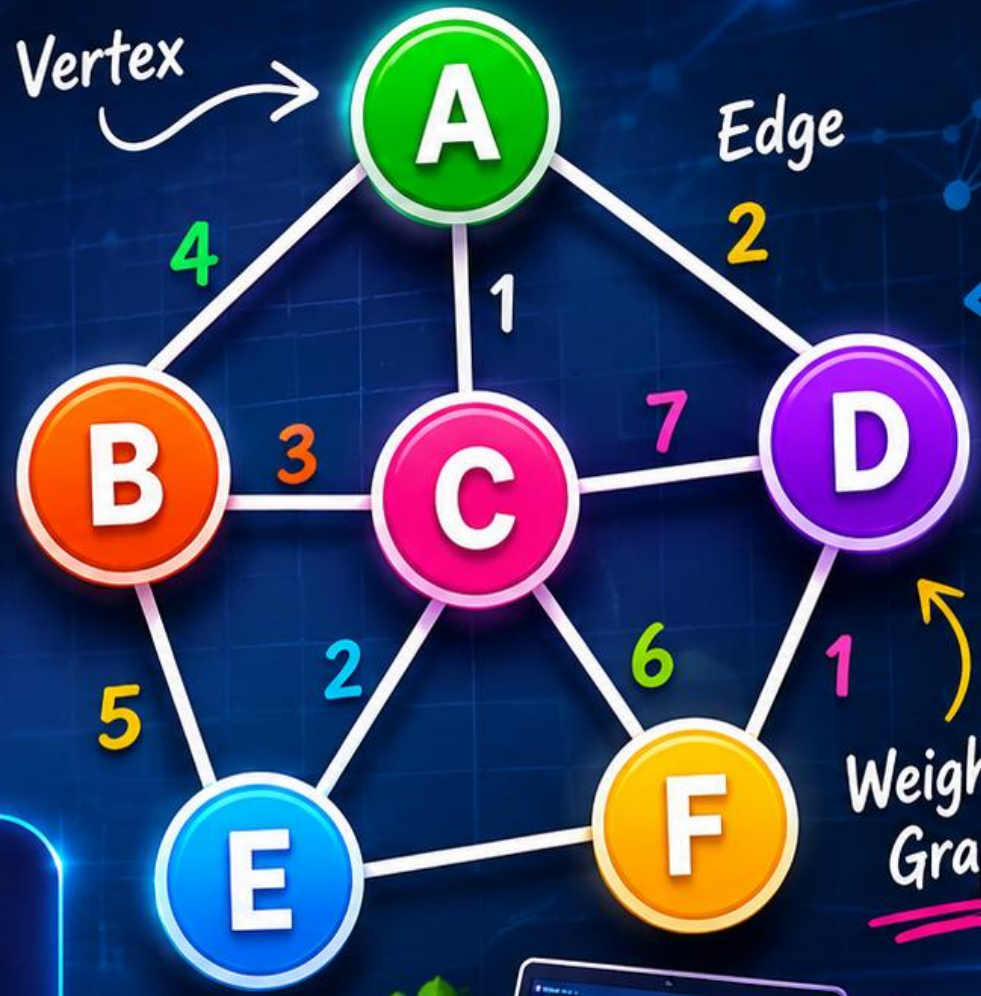
DSA

GRAPH

Explained with Examples!



- Types of Graph
- BFS & DFS Traversal
- Adjacency Matrix & List
- Real World Applications



Data Structures & Algorithms

Introduction to Graph

- A Graph is a **non-linear data structure** consisting of:
  - **Vertices (Nodes)**
  - **Edges (Connections)**
- Graphs are used to represent relationships between objects.
- **Mathematical Definition**
- A graph is represented as:
  - $G = (V, E)$
- Where:
  - **G** = Graph
  - **V** = Set of vertices
  - **E** = Set of edges

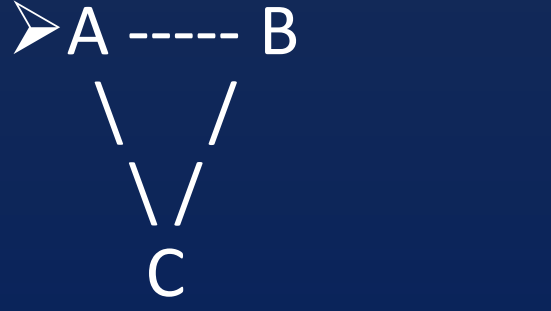
Real-Life Examples of Graph

- Social media networks
- Google Maps
- Computer networks
- Flight routes
- Web page links
- Recommendation systems
- **Example:** Facebook User Graph

```
A ---- B
|      |
C ---- D
```

- Each user is a vertex and friendship is an edge

Components of Graph

- **1. Vertex (Node)**
  - Basic element of graph.
- **2. Edge**
  - Connection between two vertices.
- **Example**
- 

- Vertices: A, B, C
- Edges: AB, BC, AC

Graph Terminologies

Term	Description
Vertex	Node in graph
Edge	Connection between vertices
Degree	Number of connected edges
Adjacent Vertex	Connected vertices
Path	Sequence of connected vertices
Cycle	Closed path
Connected Graph	All vertices connected
Weight	Cost/value on edge

Degree of Vertex

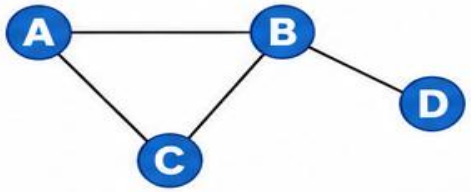
- The number of edges connected to a vertex is called its degree.
- **Formula**
- $Degree(v) = Number\ of\ edges\ connected\ to\ v$
- **Example**
- A ----- B
  \ /
  \/
  C
- Degree of:
- A = 2
- B = 2
- C = 2

