

# GRAPHS

**UNIT IV:** Graph Theory Terminology, Graph Representations, Graph operations- Graph Traversals (BFS & DFS), Connected components, Spanning Trees, Biconnected Components, Minimum Spanning Trees- Krushkal's Algorithm , Prim's Algorithm, Shortest paths, Transitive closure, All pairs Shortest path-Marshall's Algorithm.

## BASIC CONCEPTS

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

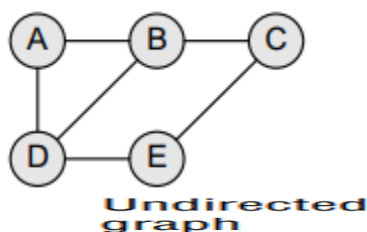
## WHY GRAPHS ARE USEFUL

Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:

- Family trees: in which the member nodes have an edge from parent to each of their children.
- Transportation networks: in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

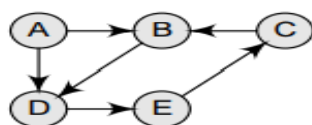
### Definition

A graph  $G$  is defined as an ordered set  $(V, E)$ , where  $V(G)$  represents the set of vertices and  $E(G)$  represents the edges that connect these vertices.



A graph with  $V(G) = \{A, B, C, D \text{ and } E\}$  and  $E(G) = \{(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)\}$ . Note that there are five vertices or nodes and six edges in the graph.

A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A.



In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

Directed Graph

## **Graph Terminology**

### **Adjacent nodes or neighbours**

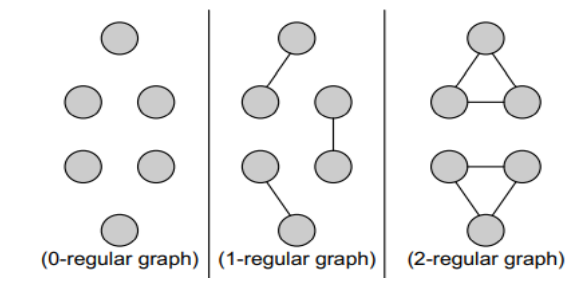
For every edge,  $e = (u, v)$  that connects nodes  $u$  and  $v$ , the nodes  $u$  and  $v$  are the end-points and are said to be the adjacent nodes or neighbours.

### **Degree of a node**

Degree of a node  $u$ ,  $\text{deg}(u)$ , is the total number of edges containing the node  $u$ . If  $\text{deg}(u) = 0$ , it means that  $u$  does not belong to any edge and such a node is known as an isolated node.

### **Regular graph**

It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree  $k$  is called a  $k$ -regular graph or a regular graph of degree  $k$ .



### **Path**

A path  $P$  written as  $P = \{v_0, v_1, v_2, \dots, v_n\}$ , of length  $n$  from a node  $u$  to  $v$  is defined as a sequence of  $(n+1)$  nodes. Here,  $u = v_0$ ,  $v = v_n$  and  $v_{i-1}$  is adjacent to  $v_i$  for  $i = 1, 2, 3, \dots, n$ .

### **Closed path**

A path  $P$  is known as a closed path if the edge has the same end-points. That is, if  $v_0 = v_n$ .

### **Simple path**

A path  $P$  is known as a simple path if all the nodes in the path are distinct with an exception that  $v_0$  may be equal to  $v_n$ . If  $v_0 = v_n$ , then the path is called a closed simple path.

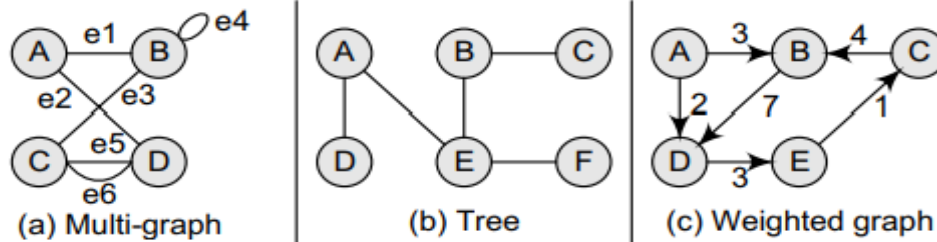
### **Cycle**

A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).

### **Connected graph**

A graph is said to be connected if for any two vertices  $(u, v)$  in  $V$  there is a path from  $u$  to  $v$ . That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph

**Complete graph** A graph  $G$  is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has  $n(n-1)/2$  edges, where  $n$  is the number of nodes in  $G$



### Labelled graph or weighted graph

A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by  $w(e)$  is a positive value which indicates the cost of traversing the edge.

**Multiple edges** Distinct edges which connect the same end-points are called multiple edges. That is,  $e = (u, v)$  and  $e' = (u, v)$  are known as multiple edges of  $G$ .

**Loop** An edge that has identical end-points is called a loop. That is,  $e = (u, u)$ .

**Multi-graph** A graph with multiple edges and/or loops is called a multi-graph.

**Size of a graph** The size of a graph is the total number of edges in it

### Directed Graphs

A directed graph  $G$ , also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair  $(u, v)$  of nodes in  $G$ . For an edge  $(u, v)$ ,

- The edge begins at  $u$  and terminates at  $v$ .
- $u$  is known as the origin or initial point of  $e$ . Correspondingly,  $v$  is known as the destination or terminal point of  $e$ .
- $u$  is the predecessor of  $v$ . Correspondingly,  $v$  is the successor of  $u$ .  $\sum$  Nodes  $u$  and  $v$  are adjacent to each other

### Terminology of a Directed Graph

**Out-degree of a node** The out-degree of a node  $u$ , written as  $\text{outdeg}(u)$ , is the number of edges that originate at  $u$ .

**In-degree of a node** The in-degree of a node  $u$ , written as  $\text{indeg}(u)$ , is the number of edges that terminate at  $u$ .

**Degree of a node** The degree of a node, written as  $\text{deg}(u)$ , is equal to the sum of in-degree and out-degree of that node. Therefore,  $\text{deg}(u) = \text{indeg}(u) + \text{outdeg}(u)$ .

**Isolated vertex** A vertex with degree zero. Such a vertex is not an end-point of any edge.

**Pendant vertex** (also known as leaf vertex) A vertex with degree one.

**Cut vertex** A vertex which when deleted would disconnect the remaining graph.

**Source** A node  $u$  is known as a source if it has a positive out-degree but a zero in-degree.

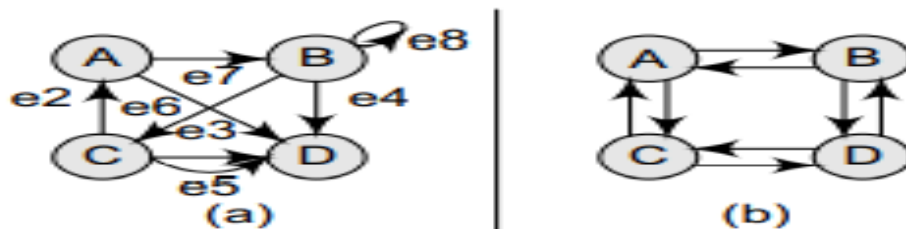
**Sink** A node  $u$  is known as a sink if it has a positive in-degree but a zero out-degree.

