

Minimum Cost Spanning Trees

Varun Kumar

July 8, 2025

Contents

1	Minimum Cost Spanning Tree (MST)	3
1.1	Minimum Cost Spanning Tree	3
1.2	General Approach (Greedy Strategy)	3
1.3	Dynamic Programming Approach to MCST (Theoretical) .	4
1.4	When to Use Dynamic Programming for Minimum Cost Spanning Tree (MCST)	7
2	Prim's Algorithm	8
2.1	Algorithm Steps	8
2.2	Core Idea	8
2.3	Prim's Algorithm: Step-by-Step Dry Run Example	10
2.4	Python Code	11
2.5	C++ Code	11
3	Kruskal's Algorithm	12
3.1	Algorithm Steps	12
3.2	Core Idea	12
3.3	Pseudocode	13
3.4	Dry Run Example: Kruskal's Algorithm	14
3.5	Python Code	16
3.6	Working of Kruskal's Algorithm (Python)	17
3.7	C++ Code	19
4	Dijkstra's Algorithm (Shortest Path, Not MST)	20
4.1	Dijkstra's Algorithm: Logic	20
4.2	Dijkstra's Algorithm Pseudocode	21
4.3	Dijkstra's Algorithm: Worked Example	22

4.4	GATE CSE 2004	25
4.5	GATE CSE 2005	25
4.6	GATE CSE 2012	26
5	MCST-Based Competitive Programming Problems	28
6	Key Research Papers	30

Listings

1	Prim's Algorithm in Python	11
2	Prim's Algorithm in C++	11
3	Kruskal's Algorithm in Python	16
4	Kruskal's Algorithm in C++	19
5	Dijkstra's Algorithm in Python	24
6	Dijkstra's Algorithm in C++	24

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

```
1: Input: Connected undirected weighted graph  $G = (V, E)$ 
2: Output: Set of edges forming the MCST
3: function MCST( $G$ )
4:   Initialize an empty set MST
5:   Initialize totalCost  $\leftarrow 0$ 
6:   while MST does not have  $V - 1$  edges do
7:     Pick the minimum weight edge  $(u, v)$  that doesn't form a cycle
       in MST
8:     Add  $(u, v)$  to MST
9:     Update totalCost  $\leftarrow$  totalCost + weight of edge
10:  end while
11:  return MST, totalCost
12: 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)

```

1: Input: Graph  $G = (V, E)$ 
2: Let:  $dp[S][u]$  = minimum cost to connect subset  $S \subseteq V$  ending at vertex  $u$ 
3: Initialize all  $dp[S][u] \leftarrow \infty$ 
4: for each vertex  $u \in V$  do
5:      $dp[2^u][u] \leftarrow 0$  ▷ Cost to start at each node
6: end for
7: for each subset  $S \subseteq 2^V$  do
8:     for each vertex  $u \in V$  where  $S$  includes  $u$  do
9:         for each neighbor  $v$  of  $u$  do
10:            if  $v \notin S$  then
11:                 $newS \leftarrow S \cup \{v\}$ 
12:                 $dp[newS][v] \leftarrow \min(dp[newS][v], dp[S][u] + \text{weight}(u, v))$ 
13:            end if
14:        end for
15:    end for
16: end for
17: 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: $\text{dp}[S][u]$ represents the minimum cost to connect all vertices in subset $S \subseteq V$, ending at vertex u .
- Initially, set all $\text{dp}[S][u]$ to ∞ (unknown or unreachable).
- For each vertex u , set $\text{dp}[\{u\}][u] = 0$, 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 $\text{newS} = S \cup \{v\}$, which means $S \cup \{v\}$.
 - Update the DP value for $\text{dp}[\text{newS}][v]$ as the minimum of its current value or the cost of extending $\text{dp}[S][u]$ by the edge (u, v) .
- After all subsets have been processed, the final answer is the minimum of $\text{dp}[2^V - 1][u]$ 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[2] 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

