech Academy (DML-CZ)
Why Important: First known algorithm for MST; useful in parallel/distributed computing.

2. Kruskal's Algorithm (1956)

Author: J. B. Kruskal
Paper: *On the shortest spanning subtree of a graph and the traveling salesman problem*
Published in: Proceedings of the American Mathematical Society
Link: JSTOR
Why Important: Introduced the greedy edge-based MST algorithm using Disjoint Sets (DSU).

3. Prim's Algorithm (1957)

Author: R. C. Prim
Paper: *Shortest connection networks and some generalizations*
Published in: Bell System Technical Journal
Link: IEEE Xplore
Why Important: Presents a vertex-based greedy algorithm; efficient for dense graphs.

4. Tarjan's Union-Find Optimization (1975)

Author: R. E. Tarjan
Paper: *Efficiency of a good but not linear set union algorithm*
Published in: Journal of the ACM (JACM)
Link: ACM Digital Library
Why Important: Introduced union-by-rank and path compression used in Kruskal's algorithm.

5. Karger's Randomized MST (1995)

Authors: D. Karger, P. Klein, R. Tarjan
Paper: *A randomized linear-time algorithm to find minimum spanning trees*
Published in: Journal of the ACM
Link: ACM Digital Library
Why Important: Randomized linear-time MST; milestone in theoretical CS.

6. Distributed MST (1983)

Authors: R. Gallager, P. Humblet, P. Spira
Paper: *A distributed algorithm for minimum-weight spanning trees*
Published in: ACM Transactions on Programming Languages and Systems (TOPLAS)
Link: ACM Digital Library
Why Important: Pioneered MST in distributed systems, used in network protocols.

7. Matroid Theory and Greedy MST (1971)

Author: Jack Edmonds
Paper: *Matroids and the greedy algorithm*
Published in: Mathematical Programming
Link: SpringerLink
Why Important: Shows MST as a matroid problem; theoretical foundation for greedy correctness.

References

[1] Otakar Borůvka. O jistém problému minimálním (on a certain minimal problem). *Práce Moravské Přírodovědecké Společnosti*, 1926. https://dml.cz/handle/10338.dmlcz/401613.

[2] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1(1):127–136, 1971.

[3] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):66–77, 1983.

[4] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[5] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[6] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[7] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.