**Reachability** A node  $v$  is said to be reachable from node  $u$ , if and only if there exists a (directed) path from node  $u$  to node  $v$ . For example, if you consider the directed graph given in Fig. 13.5(a), you will observe that node  $D$  is reachable from node  $A$ .

**Strongly connected directed graph** A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in  $G$ . That is, if there is a path from node  $u$  to  $v$ , then there must be a path from node  $v$  to  $u$ .

**Weakly connected digraph** A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

**Parallel/Multiple edges** Distinct edges which connect the same end-points are called multiple edges. That is,  $e = (u, v)$  and  $e' = (u, v)$  are known as multiple edges of  $G$ . In below diagram  $e_3$  and  $e_5$  are multiple edges connecting nodes  $C$  and  $D$ .



(a) Directed acyclic graph and (b) strongly connected directed acyclic graph

**Simple directed graph** A directed graph  $G$  is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycles with an exception that it cannot have more than one loop at a given node.

## **REPRESENTATION OF GRAPHS**

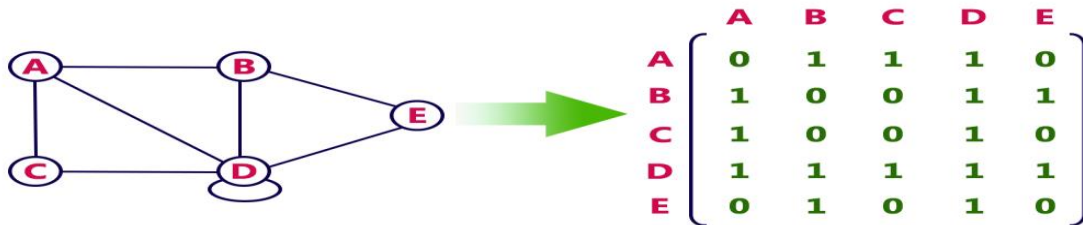
There are three common ways of storing graphs in the computer's memory.

1. Adjacency Matrix
2. Adjacency List

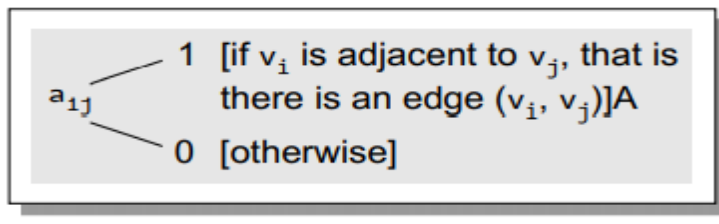
## Adjacency Matrix Representation

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

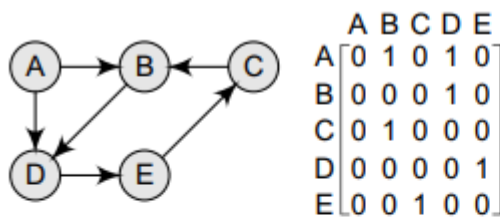
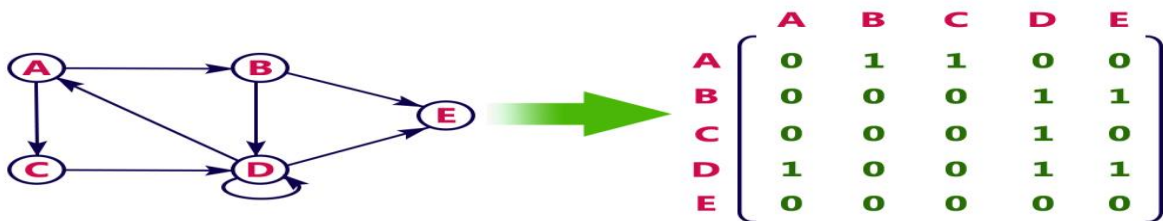
For example, consider the following undirected graph representation...



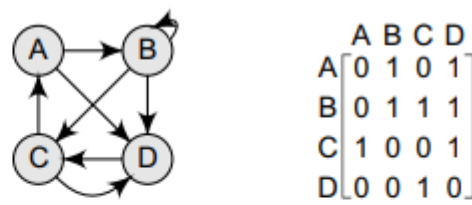
Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.



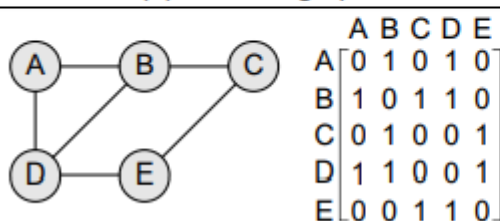
Directed graph representation...



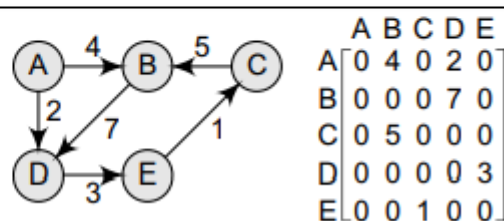
(a) Directed graph



(b) Directed graph with loop



(c) Undirected graph



(d) Weighted graph

Graphs and their corresponding adjacency matrices

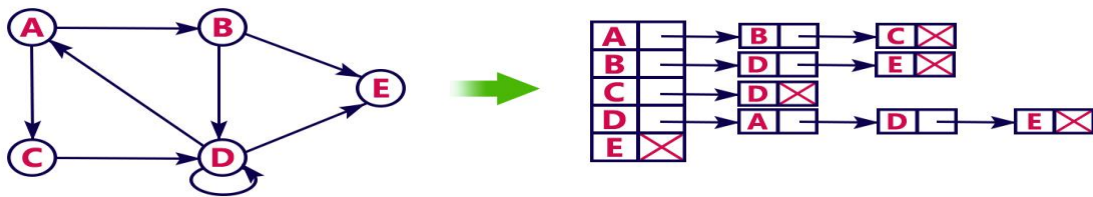
From the above examples, we can draw the following conclusions:

1. For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
2. The adjacency matrix of an undirected graph is symmetric.
3. The memory use of an adjacency matrix is  $O(n^2)$ , where  $n$  is the number of nodes in the graph.
4. Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
5. The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