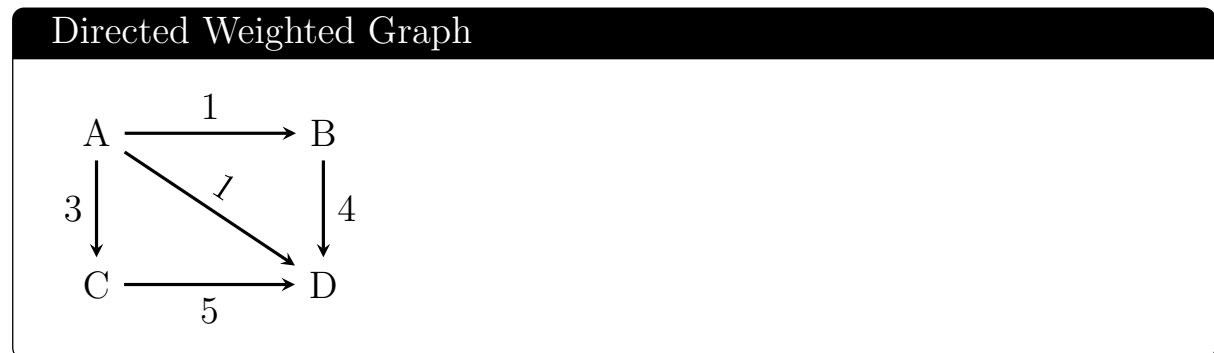
```
1: Input: Weighted undirected graph  $G = (V, E)$ 
2: Output: Minimum Cost of Spanning Tree
3: function PRIM(Graph  $G$ , Start Vertex  $s$ )
4:   Initialize a min-heap  $Q$ 
5:    $visited \leftarrow$  empty set
6:    $cost \leftarrow 0$ 
7:   Insert  $(0, s)$  into  $Q$  ▷ Start with source node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



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

```
1 import heapq
2
3 def prim(graph, start):
4     visited = set()
5     min_heap = [(0, start)]
6     mst_cost = 0
7
8     while min_heap:
9         weight, u = heapq.heappop(min_heap)
10        if u not in visited:
11            visited.add(u)
12            mst_cost += weight
13            for v, w in graph[u]:
14                if v not in visited:
15                    heapq.heappush(min_heap, (w, v))
16
17    return mst_cost
```

Listing 1: Prim's Algorithm in Python

2.5 C++ Code

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int prim(vector<vector<pair<int,int>>> &graph, int V) {
5     vector<bool> visited(V, false);
6     priority_queue<pair<int,int>, vector<pair<int,int>>,
7         greater<>> pq;
8     pq.push({0, 0});
9     int cost = 0;
10
11     while (!pq.empty()) {
12         auto [w, u] = pq.top(); pq.pop();
13         if (!visited[u]) {
14             visited[u] = true;
15             cost += w;
16             for (auto [v, wt] : graph[u])
17                 if (!visited[v])
18                     pq.push({wt, v});
19         }
20     }
21     return cost;
22 }
```

Listing 2: Prim's Algorithm in C++

3 Kruskal's Algorithm

Kruskal's[1] 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.

