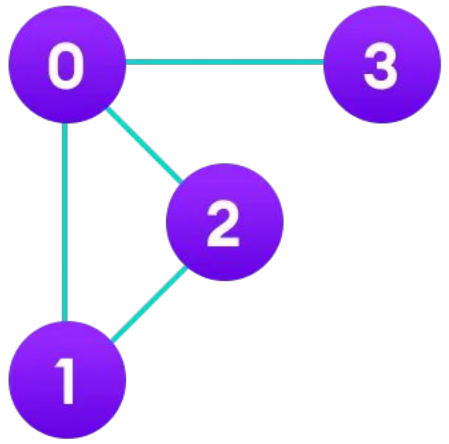
Graph

Abdul Ghafoor

Graphs

- A Graph is a non-linear data structure that consists of vertices (nodes) and edges.
- A vertex, also called a node, is a point or an object in the Graph, and an edge is used to connect two vertices with each other.



Graphs

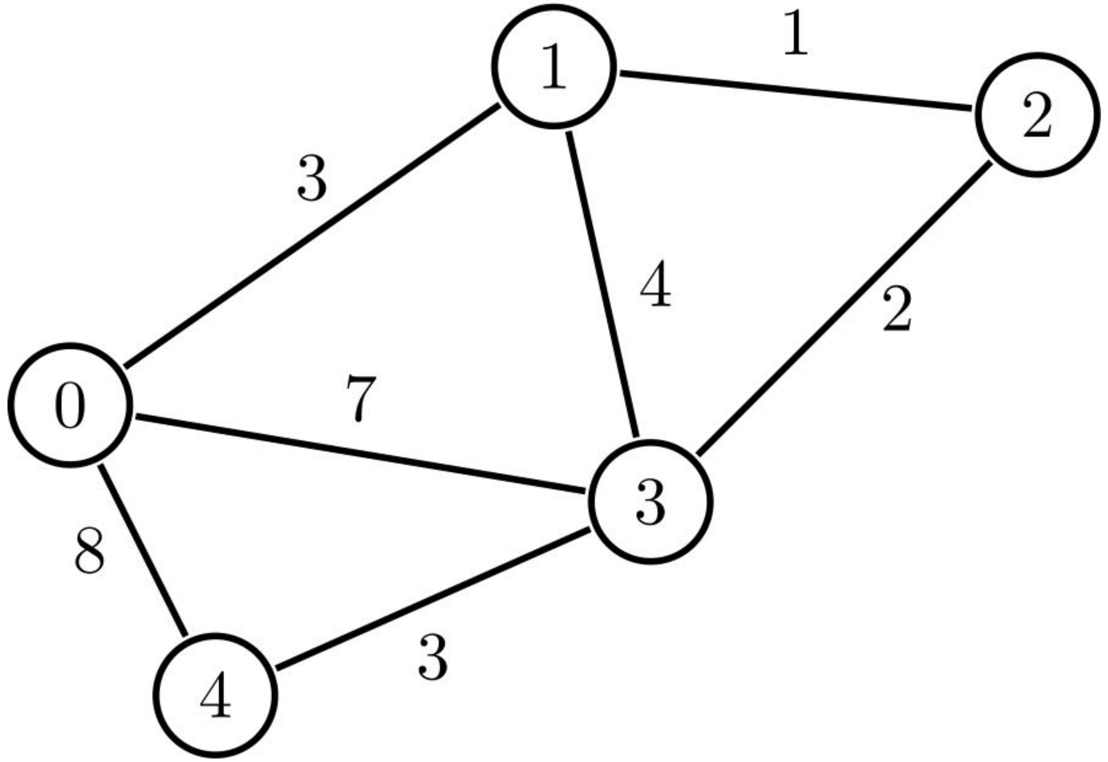
- Graphs are non-linear because the data structure allows us to have different paths to get from one vertex to another, unlike with linear data structures like Arrays or Linked Lists.
- Graphs are used to represent and solve problems where the data consists of objects and relationships between them, such as:

Graphs

- **Social Networks:** Each person is a vertex, and relationships (like friendships) are the edges. Algorithms can suggest potential friends.
- **Maps and Navigation:** Locations, like a town or bus stops, are stored as vertices, and roads are stored as edges. Algorithms can find the shortest route between two locations when stored as a Graph.
- **Internet:** Can be represented as a Graph, with web pages as vertices and hyperlinks as edges.
- **Biology:** Graphs can model systems like neural networks or the spread of diseases.

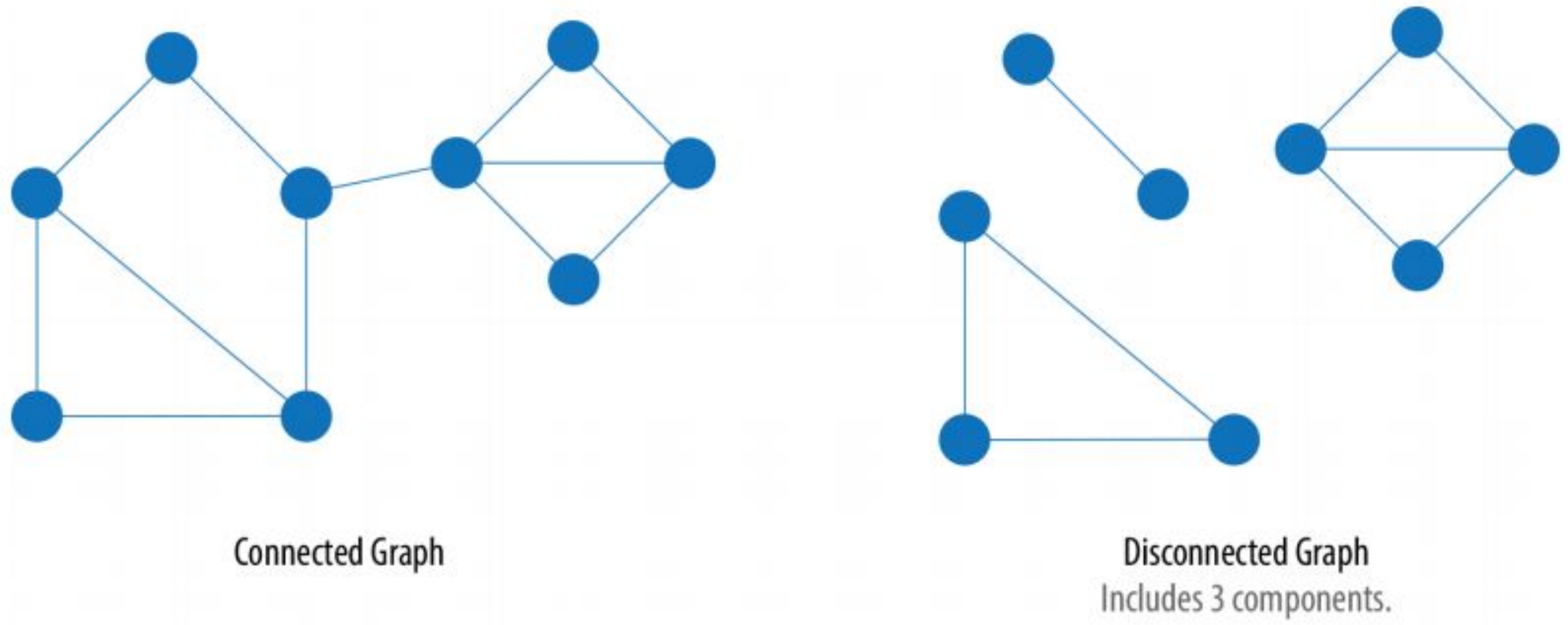
Graph Properties

A **weighted** Graph is a Graph where the edges have values. The weight value of an edge can represent things like distance, capacity, time, or probability.