Types of Graphs

- Undirected Graph
- Directed Graph
- Weighted Graph
- Unweighted Graph
- Cyclic Graph
- Acyclic Graph
- Connected Graph
- Disconnected Graph
- Complete Graph

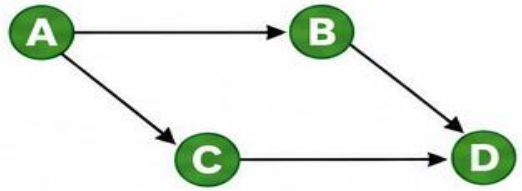
TYPES OF GRAPHS

1. UNDIRECTED GRAPH



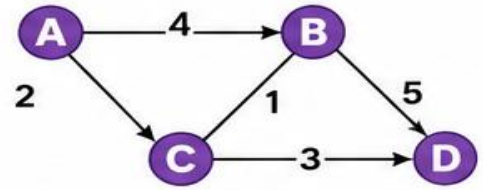
A graph in which edges have no direction.
(Edges are bidirectional)

2. DIRECTED GRAPH



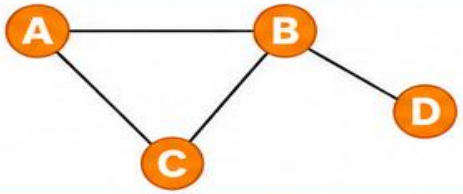
A graph in which edges have direction.
(Edges are unidirectional)

3. WEIGHTED GRAPH



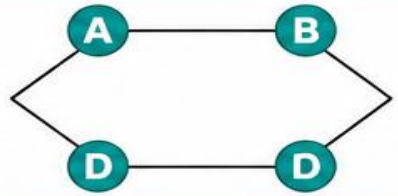
A graph in which edges have weights
or costs associated with them.

4. UNWEIGHTED GRAPH



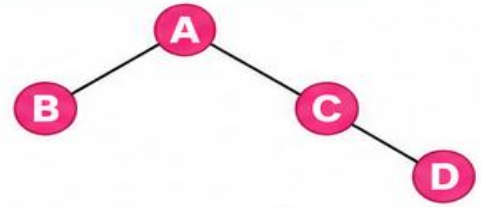
A graph in which edges do not have
any weights or costs.

5. CYCLIC GRAPH



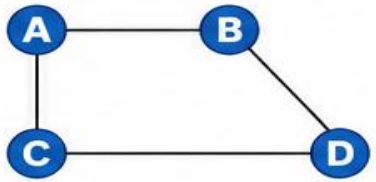
A graph that contains at least one cycle.
(A path that starts and ends at the same node)

6. ACYCLIC GRAPH



A graph that does not contain any cycle.

7. CONNECTED GRAPH



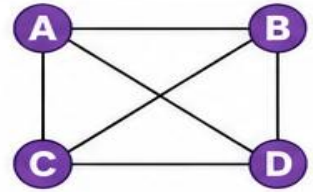
A graph in which there is a path between
every pair of vertices.

8. DISCONNECTED GRAPH



A graph in which at least one pair of vertices
has no path between them.

9. COMPLETE GRAPH



A simple graph in which every pair of
distinct vertices is connected by an edge.

Note: A simple graph has no self-loops and no multiple edges between the same pair of vertices.

Graph Representation

- **Methods of Graph Representation**
- **1. Adjacency Matrix**
- **2. Adjacency List**

Adjacency Matrix

- A 2D array used to represent graph connections.

- **Example**

	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

Adjacency List

- Stores connected vertices using linked lists.
-

- **Example**
- A → B → C
  B → A → C
  C → A → B

Graph Traversal

- Traversal means visiting all graph vertices.
-

- **Types**
- Breadth First Search (BFS)
- Depth First Search (DFS)

Breadth First Search (BFS)

- **BFS Algorithm**
  - Visits vertices level by level.
- Uses:
  - Queue
  - FIFO principle
- **Steps**
  - Start from source node
  - Visit all neighbors
  - Add neighbors into queue
  - Repeat until queue becomes empty

- **Example**
- 

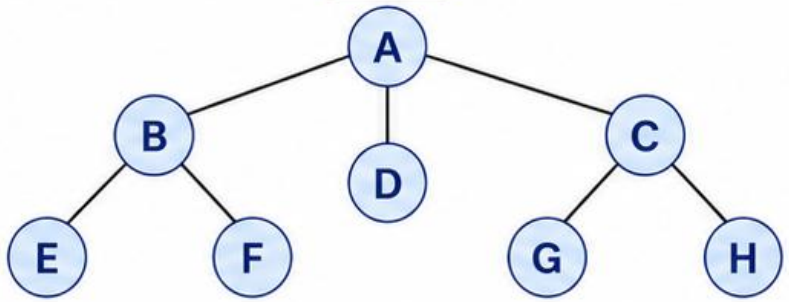
- **BFS Traversal**
- A → B → C → D → E

BFS Algorithm

1. Create queue
2. Insert starting vertex
3. Mark visited
4. Remove vertex from queue
5. Visit adjacent vertices
6. Repeat

BFS (Breadth-First Search) Example

Graph

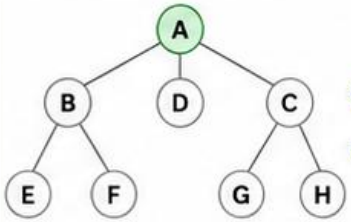


BFS visits nodes level by level from the start node, exploring all neighbors before moving to the next level.