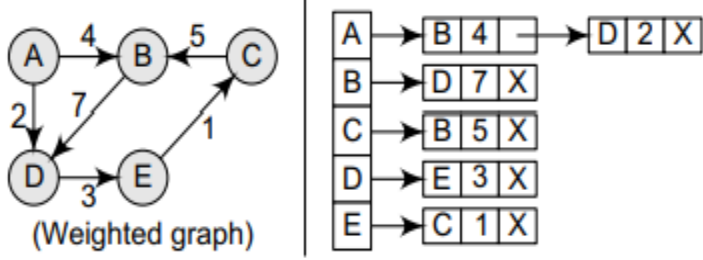
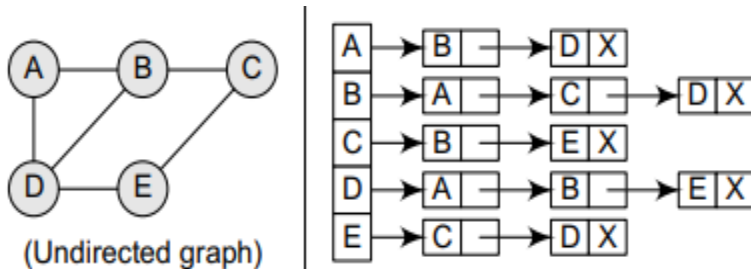
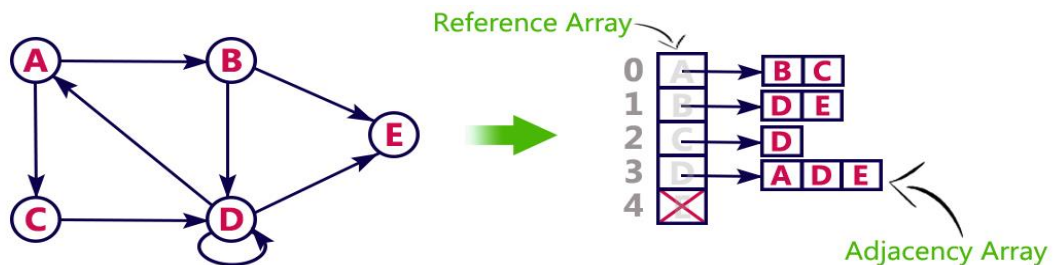
## Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



Adjacency list for an undirected graph and a weighted graph

The key advantages of using an adjacency list are:

- 1 It is easy to follow and clearly shows the adjacent nodes of a particular node.
2. It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
3. Adding new nodes in  $G$  is easy and straightforward when  $G$  is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.
4. For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in  $G$ .
5. For an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in  $G$  because an edge  $(u, v)$  means an edge from node  $u$  to  $v$  as well as an edge from  $v$  to  $u$ .

## Graph Traversal

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

### Breadth First Search (BFS) Algorithm

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighboring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

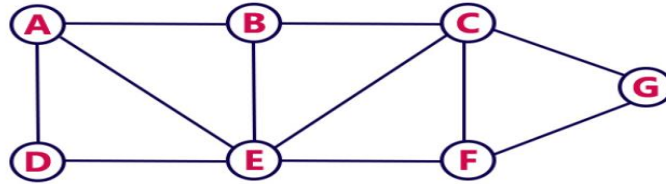
We use the following steps to implement BFS traversal...

- **Step 1** - Define a Queue of size total number of vertices in the graph.
- **Step 2** - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3** - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4** - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5** - Repeat steps 3 and 4 until queue becomes empty.
- **Step 6** - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph



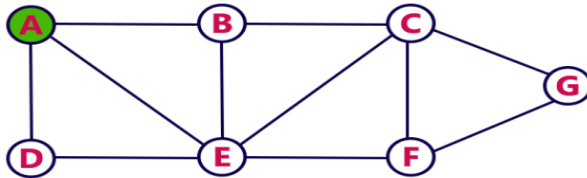
Example

Consider the following example graph to perform BFS traversal



**Step 1:**

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

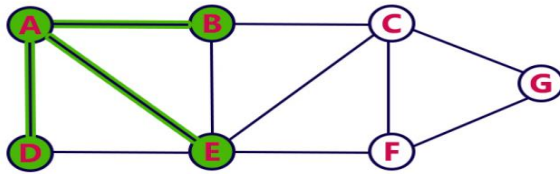


Queue



**Step 2:**

- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

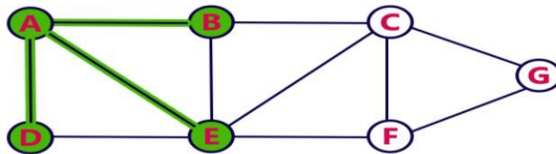


Queue



**Step 3:**

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

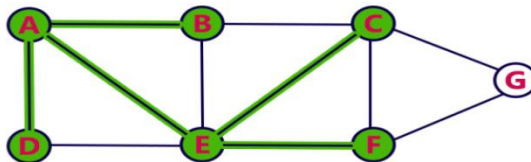


Queue



**Step 4:**

- Visit all adjacent vertices of **E** which are not visited (**C, F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

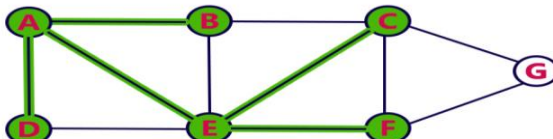


Queue



**Step 5:**

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



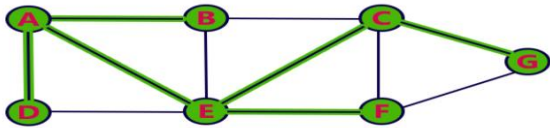
Queue





**Step 6:**

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

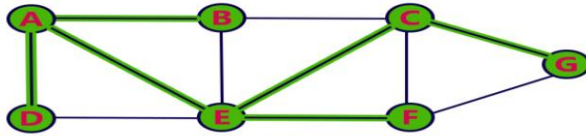


Queue



**Step 7:**

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

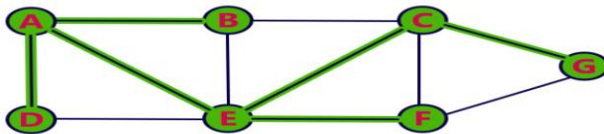


Queue



**Step 8:**

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



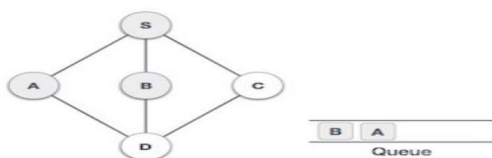
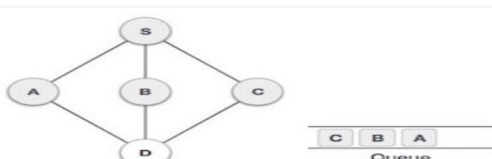
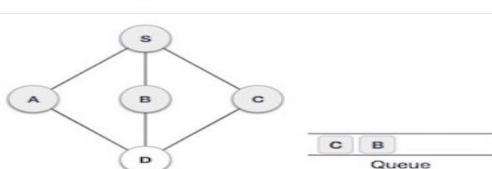
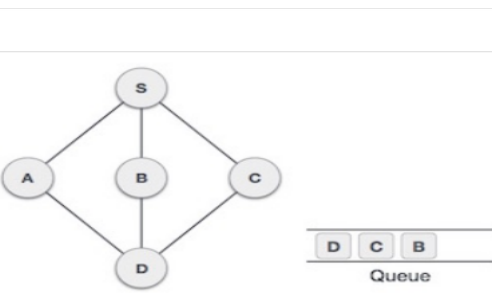
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Example 2:

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

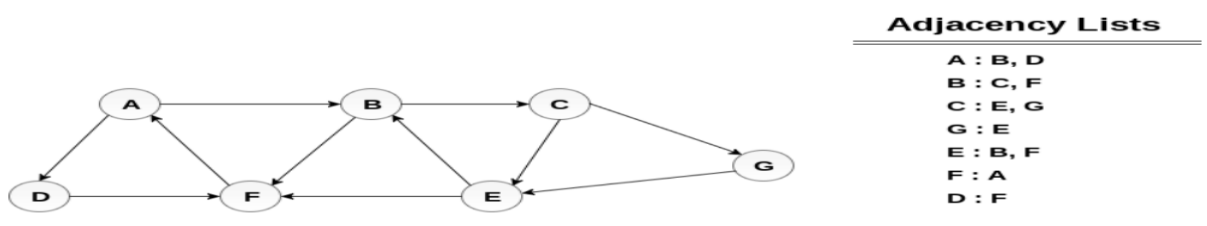
Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting <b>S</b> (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> , mark it as visited and enqueue it.

4		<p>Next, the unvisited adjacent node from <b>S</b> is <b>B</b>. We mark it as visited and enqueue it.</p>
5		<p>Next, the unvisited adjacent node from <b>S</b> is <b>C</b>. We mark it as visited and enqueue it.</p>
6		<p>Now, <b>S</b> is left with no unvisited adjacent nodes. So, we dequeue and find <b>A</b>.</p>
7		<p>From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it as visited and enqueue it.</p>

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

**Example**

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



**Solution:**

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

**Lets start examining the graph from Node A.**

1. Add A to **QUEUE1** and **NULL** to **QUEUE2**.

**QUEUE1** = {A}  
**QUEUE2** = {NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

QUEUE1 = {B, D}

QUEUE2 = {A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

QUEUE1 = {D, C, F}

QUEUE2 = {A, B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

QUEUE1 = {C, F}

QUEUE2 = {A, B, D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

QUEUE1 = {F, E, G}

QUEUE2 = {A, B, D, C}

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

QUEUE1 = {E, G}

QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.

QUEUE1 = {G}

QUEUE2 = {A, B, D, C, F, E}

## Applications of BFS Algorithm

Some of the real-life applications where a BFS algorithm implementation can be highly effective.

- **Un-weighted Graphs:** BFS algorithm can easily create the shortest path and a minimum spanning tree to visit all the vertices of the graph in the shortest time possible with high accuracy.
- **P2P Networks:** BFS can be implemented to locate all the nearest or neighboring nodes in a peer to peer network. This will find the required data faster.
- **Web Crawlers:** Search engines or web crawlers can easily build multiple levels of indexes by employing BFS. BFS implementation starts from the source, which is the web page, and then it visits all the links from that source.
- **Navigation Systems:** BFS can help find all the neighboring locations from the main or source location.
- **Network Broadcasting:** A broadcasted packet is guided by the BFS algorithm to find and reach all the nodes it has the address for.

# DFS (Depth First Search)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal

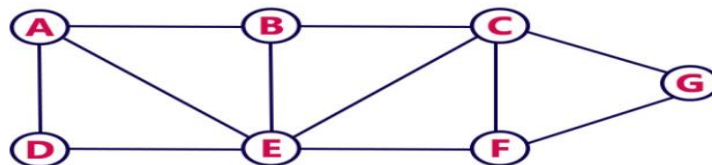
we use the following steps to implement DFS traversal...

- **Step 1** - Define a Stack of size total number of vertices in the graph.
- **Step 2** - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3** - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4** - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5** - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6** - Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7** - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

**Back tracking** is coming back to the vertex from which we reached the current vertex.

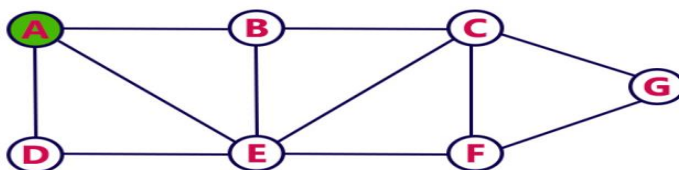
Example

Consider the following example graph to perform DFS traversal



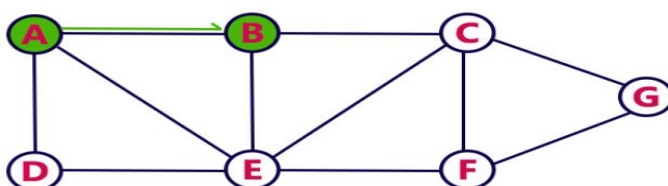
**Step 1:**

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



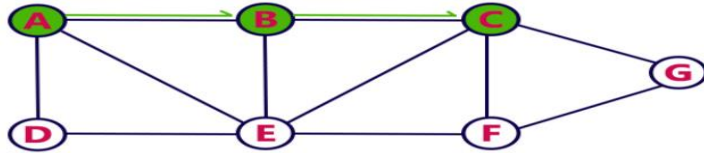
**Step 2:**

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



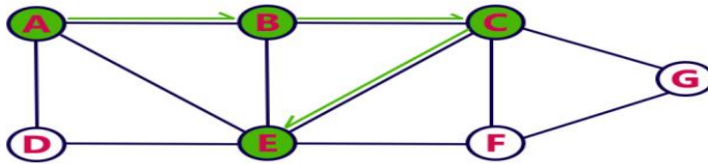
**Step 3:**

- Visit any adjacent vertex of **B** which is not visited (**C**).
- Push C on to the Stack.