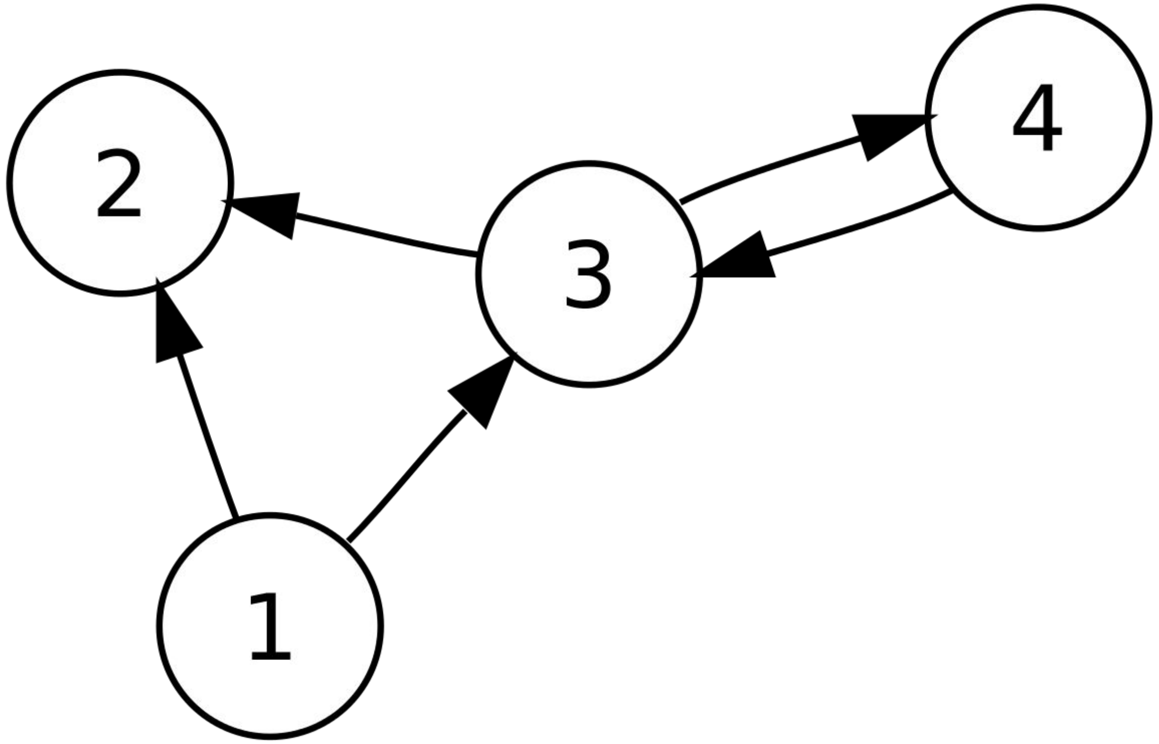
Graph Properties

- A **connected** Graph is when all the vertices are connected through edges somehow. A Graph that is not connected, is a Graph with isolated (disjoint) subgraphs, or single isolated vertices.



Graph Properties

- A **directed** Graph, also known as a digraph, is when the edges between the vertex pairs have a direction. The direction of an edge can represent things like hierarchy or flow.



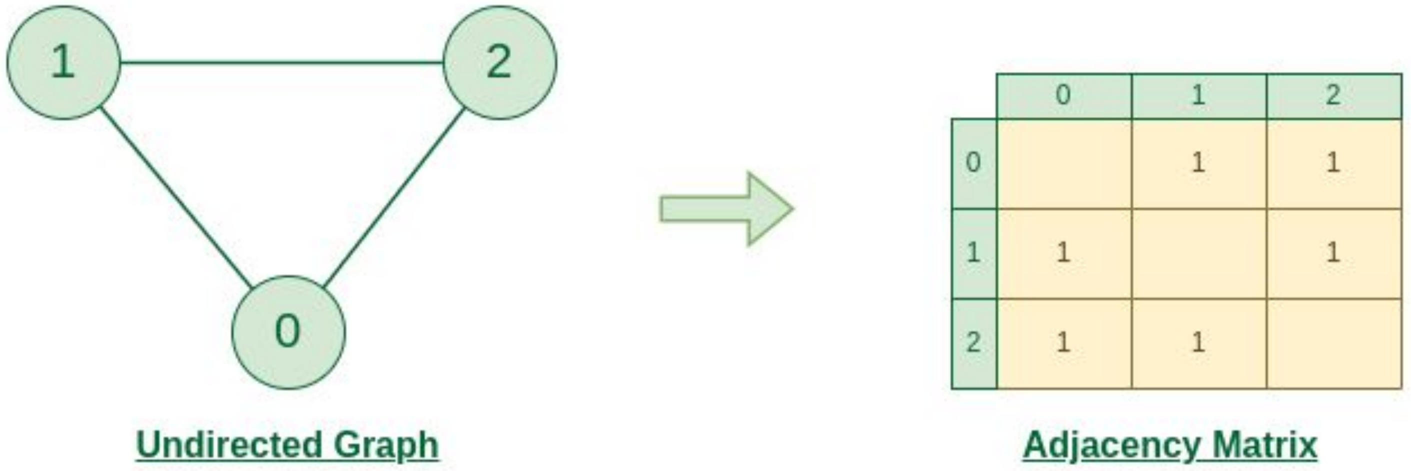
DSA Graphs Implementation

- *Adjacency Matrix*
- *adjacency List*

DSA Graphs Implementation: ***Adjacency Matrix***

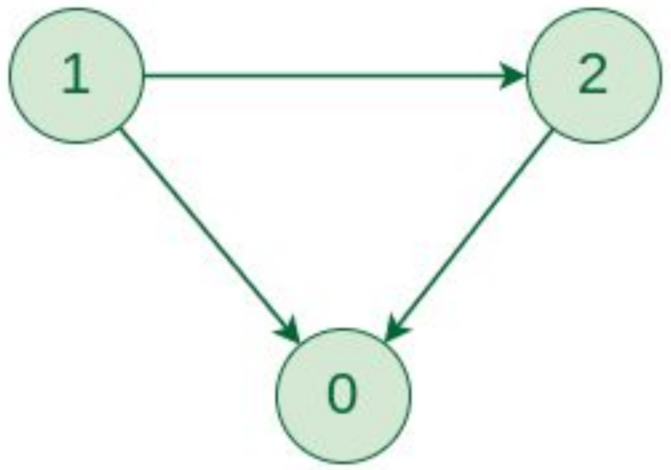
DSA Graphs Implementation: *Adjacency Matrix*

- *An adjacency matrix is a square matrix of N x N size where N is the number of nodes in the graph and it is used to represent the connections between the edges of a graph.*

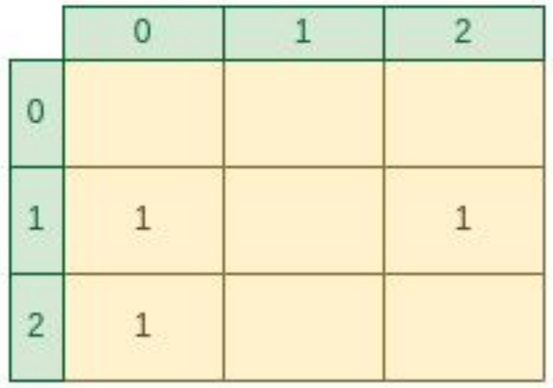


	0	1	2
0		1	1
1	1		1
2	1	1	

Graph Representation of Undirected graph to Adjacency Matrix



Directed Graph



	0	1	2
0			
1	1		1
2	1		

Adjacency Matrix

Graph Representation of Directed graph to Adjacency Matrix

DSA Graphs Implementation: ***Adjacency Matrix***

- Representation of unweighted and undirected graph
- Representation of weighted and undirected graph
- Representation of weighted and directed graph
- Representation of unweighted and directed graph