Start Node: A

Goal: Visit all nodes using BFS

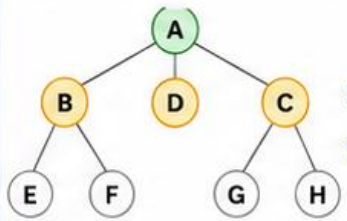
Step 0



Queue: [A]

Visited Order: [ ]

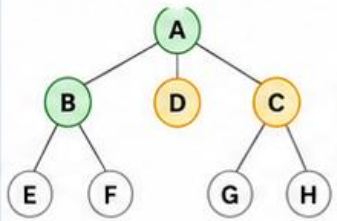
Step 1



Queue: [B, D, C]

Visited Order: [A]

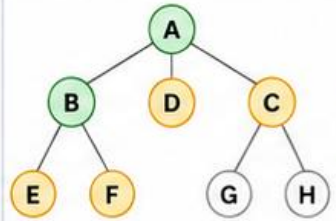
Step 2



Queue: [D, C, E, F]

Visited Order: [A, B]

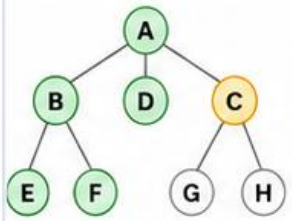
Step 3



Queue: [C, E, F]

Visited Order: [A, B, D]

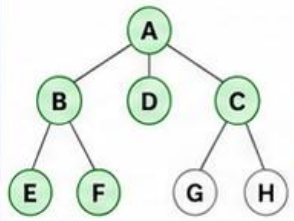
Step 4



Queue: [E, F, G, H]

Visited Order: [A, B, D, C]

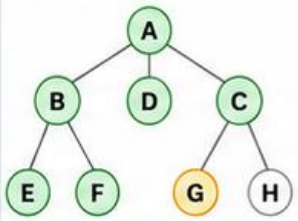
Step 5



Queue: [F, G, H]

Visited Order: [A, B, D, C, E]

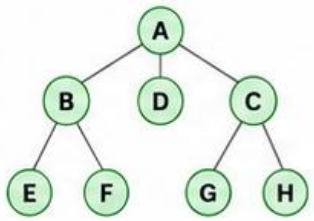
Step 6



Queue: [G, H]

Visited Order: [A, B, D, C, E, F]

Step 7



Queue: [ ]

Visited Order: [A, B, D, C, E, F, G, H]

BFS Traversal Order: A → B → D → C → E → F → G → H

Depth First Search (DFS)

- Visits vertices deeply before backtracking.
- Uses:
  - Stack
  - Recursion

- **Steps**
  - Visit node
  - Go deeper
  - Backtrack when needed

DFS Example

- Example:



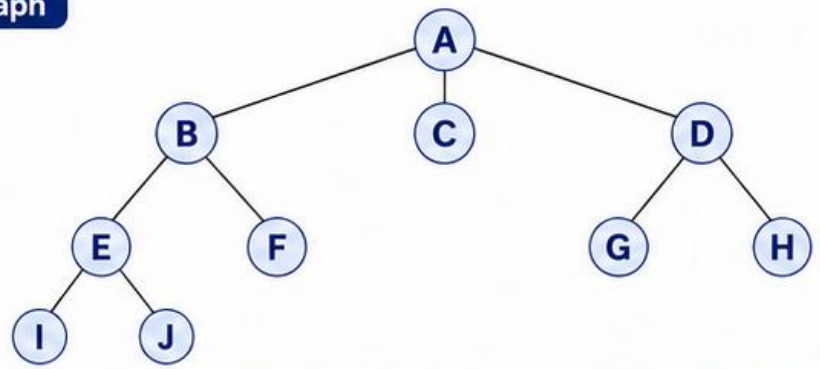
- **DFS Traversal**
- A → B → D → E →

DFS Algorith

1. Visit node
2. Mark visited
3. Visit unvisited adjacent node
4. Repeat recursively

DFS (Depth-First Search) Example

Graph

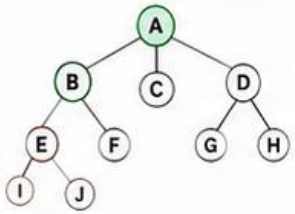


DFS explores as far as possible along each branch before backtracking.
It uses a stack (or recursion) to go deep first.

Start Node: A

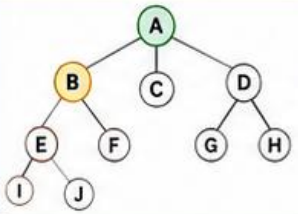
Goal: Visit all nodes using DFS

Step 0



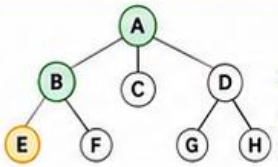
Stack: [A]
Visited Order: [ ]

Step 1



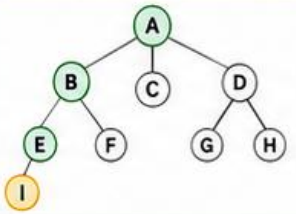
Stack: [B]
Visited Order: [A]

Step 2



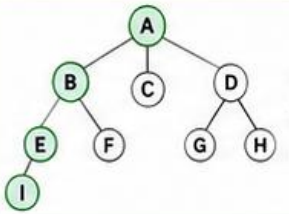
Stack: [E]
Visited Order: [A, B]

Step 3



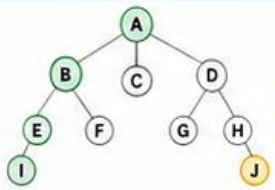
Stack: [I]
Visited Order: [A, B, E]

Step 4



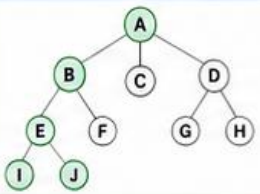
Stack: [ ]
Visited Order: [A, B, E, I]