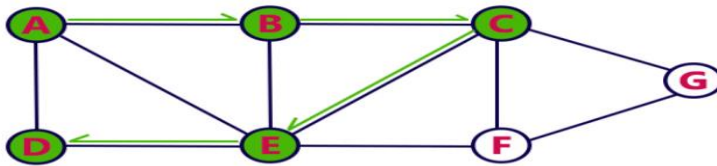
**Step 4:**

- Visit any adjacent vertex of **C** which is not visited (**E**).
- Push E on to the Stack



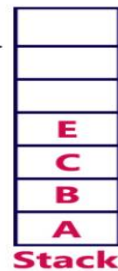
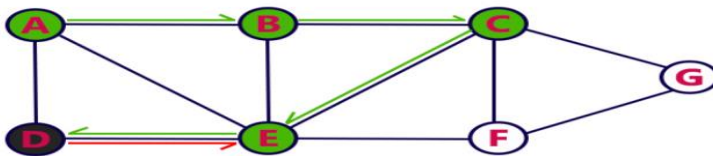
**Step 5:**

- Visit any adjacent vertex of **E** which is not visited (**D**).
- Push D on to the Stack



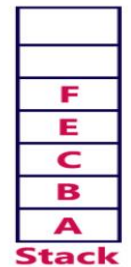
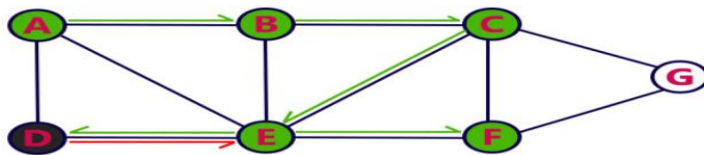
**Step 6:**

- There is no new vertex to be visited from D. So use back track.
- Pop D from the Stack.



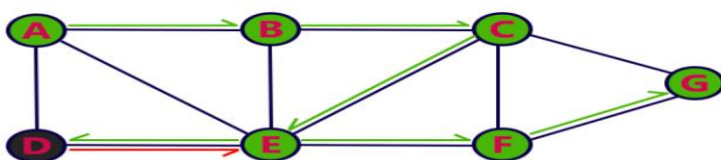
**Step 7:**

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



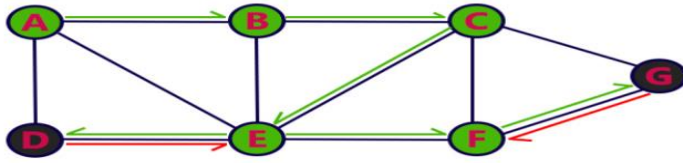
**Step 8:**

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



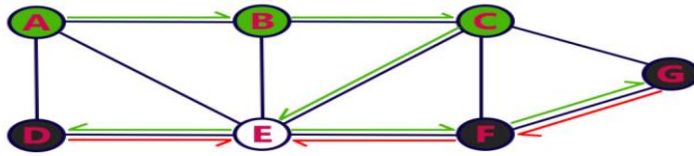
**Step 9:**

- There is no new vertex to be visited from G. So use back track.
- Pop G from the Stack.



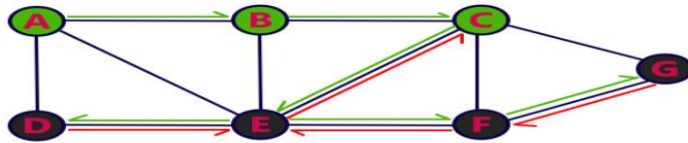
**Step 10:**

- There is no new vertex to be visited from F. So use back track.
- Pop F from the Stack.



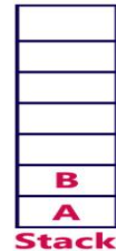
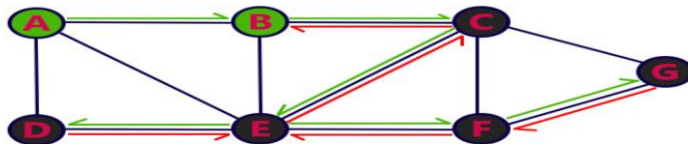
**Step 11:**

- There is no new vertex to be visited from E. So use back track.
- Pop E from the Stack.



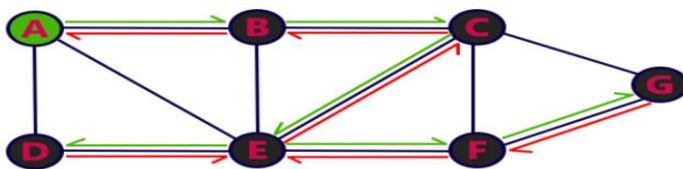
**Step 12:**

- There is no new vertex to be visited from C. So use back track.
- Pop C from the Stack.



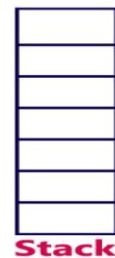
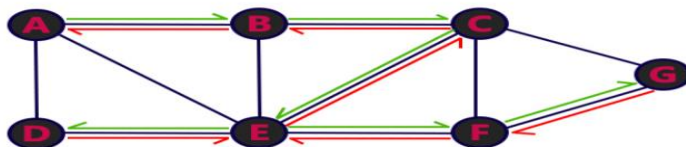
**Step 13:**

- There is no new vertex to be visited from B. So use back track.
- Pop B from the Stack.



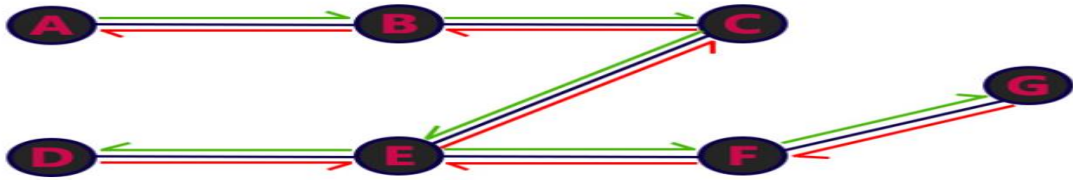
**Step 14:**

- There is no new vertex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Traversal.
- Final result of DFS traversal is following spanning tree.





## Applications of Depth-First Search Algorithm

Depth-first search is useful for:

1. Finding a path between two specified nodes, u and v, of an unweighted graph.
2. Finding a path between two specified nodes, u and v, of a weighted graph.
3. Finding whether a graph is connected or not.
4. Computing the spanning tree of a connected graph.