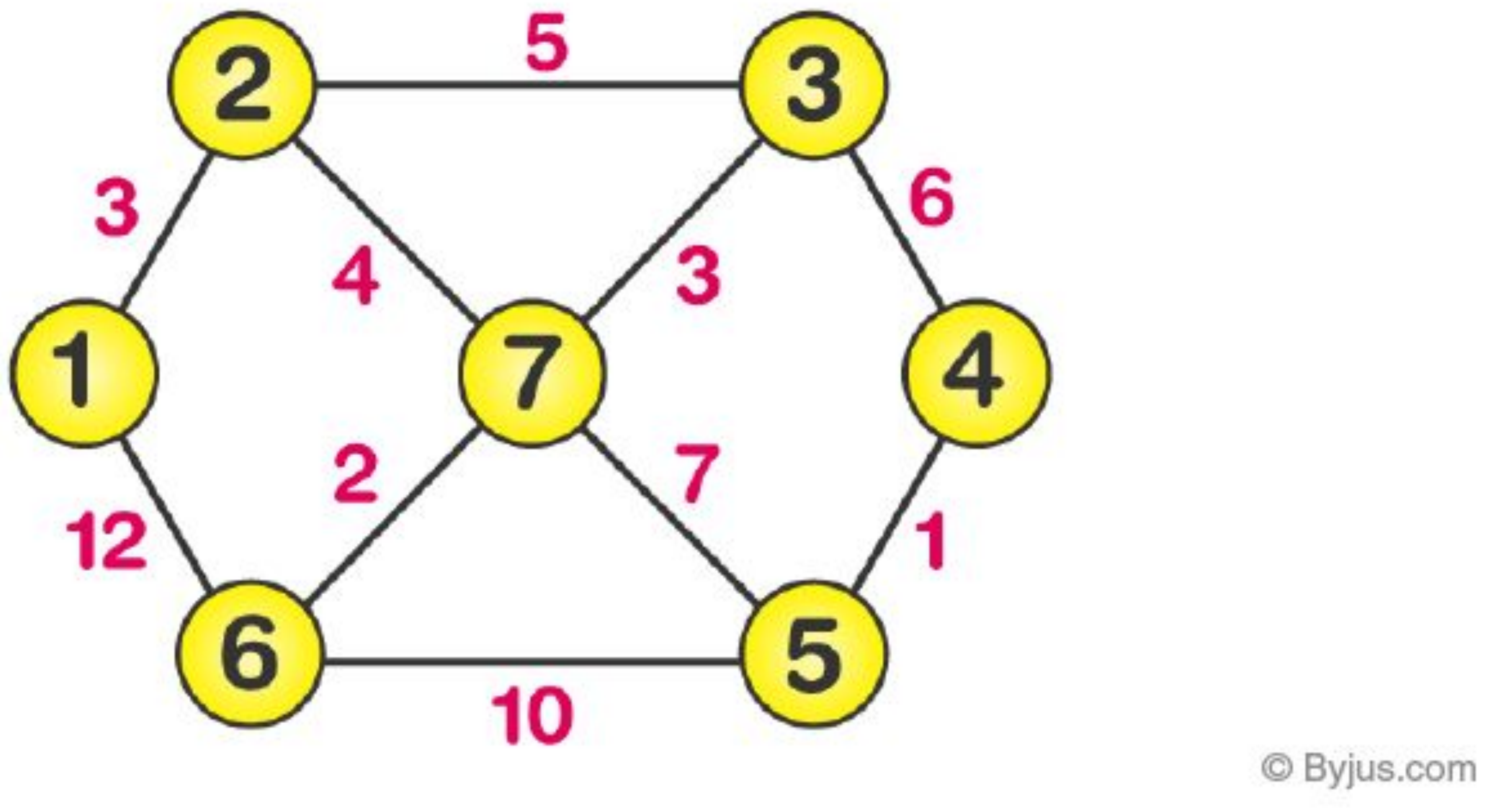
Characteristics of the adjacency matrix are:

- The size of the matrix is determined by the number of vertices (or nodes) in a graph.
- The edges in the graph are represented as values in the matrix. In case of unweighted graphs, the values are 0 or 1. In case of weighted graphs, the values are weights of the edges if edges are present, else 0.
- If the graph has few edges, the matrix will be sparse.

How to build an Adjacency Matrix:

- Create an **n x n** matrix where **n** is the number of vertices in the graph.
- Initialize all elements to 0.
- For each edge (u, v) in the graph, if the graph is undirected mark a[u][v] and a[v][u] as 1, and if the edge is directed from **u** to **v**, mark a[u][v] as the 1. (Cells are filled with edge weight if the graph is weighted)