Step 5



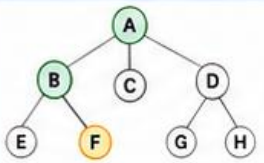
Stack: [J]
Visited Order: [A, B, E, I]

Step 6



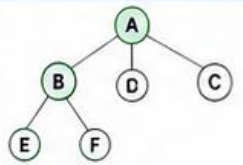
Stack: [ ]
Visited Order: [A, B, E, I, J]

Step 7



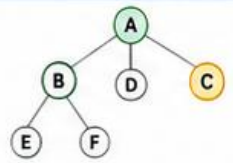
Stack: [F]
Visited Order: [A, B, E, I, J]

Step 8



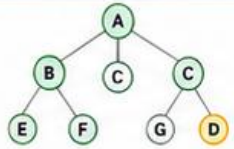
Stack: [ ]
Visited Order: [A, B, E, I, J, F]

Step 9



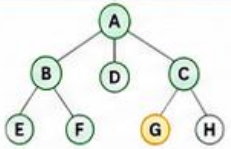
Stack: [C]
Visited Order: [A, B, E, I, J, F]

Step 10



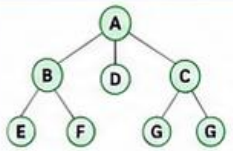
Stack: [D]
Visited Order: [A, B, E, I, J, F, C]

Step 11



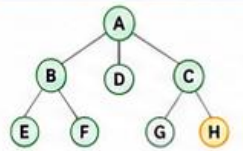
Stack: [G]
Visited Order: [A, B, E, I, J, F, C, D]

Step 12



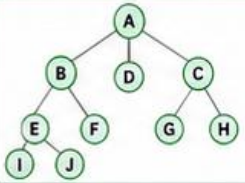
Stack: [ ]
Visited Order: [A, B, E, I, J, F, C, D, G]

Step 13



Stack: [H]
Visited Order: [A, B, E, I, J, F, C, D, G]

Step 14 (Done)



Stack: [ ]
Visited Order: [A, B, E, I, J, F, C, D, G, H]

DFS Traversal Order: A → B → E → I → J → F → C → D → G → H

Applications of Graph

- **Computer Science**
  - Computer networks
  - Compiler design
  - Routing algorithms
- **Real World**
  - Google Maps
  - Facebook friends
  - Airline systems
  - Recommendation engines

Minimum Spanning Tree

- A subset of graph edges connecting all vertices with minimum cost.
-

- **Formula**
- $MST = Minimum\ total\ edge\ weight\ connecting\ all\ vertices$

Properties of a Spanning Tree

- Contains all vertices.
- Has exactly **V – 1 edges**, where **V** is the number of vertices.
- Contains no cycles.
- There may be multiple spanning trees for a graph.

Algorithms to Find MST

- Two popular algorithms:
- **1. Prim's Algorithm**
- **2. Kruskal's Algorithm**

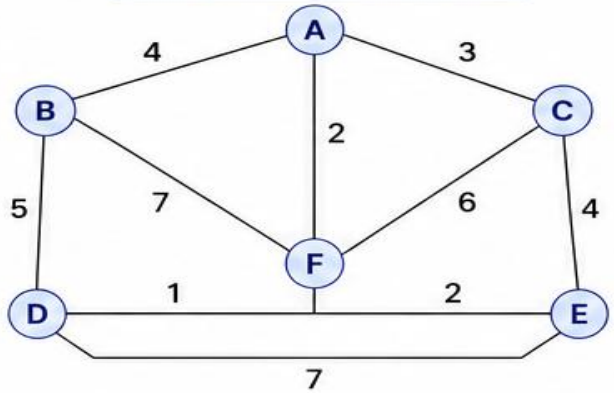
Prim's Algorithm

- Start from any vertex and repeatedly add the minimum-weight edge that connects a new vertex.
- **Steps**
  - Select a starting vertex.
  - Find the minimum edge connected to the tree.
  - Add the new vertex.
  - Repeat until all vertices are included.
- Time Complexity

Implementation	**Complexity**
Adjacency Matrix	$O(V^2)$
Priority Queue	O(E log V)

Form Minimum Spanning Tree (MST) from a Graph using Prim's Algorithm

Original Connected Graph



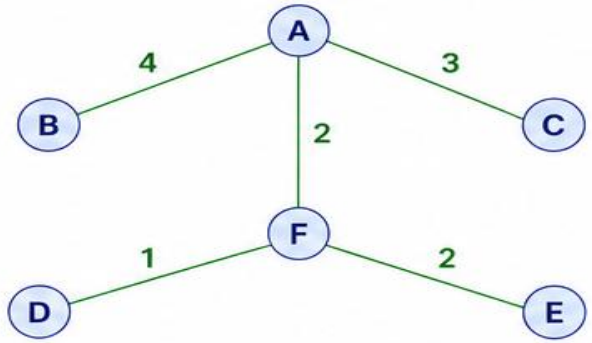
Edge Weights (for reference)

Edge	Weight	Edge	Weight
D – F	1	C – E	4
A – F	2	B – D	5
F – E	2	C – F	6
A – C	3	B – F	7
A – B	4	D – E	7

Prim's Algorithm (Idea)

- Start from any vertex.
- At each step, pick the minimum weight edge that connects the tree to a new vertex.
- Repeat until all vertices are included.

