Minimum Cost Spanning Trees

Varun Kumar

July 8, 2025

Contents

Listings

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.

Applications

- Network design (LAN, telecommunication)
- 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 Dynamic Programming 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 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.

2 Prim's Algorithm

Prim's 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

```
Input: Weighted undirected graph G = (V, E)
Output: Minimum Cost of Spanning Tree
function PRIM(Graph G, Start Vertex s)
    Initialize a min-heap Q
    visited ← empty set
    cost ← 0
    Insert (0, s) into Q                          ▷ Start with source node
    while Q is not empty do
        (w, u) ← Extract-Min from Q
        if u ∉ visited then
            Add u to visited
            cost ← cost + w
            for each neighbor (v, weight) of u do
                if v ∉ visited then
                    Insert (weight, v) into Q
                end if
            end for
        end if
    end while
    return cost
end function
```

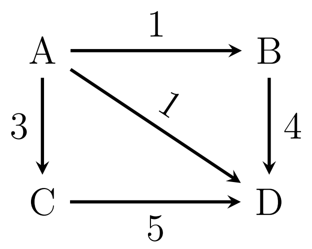
Time and Space Complexity

- 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++

3 Kruskal's Algorithm

Kruskal's 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) 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

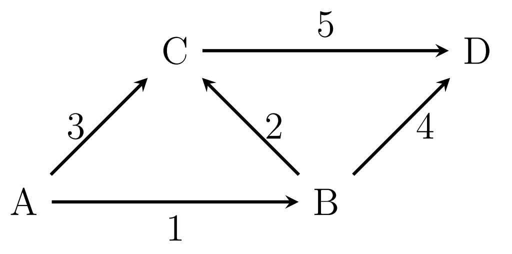
Algorithm 4 Kruskal's Algorithm

```
Input: Graph G = (V, E)
Output: MST set of edges
function Kruskal(V, E)
    Sort edges E in ascending order by weight
    Initialize DSU with each vertex in its own set
    MSTEdges ← empty set
    for each edge (u, v) in sorted E do
        if Find(u) ≠ Find(v) then
            Add (u, v) to MSTEdges
            Union(u, v)
        end if
    end for
    return MSTEdges
end function
function Find(x)
    if parent[x] ≠ x then
        parent[x] ← Find(parent[x])                ▷ Path compression
    end if
    return parent[x]
end function
function Union(x, y)
    xRoot ← Find(x)
    yRoot ← Find(y)
    if rank[xRoot] ¡ rank[yRoot] then
        parent[xRoot] ← yRoot
    else if rank[yRoot] ¡ rank[xRoot] then
        parent[yRoot] ← xRoot
    else
        parent[yRoot] ← xRoot
        rank[xRoot] ← rank[xRoot] + 1
    end if
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 $\Rightarrow$ Same set $\Rightarrow$ Ignore (would form cycle)
- **(B–D, 4)**: Find(B) = A, Find(D) = D $\Rightarrow$ Different $\Rightarrow$ Add to MST $\Rightarrow$ Union(B, D) $\Rightarrow$ Parent[D] = A
- **(C–D, 5)**: Find(C) = A, Find(D) = A $\Rightarrow$ Same set $\Rightarrow$ 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++

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
$$\mathrm{dist}[u] + \mathrm{weight}(u, v) < \mathrm{dist}[v]$$
then update $\mathrm{dist}[v]$.
4. Continue until all vertices are visited.

4.2 Dijkstra's Algorithm Pseudocode

Algorithm 5 Dijkstra's Algorithm

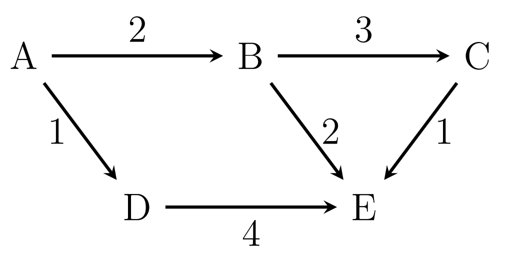
```
procedure DIJKSTRA(G, source)
    Initialize distance array dist[v] ← ∞ for all v in G
    dist[source] ← 0
    Create a min-priority queue Q
    Q.insert(source, 0)
    while Q is not empty do
        u ← Q.extract_min()
        for all neighbors v of u do
            if dist[u] + weight(u, v) < dist[v] then
                dist[v] ← dist[u] + weight(u, v)
                Q.insert_or_update(v, dist[v])
            end if
        end for
    end while
    return dist
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

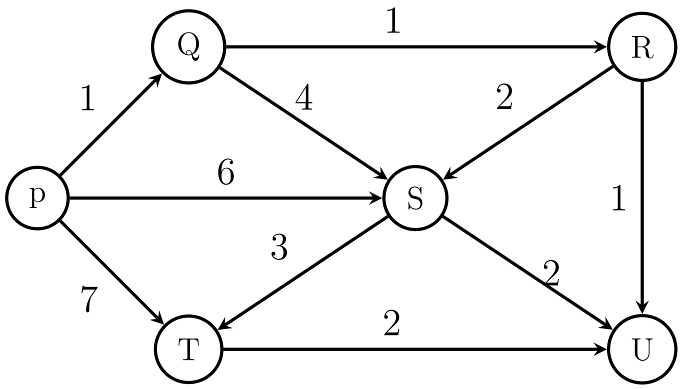
```cpp
#include <bits/stdc++.h>
using namespace std;

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 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

4.5 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?

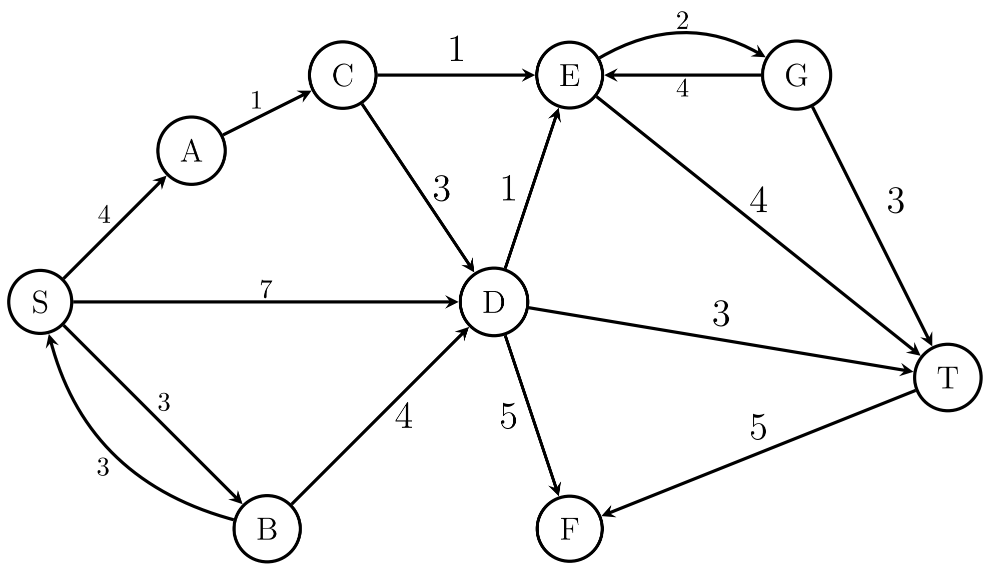
(a) $O\left(|V|^2\right)$
(b) $O\left(|E| + |V| \log |V|\right)$
(c) $O\left(|V| \log |V|\right)$
(d) $O\left(\left(|E| + |V|\right) \log |V|\right)$

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

4.6 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

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

Conclusion

Prim's and Kruskal's algorithms 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.