Represent the following undirected graph into adjacency Matrix



Shortest Path Algorithms in Graph

Shortest Path Algorithms in Graph

Algorithms for Weighted Graphs

1. Dijkstra’s Algorithm:
2. Bellman-Ford Algorithm:
3. Floyd-Warshall Algorithm:
4. *A* Search Algorithm* (A*)

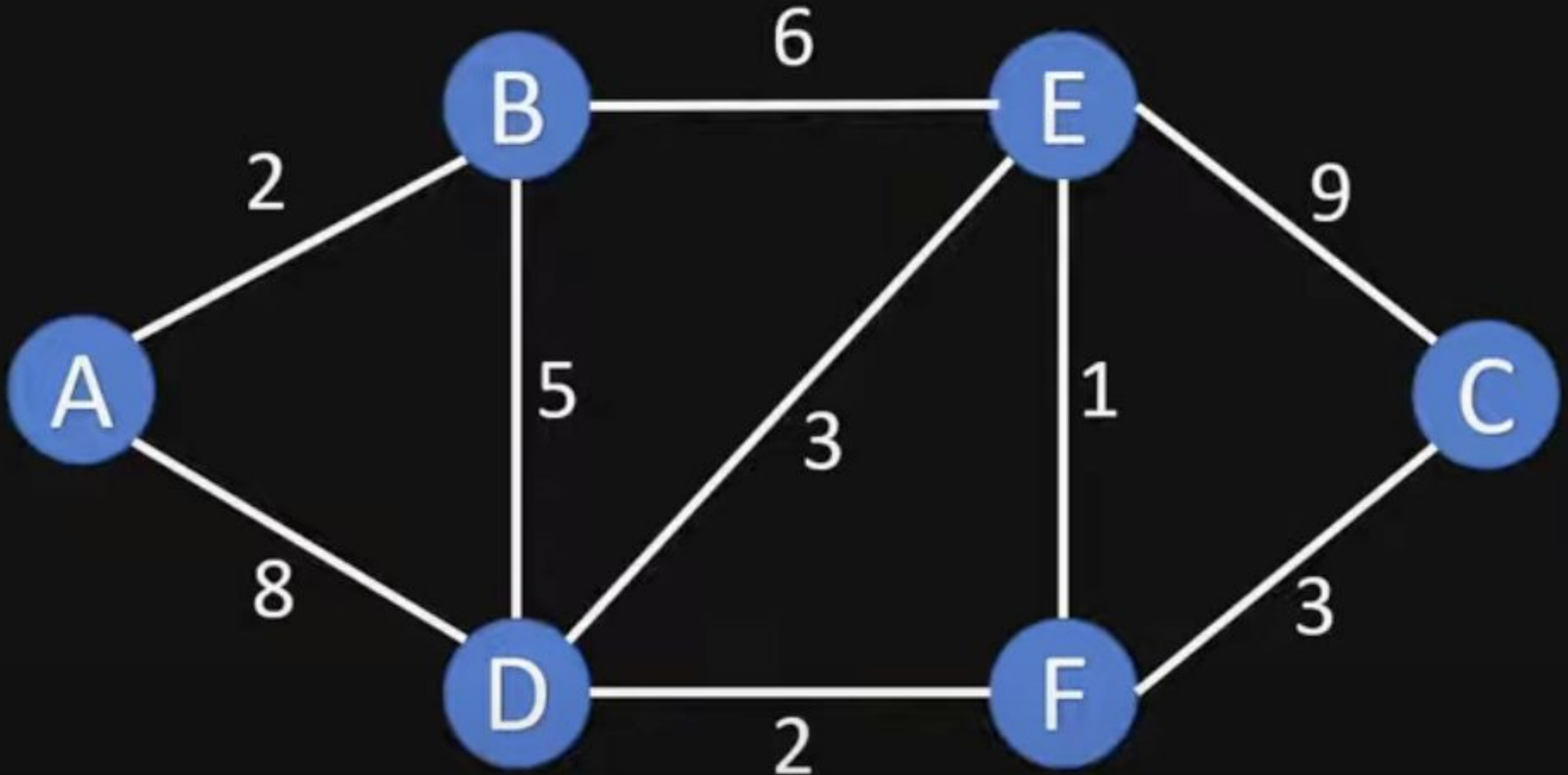
Algorithms for Unweighted Graphs

1. Breadth-First Search (BFS):
2. DFS (modified):

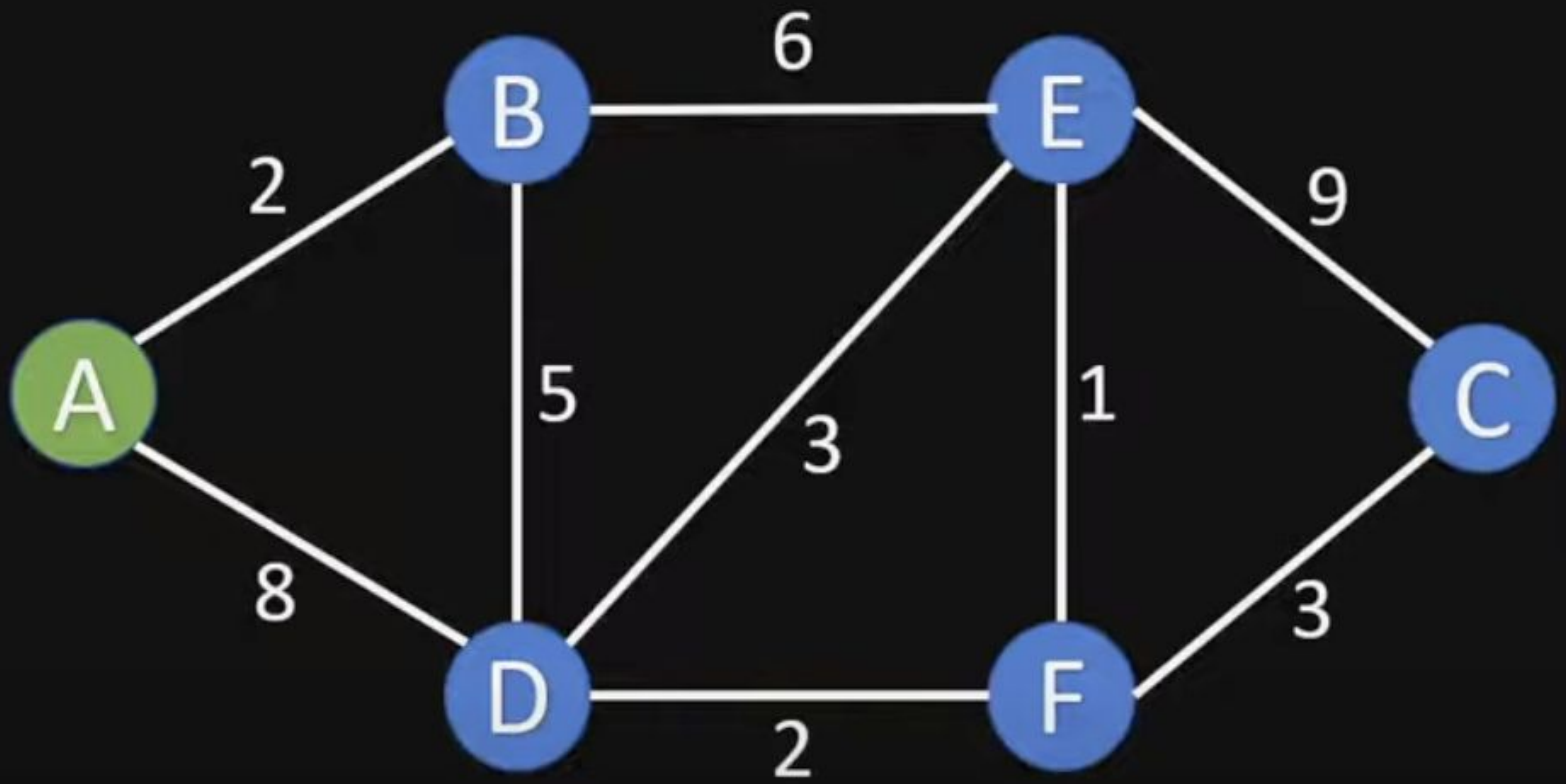
Dijkstra′s Algorithm

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Dijkstra’s Shortest Path Algorithm



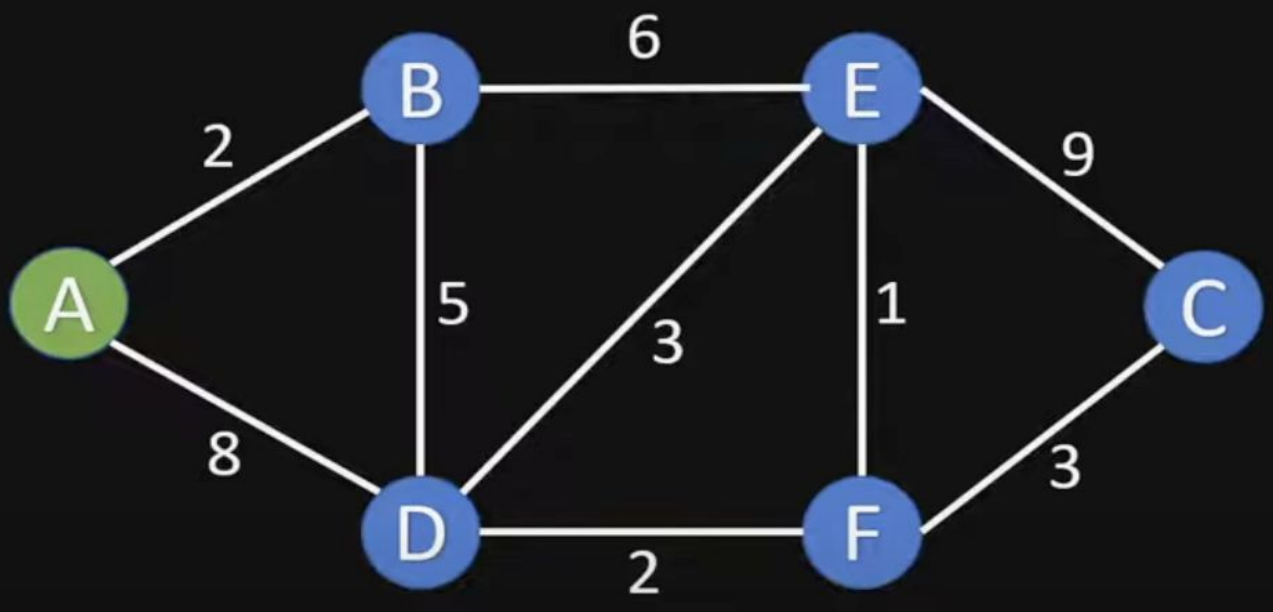
1. Mark all nodes as unvisited



Visited Nodes: []

Unvisited Nodes: [A, B, C, D, E, F]

2. Assign to all nodes a tentative distance value



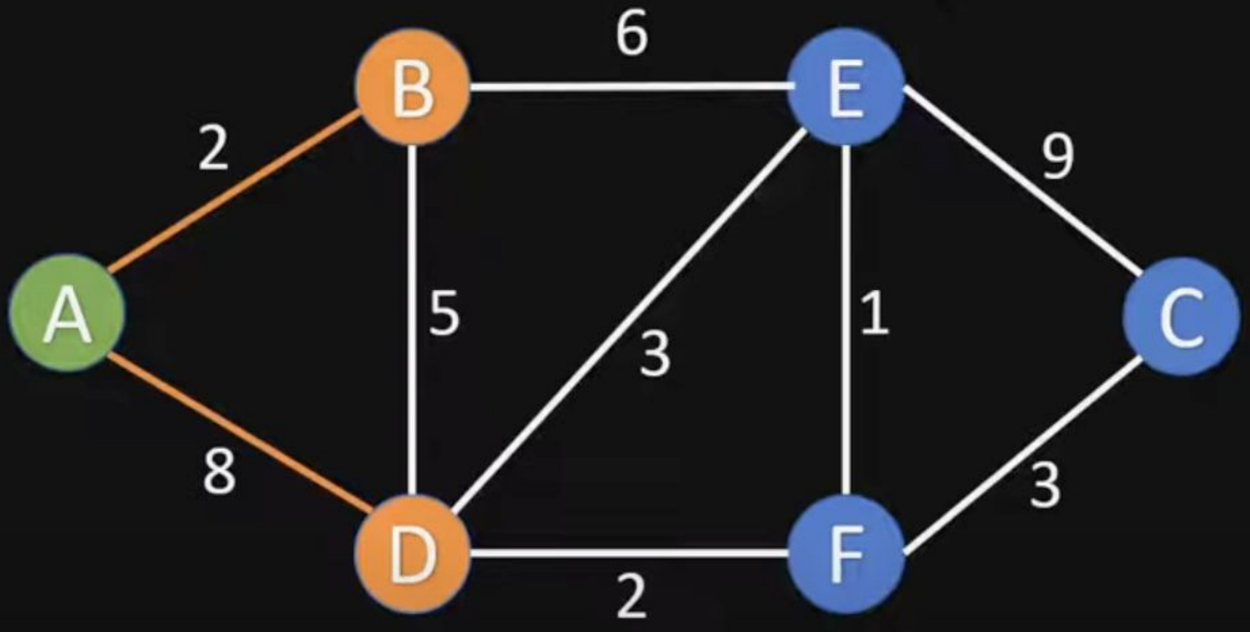
Node	Shortest Distance	Previous Node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