Minimum Spanning Tree (MST)

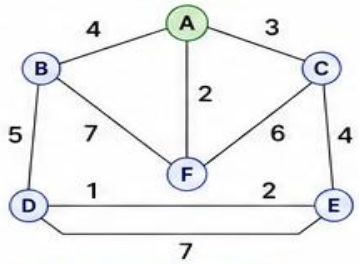


Total Weight of MST = 1 + 2 + 2 + 3 + 4 = 12

Step-by-Step Execution of Prim's Algorithm

1 Start from A

Tree = {A}



Edges from tree to outside:

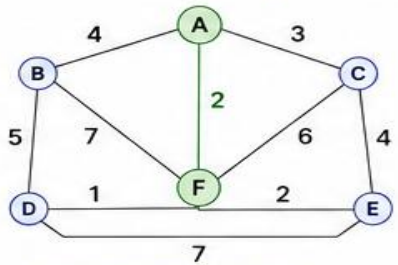
Edge	A–F	A–C	A–B
Weight	2	3	4

Pick A–F (2)

Tree = {A, F}

2 Add vertex F

Tree = {A, F}



Edges from tree to outside:

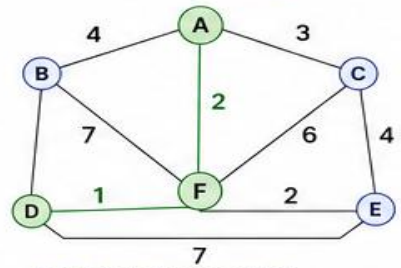
Edge	F–D	F–E	A–C	A–B	F–B	F–C
Weight	1	2	3	4	7	6

Pick F–D (1)

Tree = {A, F, D}

3 Add vertex D

Tree = {A, F, D}



Edges from tree to outside:

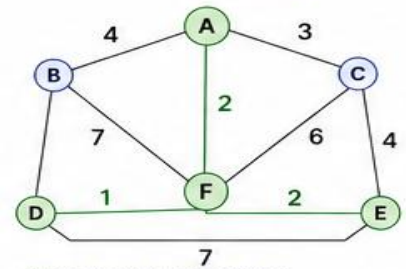
Edge	F–E	A–C	A–B	F–B	F–C	D–E
Weight	2	3	4	7	6	7

Pick F–E (2)

Tree = {A, F, D, E}

4 Add vertex E

Tree = {A, F, D, E}



Edges from tree to outside:

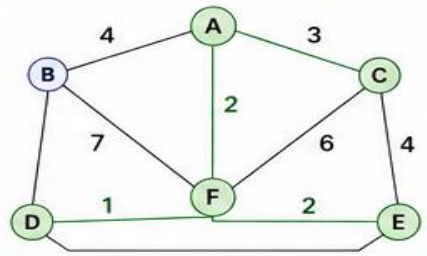
Edge	A–C	A–B	F–B	F–C	E–C	D–E
Weight	3	4	7	6	4	7

Pick A–C (3)

Tree = {A, F, D, E, C}

5 Add vertex C

Tree = {A, F, D, E, C}



Edges from tree to outside:

Edge	A–B	F–B	C–E	D–E
Weight	4	7	4	7

Pick A–B (4)

Tree = {A, F, D, E, C, B} (All vertices included)

MST Completed!

Key Points

- Prim's Algorithm grows the MST one vertex at a time.
- Always choose the minimum weight edge that connects the tree to a new vertex.
- For a graph with V vertices, MST has exactly V – 1 edges.

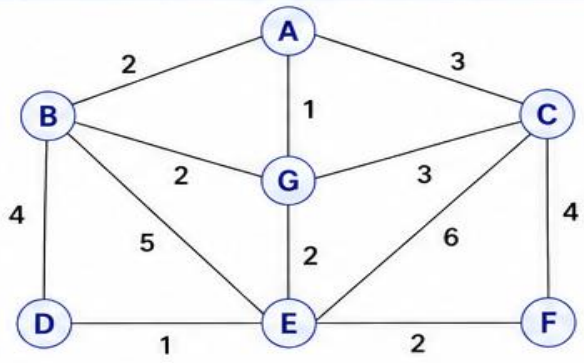
Edges in original graph

Edges in MST

Final MST Weight = 12

Form Minimum Spanning Tree (MST) from a Graph using Prim's Algorithm (Correct)

Original Connected Graph



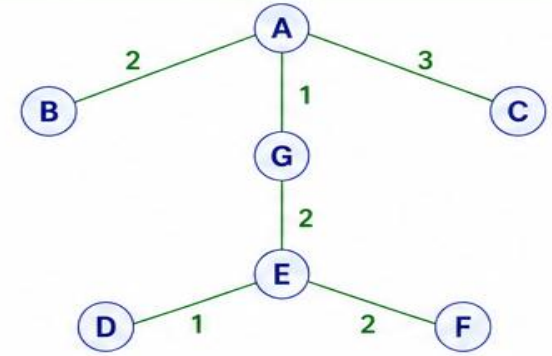
Edge Weights (for reference)

Edge	Weight	Edge	Weight
A – G	1	A – C	3
D – E	1	G – C	3
A – B	2	B – D	4
B – G	2	C – F	4
E – G	2	B – E	5
E – F	2	G – F	6

Prim's Algorithm (Idea)

- Start from any vertex.
- At each step, pick the minimum weight edge that connects the tree to a new vertex.
- Do not form a cycle.
- Repeat until all vertices are included.