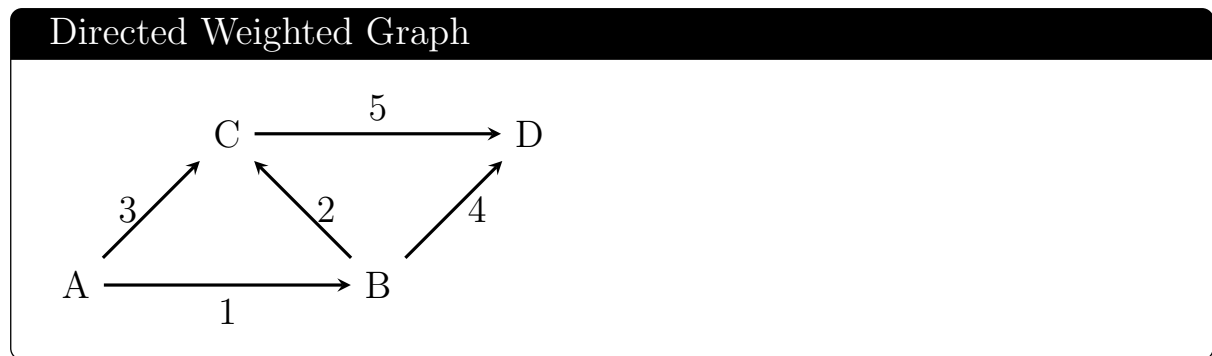
3.3 Pseudocode

Algorithm 4 Kruskal's Algorithm

```
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  $\leftarrow$  empty set
7:   for each edge  $(u, v)$  in sorted  $E$  do
8:     if FIND( $u$ )  $\neq$  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[ $x$ ]  $\neq x$  then
17:     parent[ $x$ ]  $\leftarrow$  FIND(parent[ $x$ ])            $\triangleright$  Path compression
18:   end if
19:   return parent[ $x$ ]
20: end function
21: function UNION( $x, y$ )
22:   xRoot  $\leftarrow$  FIND( $x$ )
23:   yRoot  $\leftarrow$  FIND( $y$ )
24:   if rank[xRoot]  $\leq$  rank[yRoot] then
25:     parent[xRoot]  $\leftarrow$  yRoot
26:   else if rank[yRoot]  $\leq$  rank[xRoot] then
27:     parent[yRoot]  $\leftarrow$  xRoot
28:   else
29:     parent[yRoot]  $\leftarrow$  xRoot
30:     rank[xRoot]  $\leftarrow$  rank[xRoot] + 1
31:   end if
32: end function
```

3.4 Dry Run Example: Kruskal's Algorithm

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

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

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

```
1 def find(parent, x):
2     if parent[x] != x:
3         parent[x] = find(parent, parent[x]) # Path
4         compression
5     return parent[x]
6
7 def union(parent, rank, x, y):
8     xroot = find(parent, x)
9     yroot = find(parent, y)
10    if xroot != yroot:
11        if rank[xroot] < rank[yroot]:
12            parent[xroot] = yroot
13        else:
14            parent[yroot] = xroot
15            if rank[xroot] == rank[yroot]:
16                rank[xroot] += 1
17
18 def kruskal(V, edges):
19     edges.sort(key=lambda x: x[2]) # Sort by weight
20     parent = list(range(V))
21     rank = [0] * V
22     cost = 0
23
24     for u, v, w in edges:
25         if find(parent, u) != find(parent, v):
26             union(parent, rank, u, v)
27             cost += w
28
29     return cost
30
31 # =====
32 # User Input Section
33 # =====
34
35 V, E = map(int, input("Enter number of vertices and edges: ").split())
36 edges = []
37
38 print("Enter edges in format: u v w (0-indexed)")
39 for _ in range(E):
40     u, v, w = map(int, input().split())
41     edges.append((u, v, w))
42
43 # Run Kruskal's Algorithm
44 mst_cost = kruskal(V, edges)
45 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 labels 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]

- $\text{rank} = [0, 0, 0, 0]$
- $\text{cost} = 0$

Step 3: Process Each Edge

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

```
1 struct Edge {
2     int u, v, w;
3     bool operator<(const Edge& e) const { return w < e.w; }
4 };
5
6 int find(int parent[], int x) {
7     if (parent[x] != x)
8         parent[x] = find(parent, parent[x]);
9     return parent[x];
10 }
11
12 void unite(int parent[], int rank[], int x, int y) {
13     int xroot = find(parent, x);
14     int yroot = find(parent, y);
15     if (rank[xroot] < rank[yroot])
16         parent[xroot] = yroot;
17     else {
18         parent[yroot] = xroot;
19         if (rank[xroot] == rank[yroot]) rank[xroot]++;
20     }
21 }
22
23 int kruskal(int V, vector<Edge>& edges) {
24     sort(edges.begin(), edges.end());
25     int parent[V], rank[V] = {};
26     iota(parent, parent + V, 0);
27     int cost = 0;
28
29     for (auto& e : edges) {
30         if (find(parent, e.u) != find(parent, e.v)) {
31             unite(parent, rank, e.u, e.v);
32             cost += e.w;
33         }
34     }
35     return cost;
36 }
```

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

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

then update $\text{dist}[v]$.

4. Continue until all vertices are visited.

4.2 Dijkstra's Algorithm Pseudocode

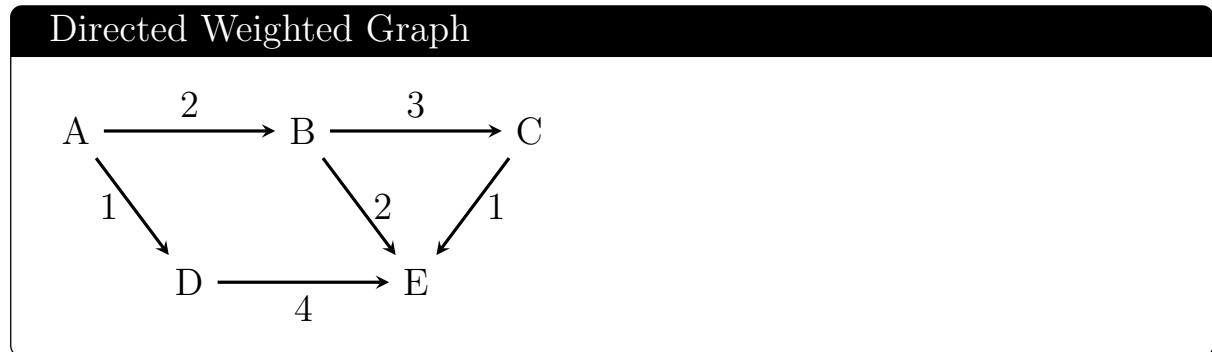
Algorithm 5 Dijkstra's Algorithm