Visited Nodes: []

Unvisited Nodes: [A, B, C, D, E, F]

3. For the current node calculate the distance to all unvisited neighbours

3.1. Update shortest distance, if new distance is shorter than old distance

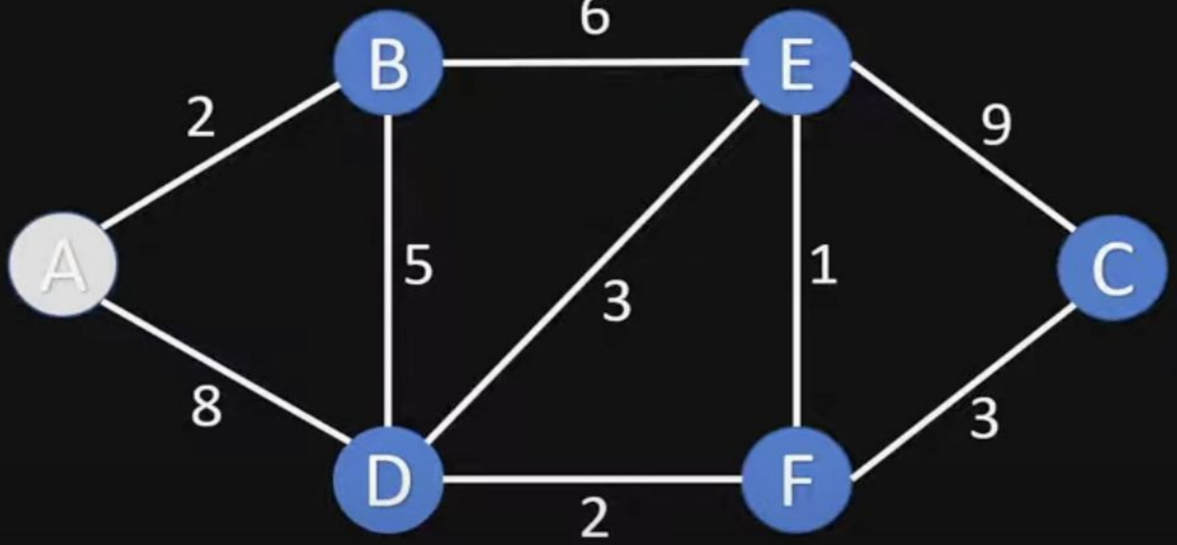


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

Visited Nodes: []

Unvisited Nodes: [A, B, C, D, E, F]

4. Mark current node as visited

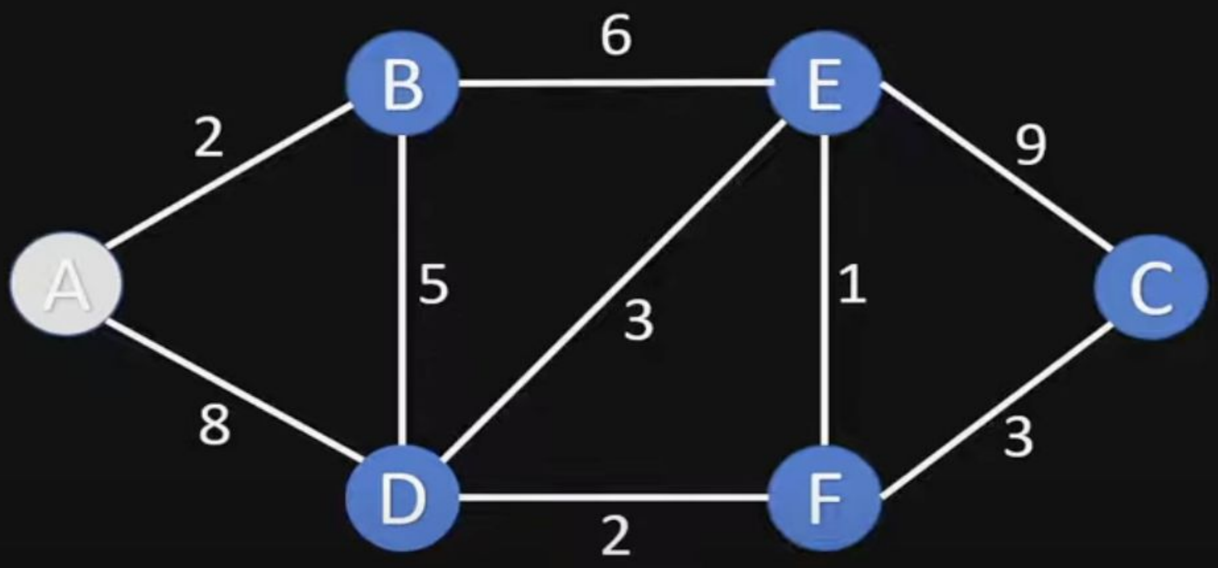


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

Visited Nodes: [A]

Unvisited Nodes: [B, C, D, E, F]

5. Choose new current node from unvisited nodes with minimal distance

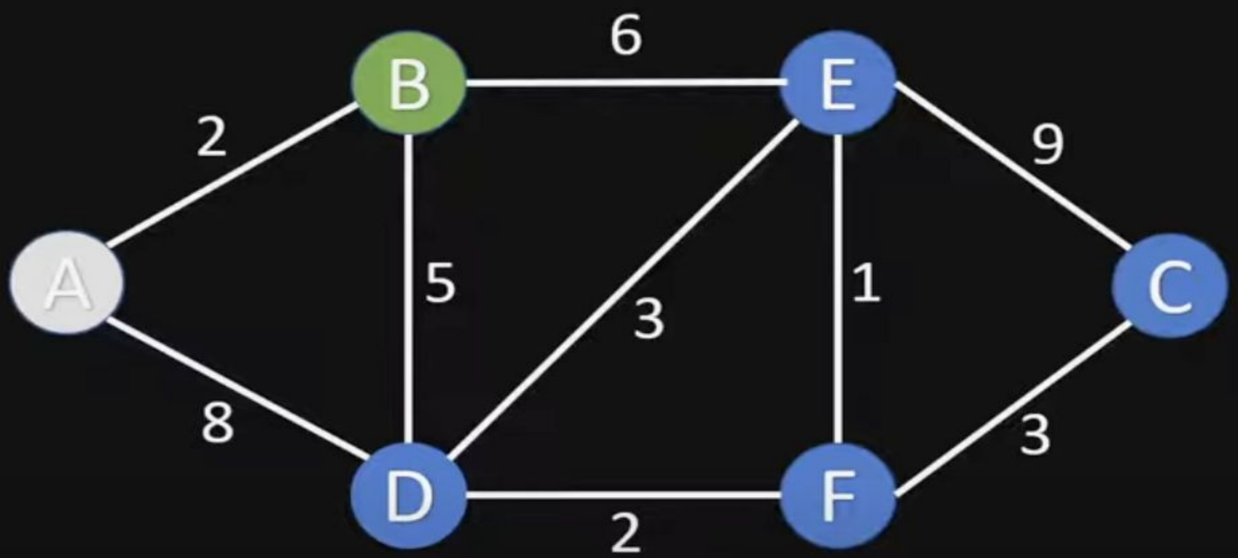


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

Visited Nodes: [A]

Unvisited Nodes: [B, C, D, E, F]