Minimum Spanning Tree (MST)

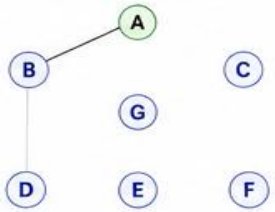


Total Weight of MST = 1 + 2 + 3 + 2 + 1 + 2 = 11

Step-by-Step Execution of Prim's Algorithm (Start from A)

1 Start from A

Tree = {A}



Edges from tree to outside:

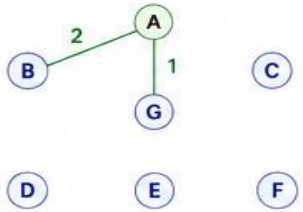
Edge	A–G	A–B	A–C
Weight	1	2	3

Pick A–G (1)

Tree = {A, G}

2 Add vertex B

Tree = {A, G, B}



Edges from tree to outside:

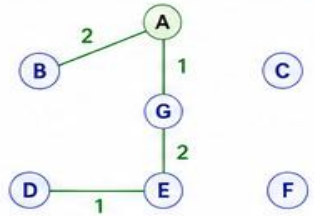
Edge	G–E	G–C	B–D	A–C
Weight	2	3	4	3

Pick G–E (2)

Tree = {A, G, B, E}

3 Add vertex D

Tree = {A, G, B, E, D}



Edges from tree to outside:

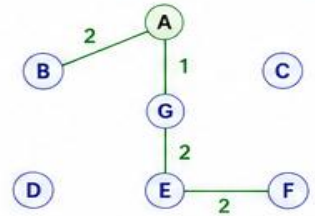
Edge	E–F	G–C	A–C	B–D
Weight	2	3	3	4

Pick E–D (1)

Tree = {A, G, B, E, D}

4 Add vertex F

Tree = {A, G, B, E, D, F}



Edges from tree to outside:

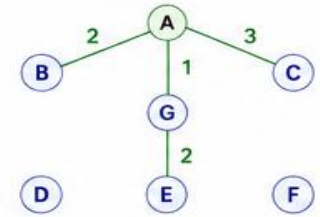
Edge	G–C	A–C	B–D
Weight	3	3	4

Pick E–F (2)

Tree = {A, G, B, E, D, F}

5 Add vertex C

Tree = {A, G, B, E, D, F, C}



Edges from tree to outside:

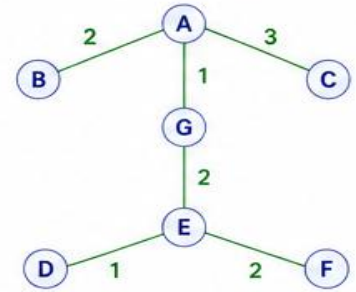
Edge	B–D	G–C
Weight	4	3

Pick A–C (3)

Tree = {A, G, B, E, D, F, C}

6 All vertices included

MST Completed



Total Weight of MST = 11

Key Points

- A spanning tree connects all vertices with exactly V – 1 edges and no cycles.
- Prim's Algorithm always picks the minimum weight edge that connects the tree to a new vertex.
- The final MST here has 7 vertices and 6 edges (V – 1 = 6).

Edges in original graph

Edges in MST

Final MST Weight = 11

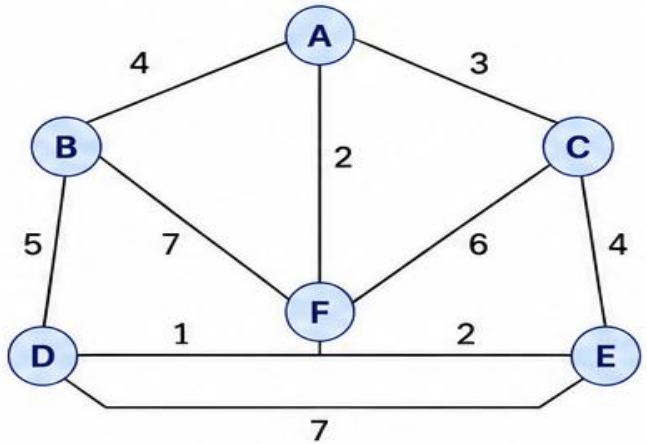
Kruskal's Algorithm

- Sort all edges by weight and repeatedly select the smallest edge that does not form a cycle.
- **Steps**
  - Sort edges in ascending order.
  - Select smallest edge.
  - Check cycle formation.
  - Add if no cycle.
  - Repeat until MST has V−1 edges.
- **Time Complexity**
  - O(E log E)

Form Minimum Spanning Tree (MST) from a Graph

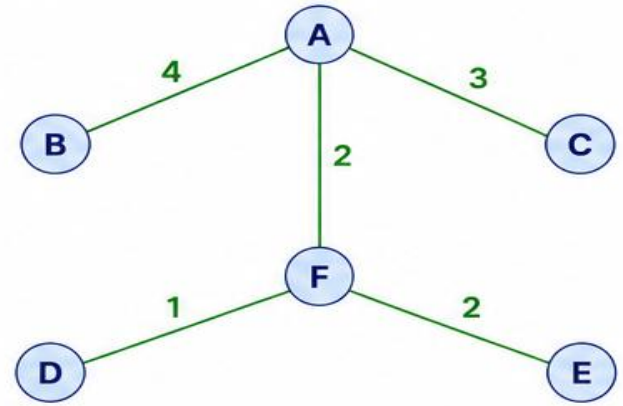
Using Kruskal's Algorithm

Original Connected Graph



Edges (sorted by weight)	
Edge	Weight
D – F	1
A – F	2
F – E	2
A – C	3
A – B	4
C – E	4
B – D	5
C – F	6
B – F	7
D – E	7

Minimum Spanning Tree (MST)

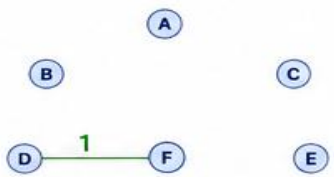


Total Weight of MST = 1 + 2 + 2 + 3 + 4 = 12

Steps (Kruskal's Algorithm)