```
1: procedure DIJKSTRA( $G, source$ )
2:   Initialize distance array  $dist[v] \leftarrow \infty$  for all  $v$  in  $G$ 
3:    $dist[source] \leftarrow 0$ 
4:   Create a min-priority queue  $Q$ 
5:    $Q.insert(source, 0)$ 
6:   while  $Q$  is not empty do
7:      $u \leftarrow Q.extract\_min()$ 
8:     for all neighbors  $v$  of  $u$  do
9:       if  $dist[u] + weight(u, v) < dist[v]$  then
10:         $dist[v] \leftarrow dist[u] + weight(u, v)$ 
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:



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 $\text{dist}[C] = 5$, $\text{dist}[E] = \min(5,4)=4$
- $Q = [(4, E), (5, C)]$

4. **Extract E (4):**

- Neighbors: D, B, C ($4+1=5$) \rightarrow 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

- $A \rightarrow D = 1$
- $A \rightarrow B = 2$
- $A \rightarrow B \rightarrow E = 4$
- $A \rightarrow B \rightarrow C = 5$

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

Python Code

```
1 import heapq
2
3 def dijkstra(graph, start):
4     dist = {node: float('inf') for node in graph}
5     dist[start] = 0
6     pq = [(0, start)]
7
8     while pq:
9         d, u = heapq.heappop(pq)
10        if d > dist[u]: continue
11        for v, w in graph[u]:
12            if dist[u] + w < dist[v]:
13                dist[v] = dist[u] + w
14                heapq.heappush(pq, (dist[v], v))
15    return dist
```