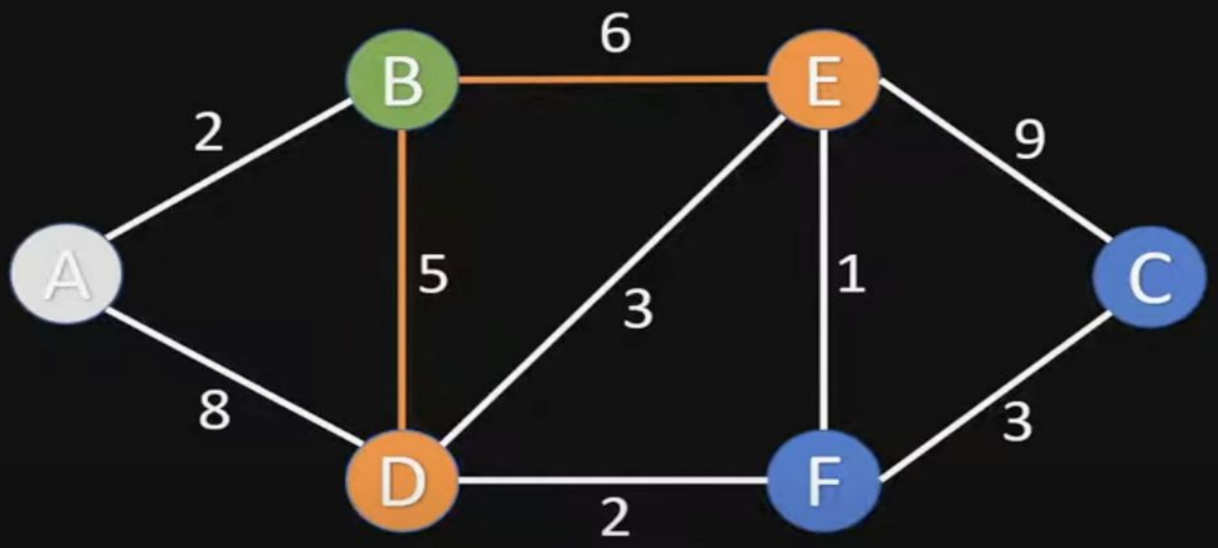
5. Choose new current node from unvisited nodes with minimal distance



Visited Nodes: [A]   Unvisited Nodes: [B, C, D, E, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

3. For the current node calculate the distance to all unvisited neighbours

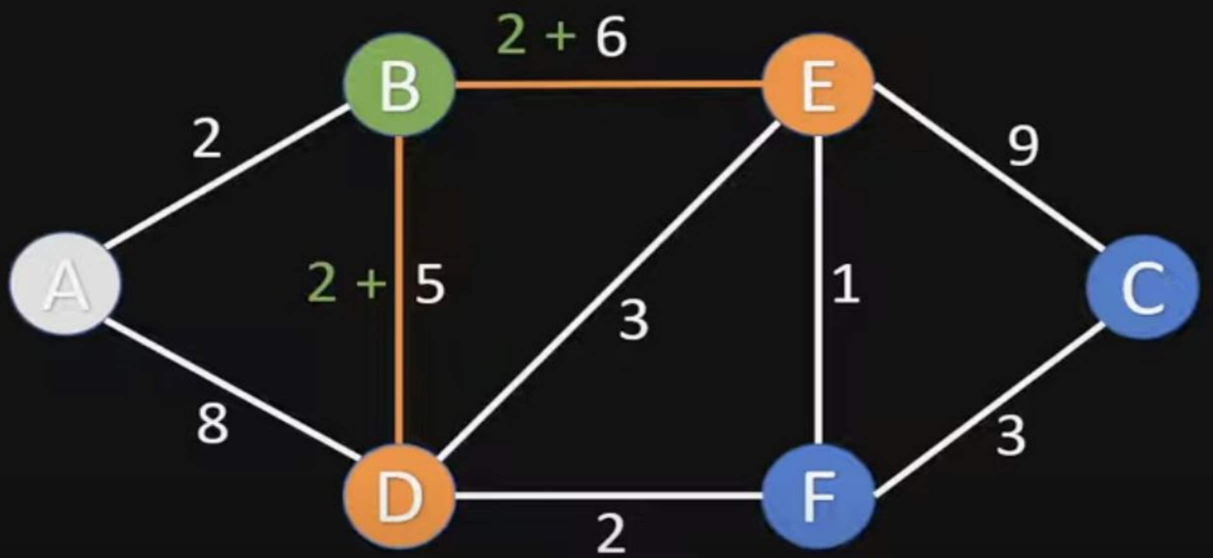


Visited Nodes: [A]

Unvisited Nodes: [B, C, D, E, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

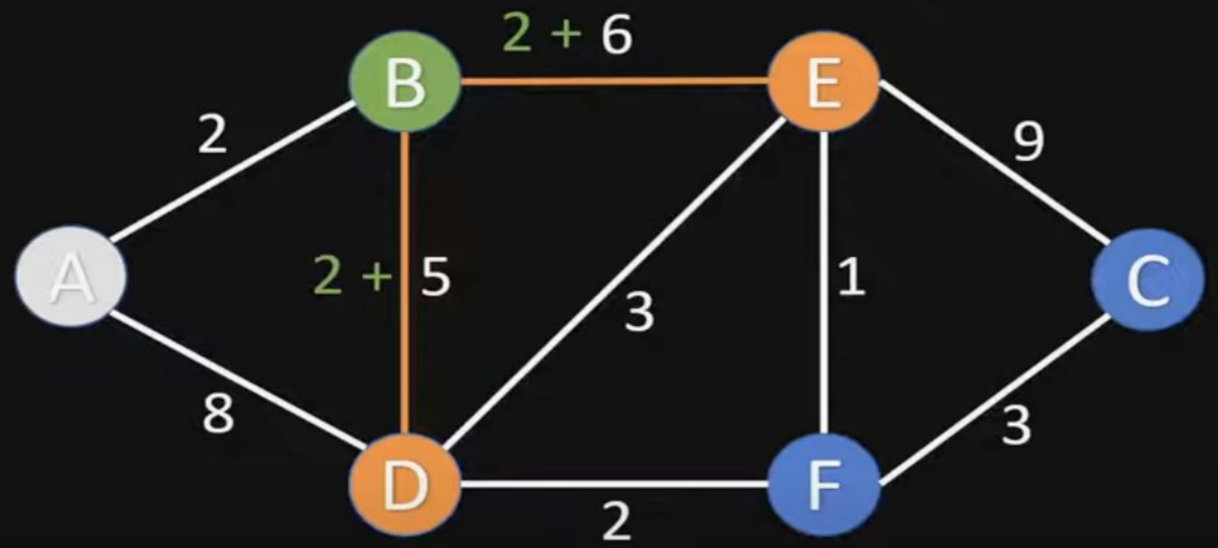
3. For the current node calculate the distance to all unvisited neighbours



Visited Nodes: [A]   Unvisited Nodes: [B, C, D, E, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

3. For the current node calculate the distance to all unvisited neighbours
3.1. Update shortest distance, if new distance is shorter than old distance

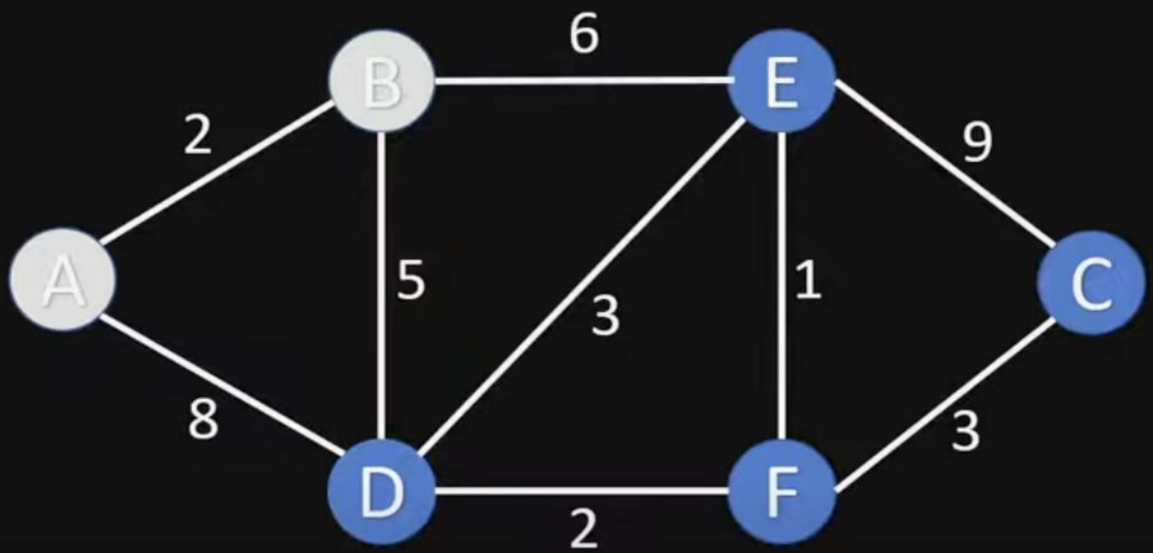


Visited Nodes: [A]