## SHORTEST PATH ALGORITHMS

Three different algorithms to calculate the shortest path between the vertices of a graph G. These algorithms include:

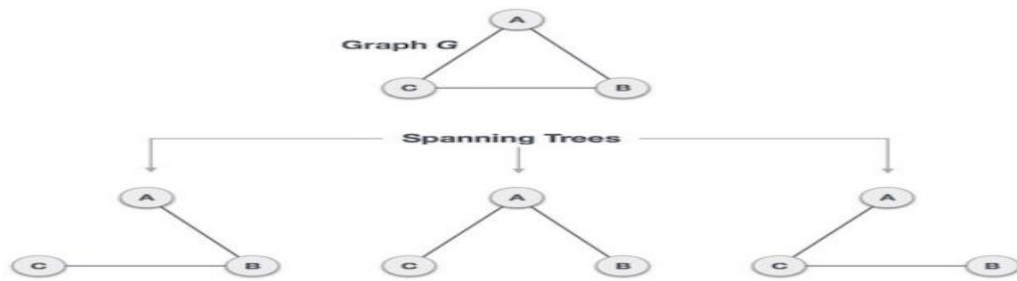
1. Minimum spanning tree
2. Dijkstra's algorithm
3. Warshall's algorithm

While the first two use an adjacency list to find the shortest path, Warshall's algorithm uses an adjacency matrix to do the same.

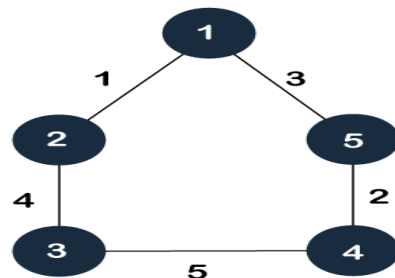
## Minimum Spanning Trees

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

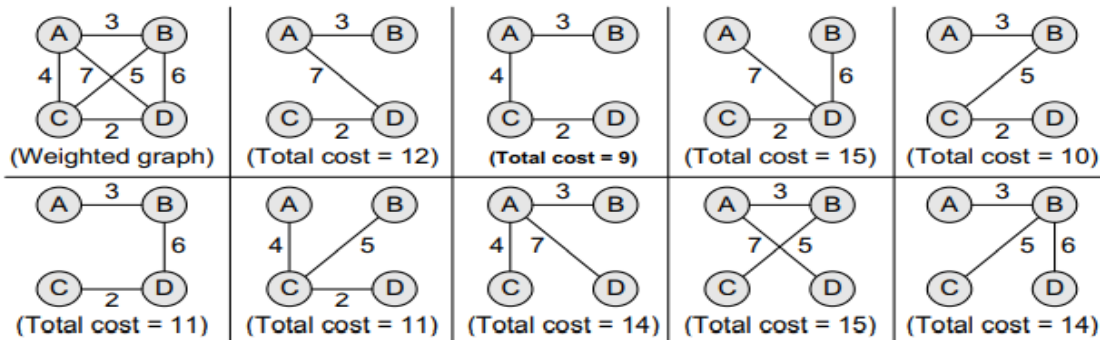
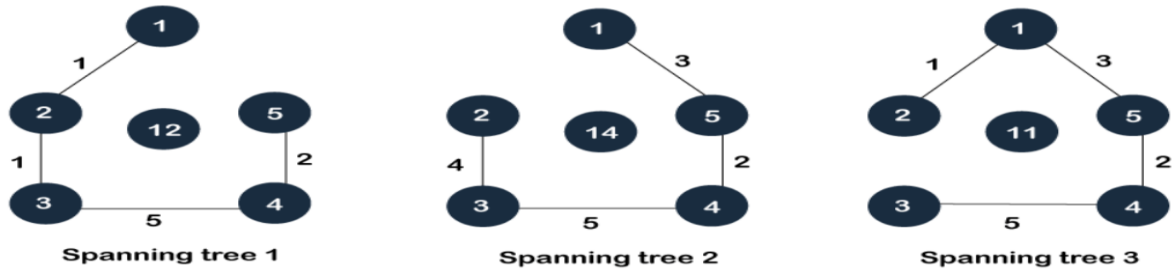
Every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum  $n^{n-2}$  number of spanning trees, where  $n$  is the number of nodes. In the above addressed example,  $n$  is 3, hence  $3^{3-2} = 3$  spanning trees are possible.







## Weighted graph and its spanning trees

### General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

### Application of Spanning Tree:

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees is –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

### **Minimum Spanning Tree (MST)**

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

### Minimum Spanning-Tree Algorithm

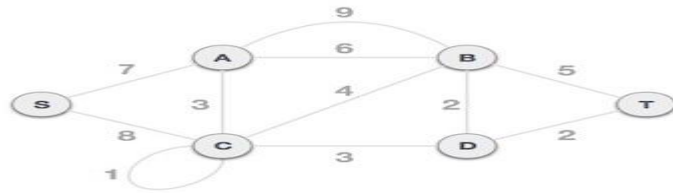
- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

## Kruskal's Minimum Spanning Tree Algorithm

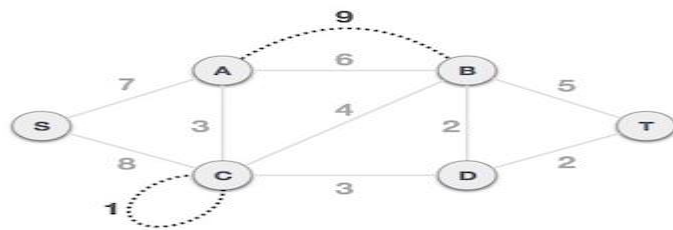
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example –

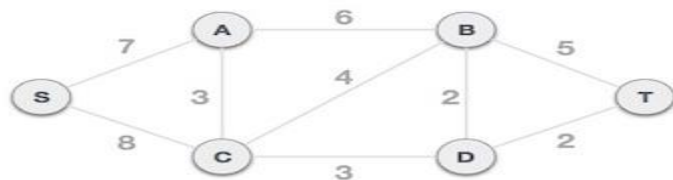


### Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



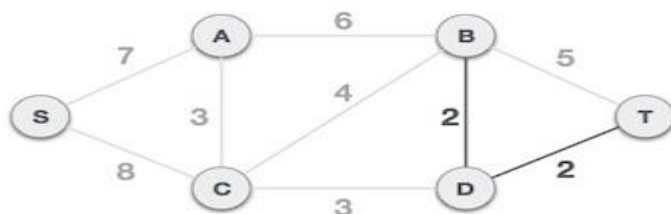
### Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

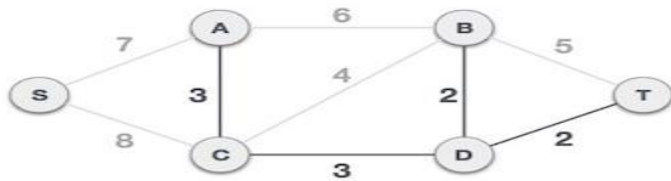
### Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

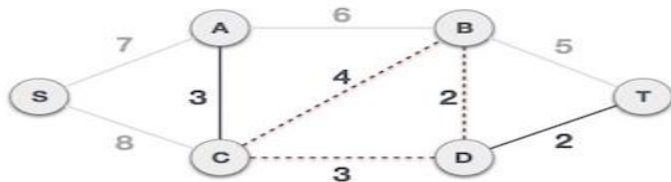


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

Next cost is 3, and associated edges are A,C and C,D. We add them again –



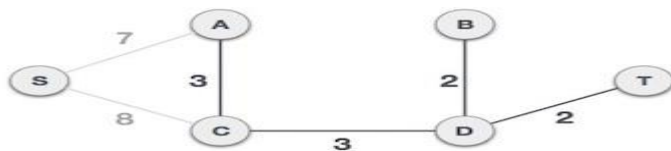
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.