Listing 5: Dijkstra's Algorithm in Python

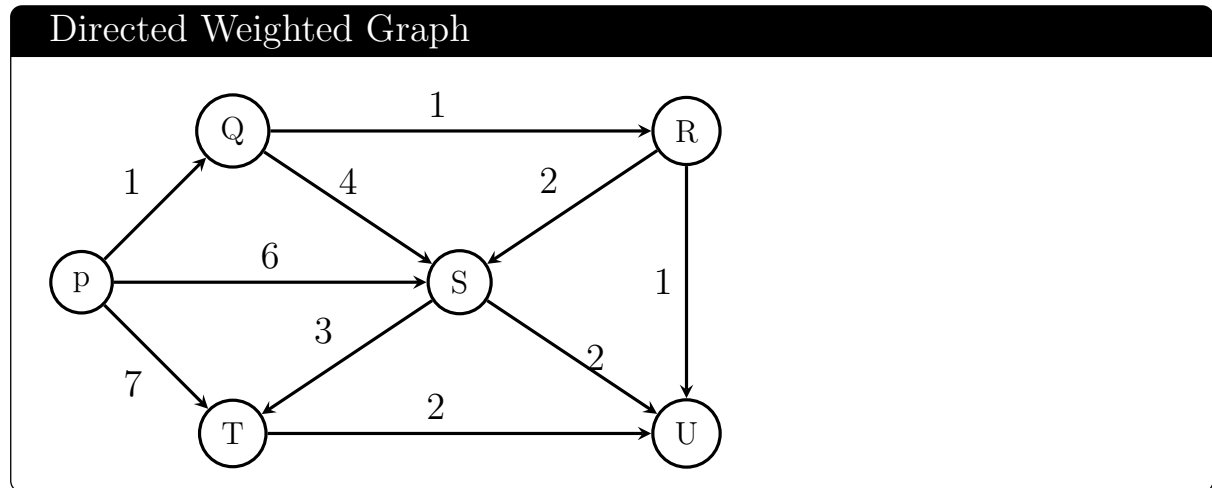
C++ Code

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 vector<int> dijkstra(int V, vector<pair<int,int>> adj[]) {
5     vector<int> dist(V, INT_MAX);
6     dist[0] = 0;
7     priority_queue<pair<int,int>, vector<pair<int,int>>,
8         greater<>> pq;
9     pq.push({0, 0});
10
11     while (!pq.empty()) {
12         auto [d, u] = pq.top(); pq.pop();
13         for (auto [v, w] : adj[u]) {
14             if (dist[u] + w < dist[v]) {
15                 dist[v] = dist[u] + w;
16                 pq.push({dist[v], v});
17             }
18         }
19     }
20     return dist;
21 }
```

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.



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

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

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

4.5 GATE CSE 2005

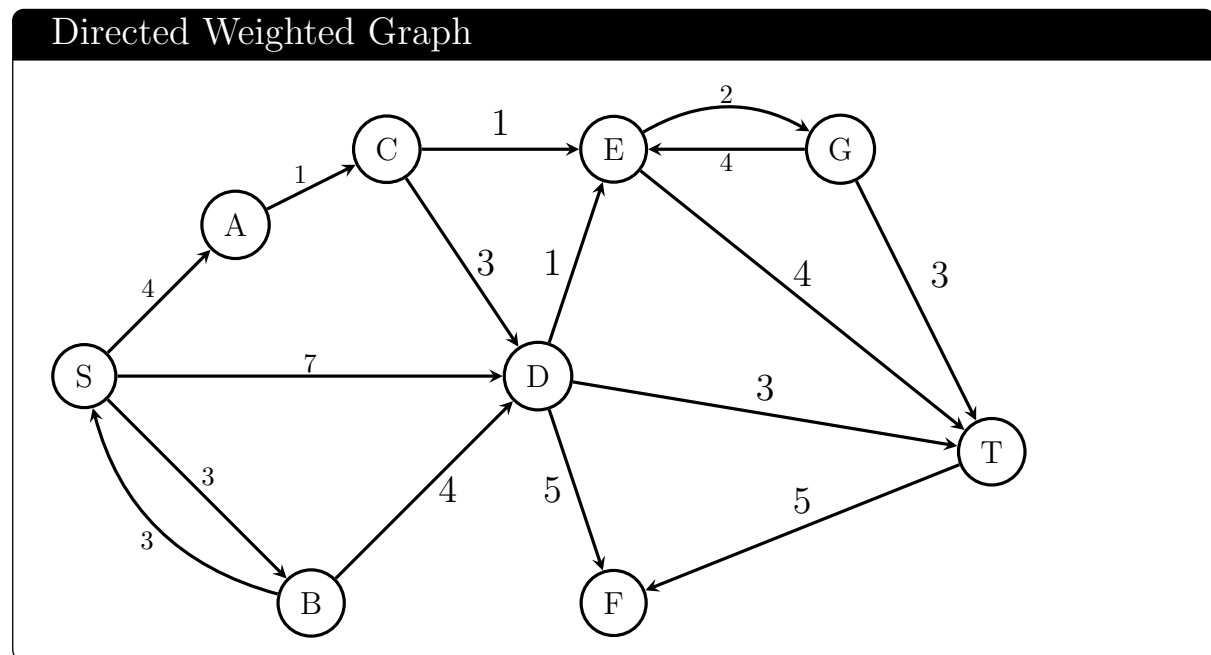
Let $G(V, e)$ an undirected graph with positive edge weights. Dijkstra's Single Source Shortest Path algorithm can be implemented using the binary heap data structure with time complexity of?

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

<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.



- (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.

5 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

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