Unvisited Nodes: [B, C, D, E, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**7**	B
E	**8**	B
F	∞	

4. Mark current node as visited

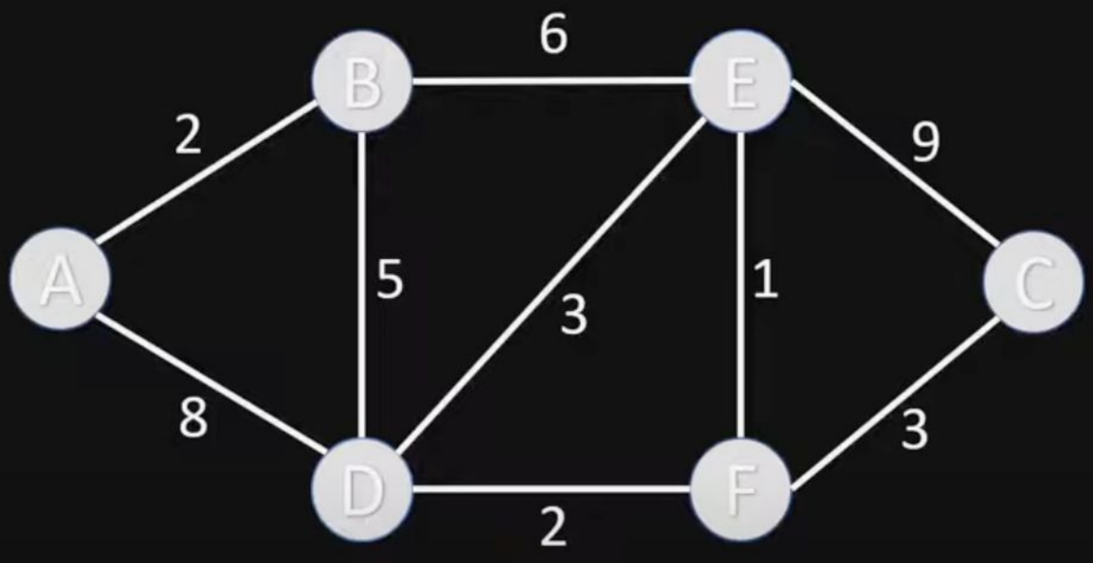


Visited Nodes: [A, B]   Unvisited Nodes: [C, D, E, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**7**	B
E	**8**	B
F	∞	

Repeat same steps for all Nodes

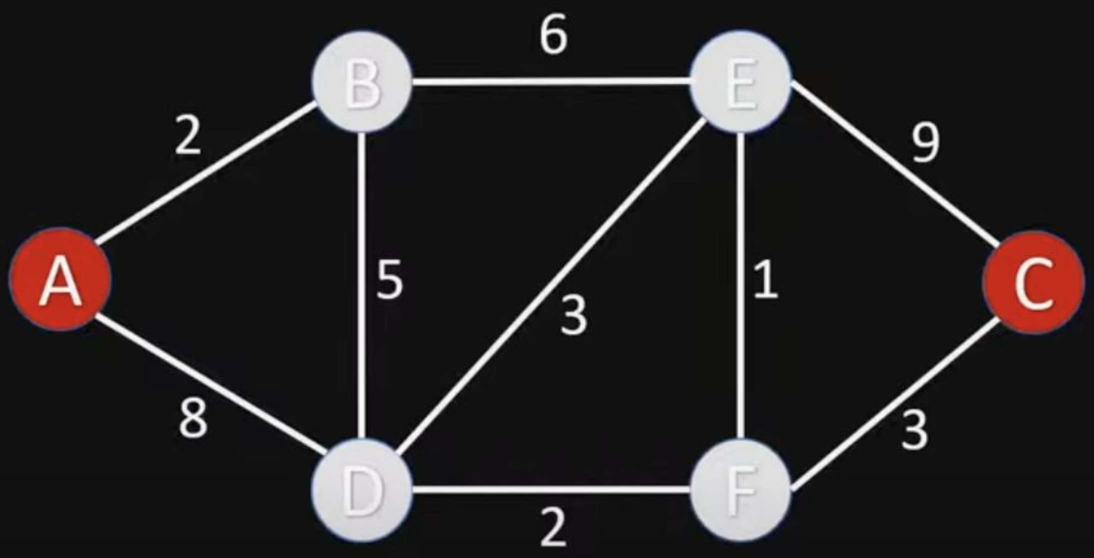
4. Mark current node as visited



Visited Nodes: [A, B, D, E, F, C]   Unvisited Nodes: []

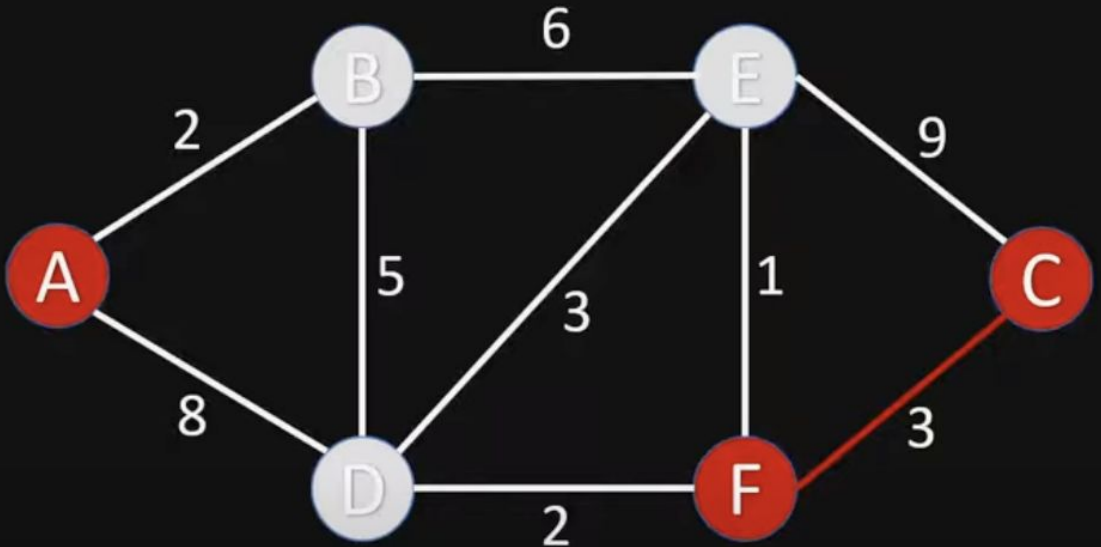
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Get shortest path from A to C



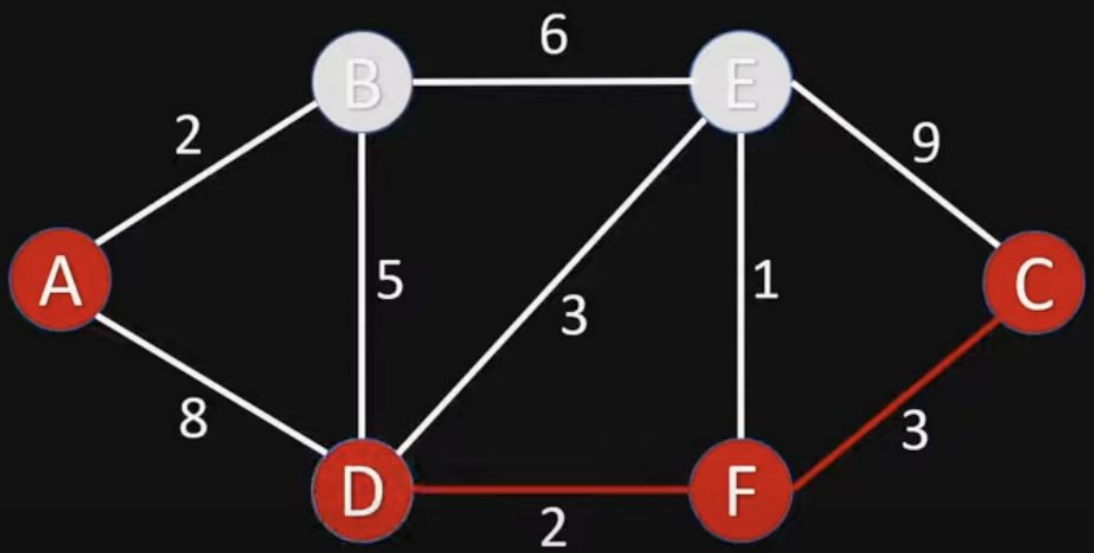
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Get shortest path from A to C