Step 1

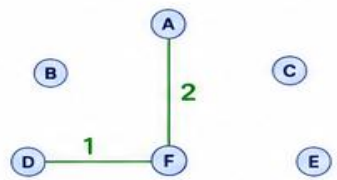
Pick edge D–F (1)
Add to MST



MST Edges: {D–F}
Total Weight: 1

Step 2

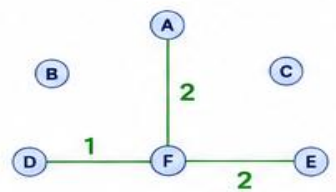
Pick edge A–F (2)
Add to MST



MST Edges: {D–F, A–F}
Total Weight: 3

Step 3

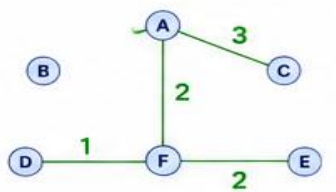
Pick edge F–E (2)
Add to MST



MST Edges: {D–F, A–F, F–E}
Total Weight: 5

Step 4

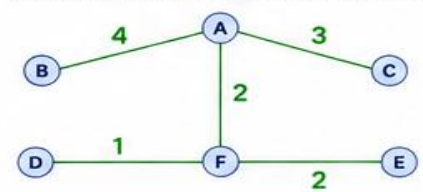
Pick edge A–C (3)
Add to MST



MST Edges: {D–F, A–F, F–E, A–C}
Total Weight: 8

Step 5

Pick edge A–B (4)
Add to MST
(now we have 5 edges for 6 nodes)



MST Edges: {D–F, A–F, F–E, A–C, A–B}
Total Weight: 12

Key Points

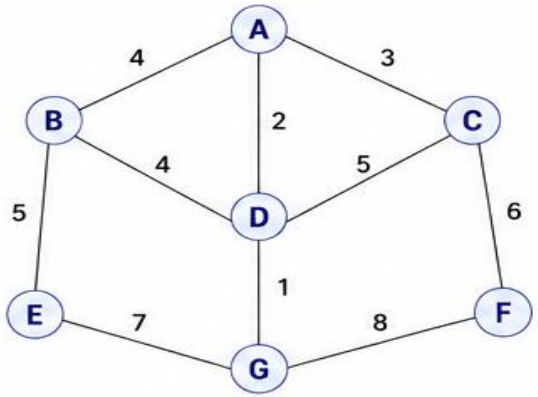
- MST connects all vertices with minimum total edge weight.
- For a graph with V vertices, MST has exactly V – 1 edges.
- Kruskal's Algorithm: Sort edges by weight and add the smallest edge that doesn't form a cycle.

Edges in original graph
Edges in MST

Final MST Weight = 12

Form Minimum Spanning Tree (MST) from a Graph using Kruskal's Algorithm

Original Connected Graph



All Edges Sorted by Weight (Ascending Order)

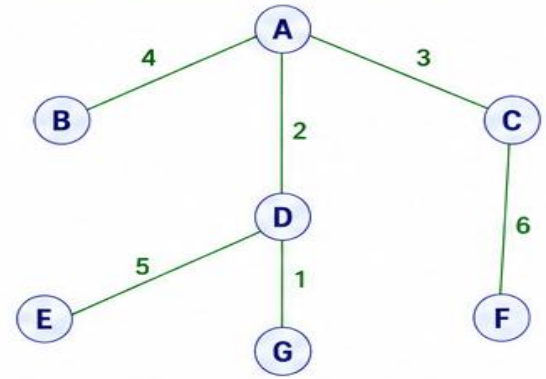
Order	Edge	Weight
1	D – G	1
2	A – D	2
3	A – C	3
4	B – D	4
5	A – B	4
6	C – D	5
7	B – E	5
8	C – F	6
9	E – G	7
10	F – G	8

Kruskal's Algorithm (Idea)

- Sort all edges in non-decreasing order of weight.
- Add the smallest edge that does not form a cycle.
- Repeat until we have (V − 1) edges in MST.

Here, V = 7 ⇒ MST will have 6 edges.

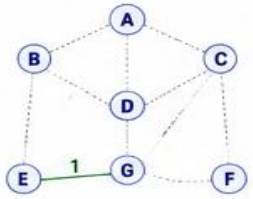
Minimum Spanning Tree (MST)



Total Weight of MST = 1 + 2 + 3 + 4 + 5 + 6 = 21

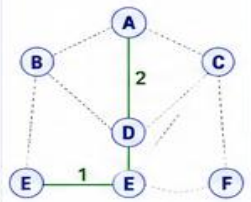
Step-by-Step Execution of Kruskal's Algorithm

1 Consider edge D – G (1)
Add (no cycle)



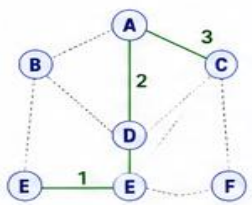
MST Edges: {D–G}
Total Weight = 1

2 Consider edge A – D (2)
Add (no cycle)



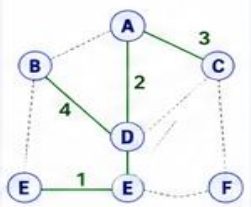
MST Edges: {D–G, A–D}
Total Weight = 3

3 Consider edge A – C (3)
Add (no cycle)



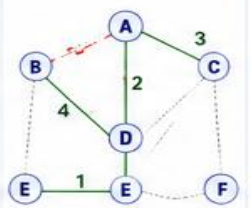
MST Edges: {D–G, A–D, A–C}
Total Weight = 6

4 Consider edge B – D (4)
Add (no cycle)



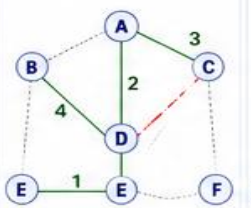
MST Edges: {D–G, A–D, A–C, B–D}
Total Weight = 10

5 Consider edge A – B (4)
Skip (would form cycle)



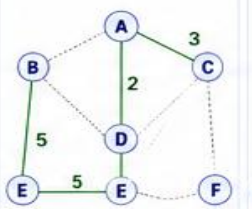
MST Edges: {D–G, A–D, A–C, B–D}
Total Weight = 10