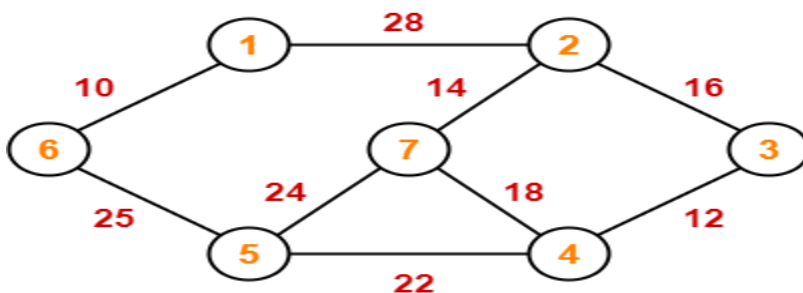


Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree

**Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm-**



### Solution-

To construct MST using Kruskal's Algorithm,

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

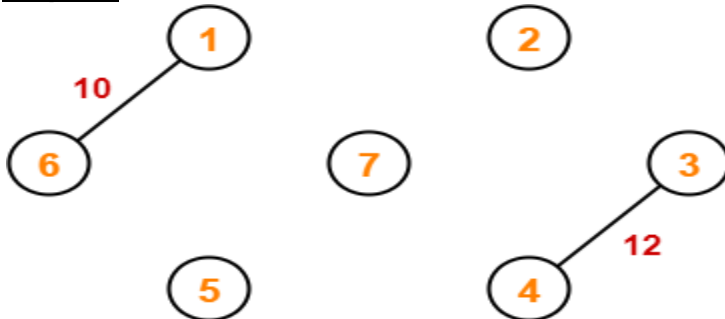
#### Step-01:



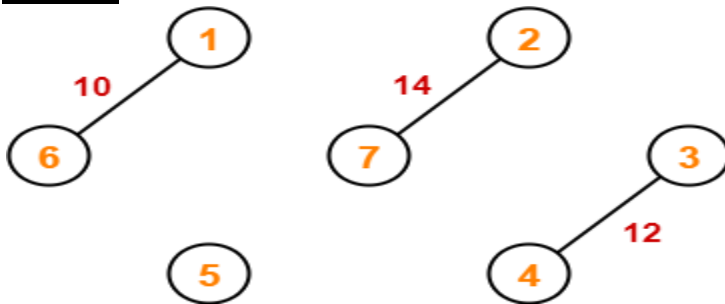
#### Step-02:



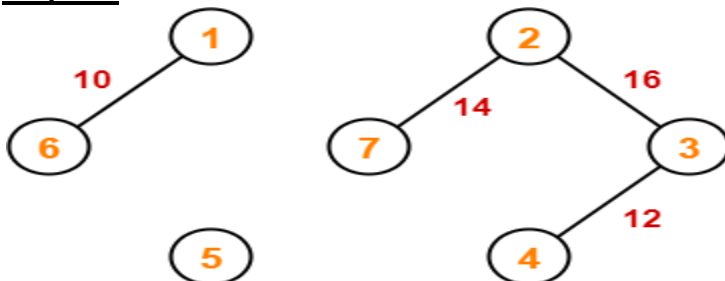
#### Step-03:



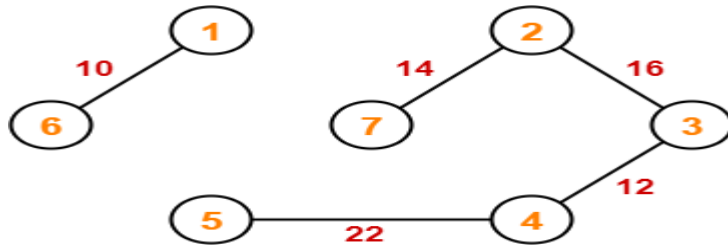
#### Step-04:



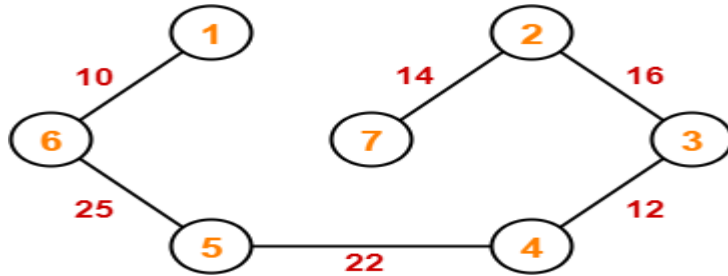
#### Step-05:



**Step-06:**



**Step-07:**



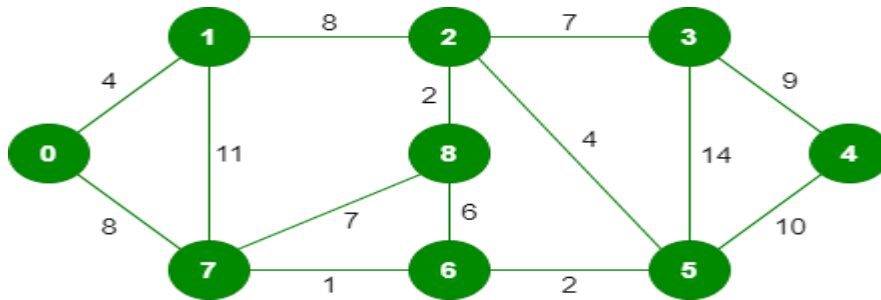
Since all the vertices have been connected / included in the MST, so we stop.

Weight of the MST

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units



**Example**

The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having  $(9 - 1) = 8$  edges.

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3

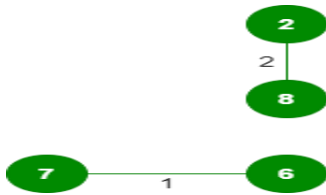
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from the sorted list of edges

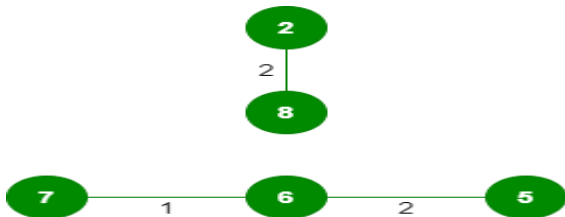
1. Pick edge 7-6: No cycle is formed, include it.