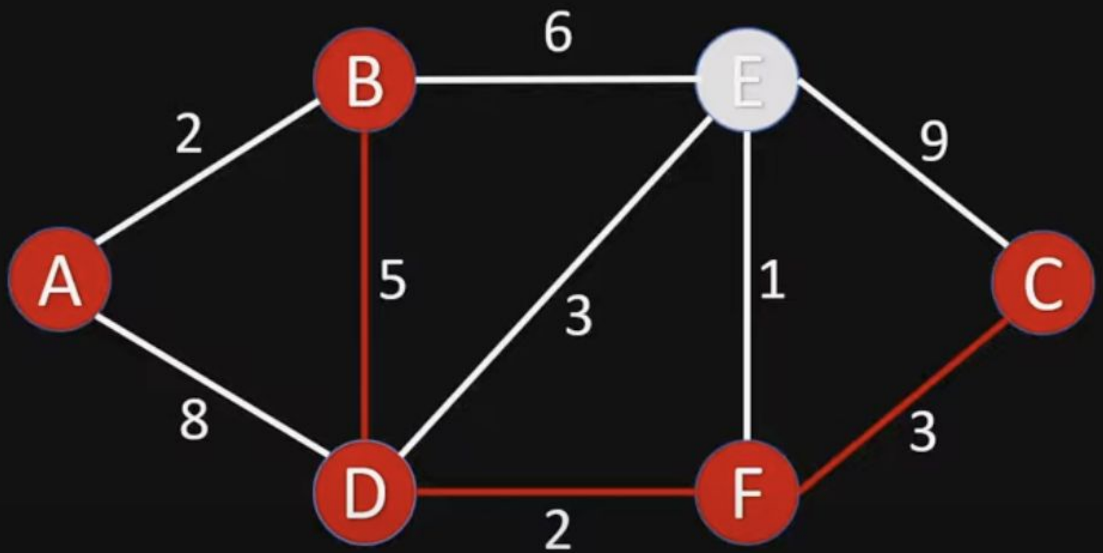
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Get shortest path from A to C



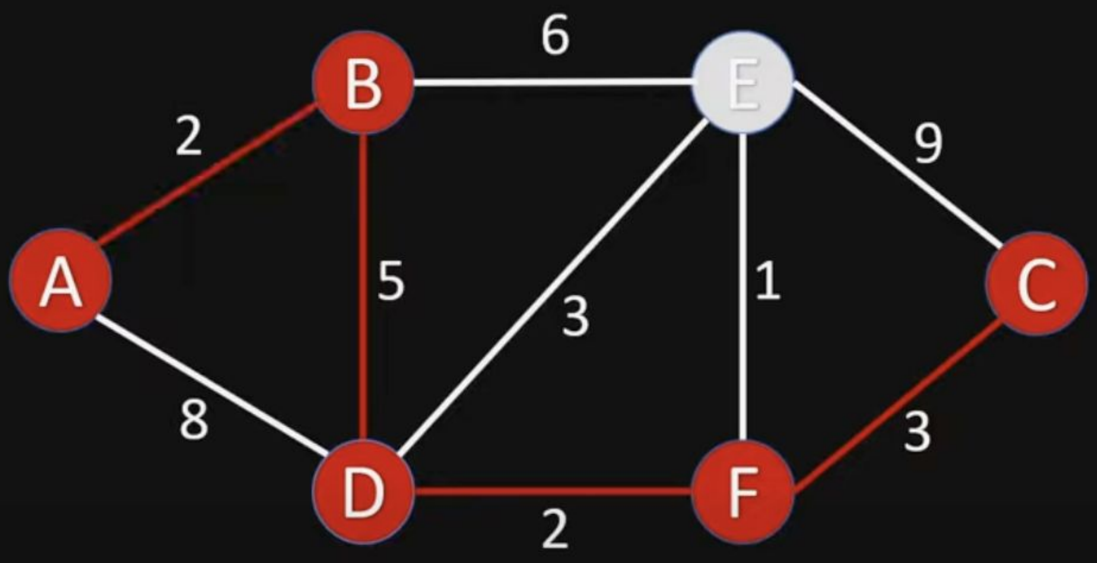
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Get shortest path from A to C



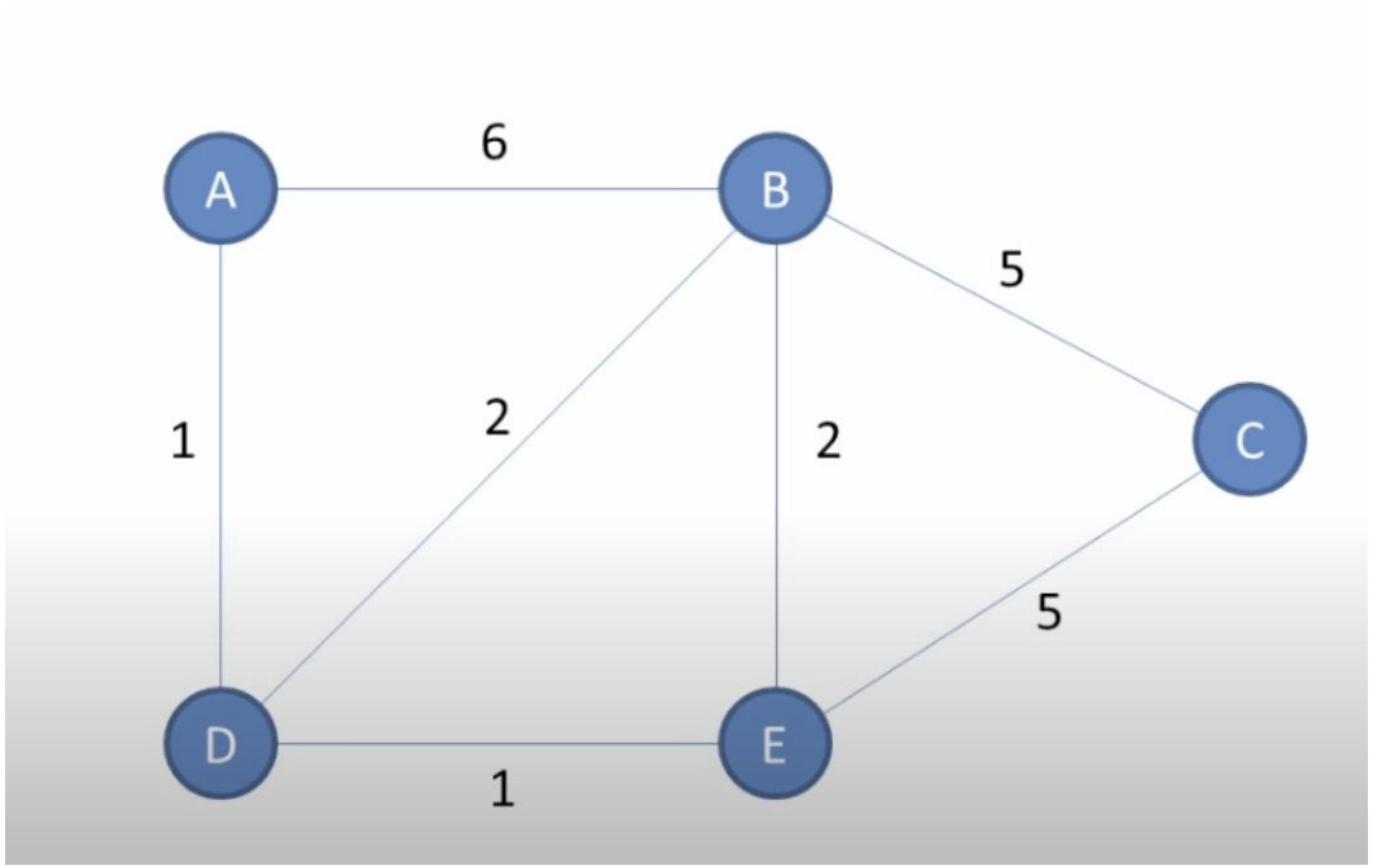
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Get shortest path from A to C



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Apply Dijkstra’s Algorithm on following to find the shortest path between A to C



DSA Graphs Implementation: *Adjacency List*