6 Consider edge C – D (5)
Skip (would form cycle)



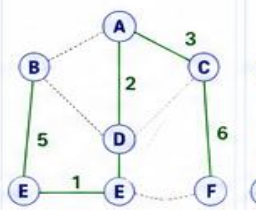
MST Edges: {D–G, A–D, A–C, B–D}
Total Weight = 10

7 Consider edge B – E (5)
Add (no cycle)



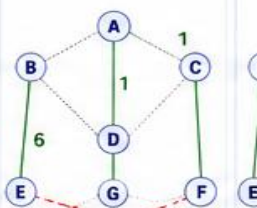
MST Edges: {D–G, A–D, A–C, B–D, B–E}
Total Weight = 15

8 Consider edge C – F (6)
Add (no cycle)



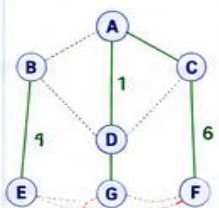
MST Edges: {D–G, A–D, A–C, B–D, B–E, C–F}
Total Weight = 21

9 Consider edge E – G (7)
Skip (would form cycle)



MST Edges: {D–G, A–D, A–C, B–D, B–E, C–F}
Total Weight = 21

10 Consider edge F – G (8)
Skip (would form cycle)



MST Edges: {D–G, A–D, A–C, B–D, B–E, C–F}
Total Weight = 21
(MST Completed)

Key Points

- Kruskal's Algorithm always picks the smallest edge that does not form a cycle.
- It builds a forest and gradually connects all vertices.
- For a graph with V vertices, MST has exactly V − 1 edges.

Edges in original graph
Edges in MST
Edges skipped (would form cycle)

Final MST Weight = 21

Shortest Path Algorithms

- A **Shortest Path Algorithm** is used to find the minimum-cost path between two vertices or from one source vertex to all other vertices in a graph.

The cost may represent:

- Distance
- Time
- Money
- Network latency
- Transportation cost

Dijkstra's Algorithm

- Dijkstra's Algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with **non-negative edge weights**.

- **Working Principle**
  - Start from source vertex.
  - Assign distance 0 to source.
  - Assign ∞ (infinity) to all other vertices.
  - Select the vertex with minimum distance.
  - Update distances of adjacent vertices.
  - Repeat until all vertices are visited.

Dijkstra Algorithm

- Dijkstra(G, Source)

1. Distance[source] = 0
2. Distance[others] = ∞

3. While unvisited vertices exist
   Pick minimum distance vertex
   Mark visited

   For each adjacent vertex
      Update distance

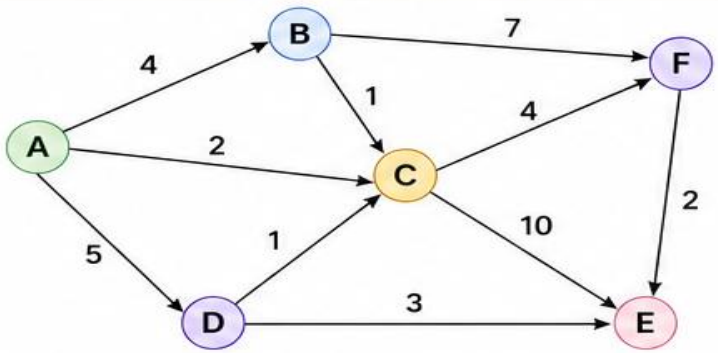
Time Complexity

Implementation	Complexity
Array	$O(V^2)$
Priority Queue	O(E log V)

Dijkstra's Algorithm to Find Shortest Path

Find shortest paths from Source Vertex A to all other vertices.

Weighted Directed Graph



Dijkstra's Algorithm (Idea)

1. Initialize: distance to source = 0, distance to others = ∞.
2. Mark all vertices unvisited.
3. Pick the unvisited vertex with smallest distance.
4. Update distances of its adjacent vertices.
5. Mark it visited.
6. Repeat steps 3–5 until all vertices are visited.

∞ : Infinity (unknown distance)
– : Not applicable

Final Shortest Distances from A

Vertex	Shortest Distance from A	Shortest Path from A
A	0	A
B	3	A → C → B
C	2	A → C
D	3	A → C → D
E	6	A → C → D → E
F	6	A → C → F

Shortest Path from A to F : A → C → F
Total Distance = 6

Step-by-Step Execution

Step	0 Initialization	1 Pick A (min dist = 0)	2 Pick C (min dist = 2)	3 Pick B (min dist = 3)	4 Pick D (min dist = 3)	5 Pick F (min dist = 6)	6 Pick E (min dist = 6)
Selected Vertex	–	A	C	B	D	F	E
Visited Set (after step)	{ }	{A}	{A, C}	{A, C, B}	{A, C, B, D}	{A, C, B, D, F}	{A, C, B, D, F, E}
Distance Table (after updating)	A 0 B ∞ C ∞ D ∞ E ∞ F ∞	A 0 B 4 C 2 D 5 E ∞ F ∞	A 0 B 3 C 2 D 3 E 12 F 6	A 0 B 3 C 2 D 3 E 12 F 6	A 0 B 3 C 2 D 3 E 6 F 6	A 0 B 3 C 2 D 3 E 6 F 6	A 0 B 3 C 2 D 3 E 6 F 6

Key Points

- Dijkstra's algorithm works only for non-negative edge weights.
- At each step, the unvisited vertex with the smallest distance is selected.
- Once a vertex is visited, its shortest distance is finalized.
- Time Complexity: O((V + E) log V) using priority queue.

→ Directed edge (u → v) with weight
⇢ Part of shortest path tree from A

Shortest Path Tree from A