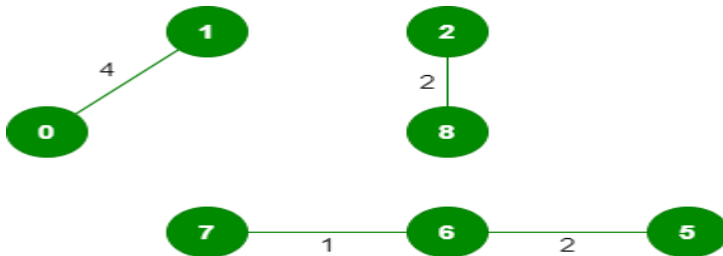
2. Pick edge 8-2: No cycle is formed, include it.



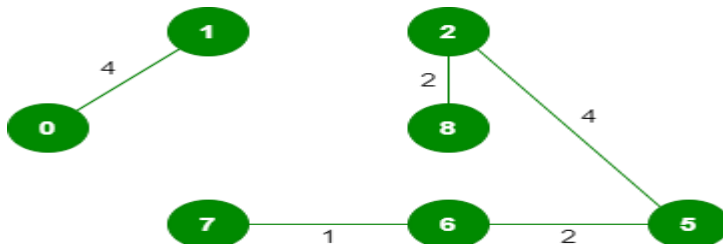
3. Pick edge 6-5: No cycle is formed, include it.



4. Pick edge 0-1: No cycle is formed, include it.

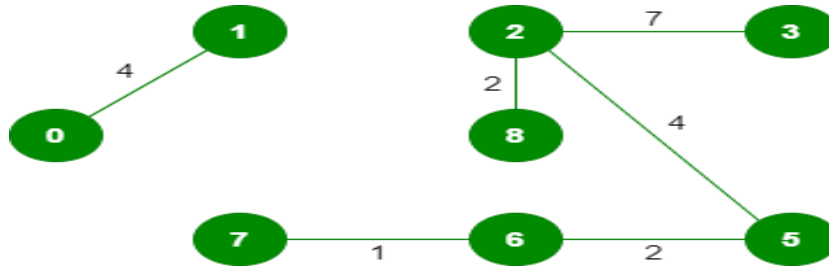


5. Pick edge 2-5: No cycle is formed, include it.



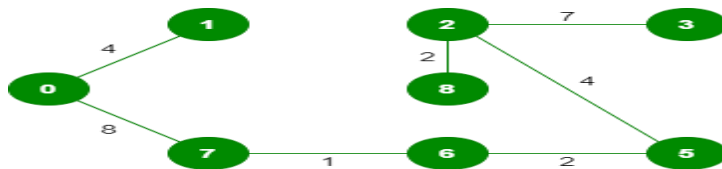
6. Pick edge 8-6: Since including this edge results in the cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



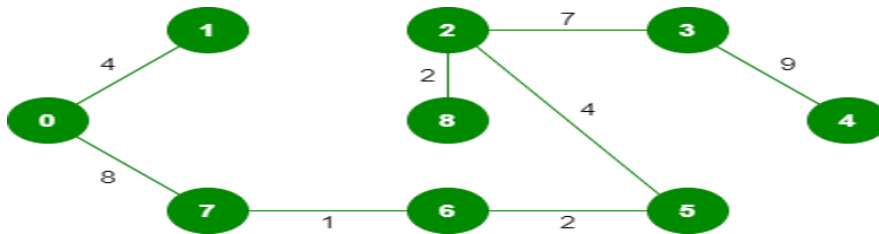
8. Pick edge 7-8: Since including this edge results in the cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in the cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals  $(V - 1)$ , the algorithm stops here.

### Prim's Algorithm-

- Prim's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Prim's algorithm, the given graph must be weighted, connected and undirected.

### Prim's Algorithm Implementation-

The implementation of Prim's Algorithm is explained in the following steps-

#### Step-01:

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.

#### Step-02:

- Find all the edges that connect the tree to new vertices.
- Find the least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

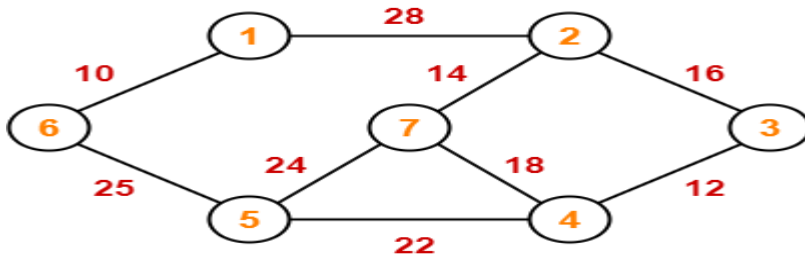
#### Step-03:

- Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.



**Problem-01:**

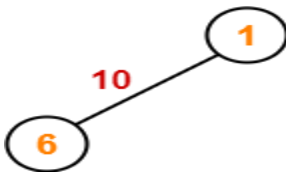
Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-



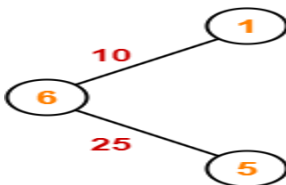
**Solution-**

The above discussed steps are followed to find the minimum cost spanning tree using Prim's Algorithm-

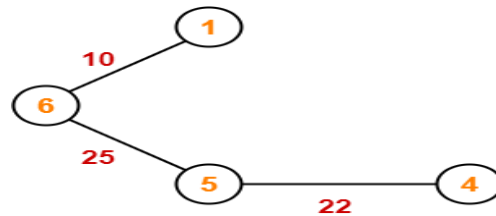
**Step-01:**



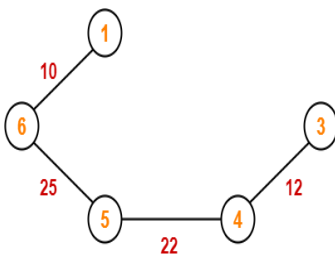
**Step-02:**



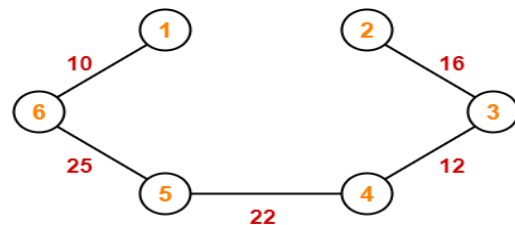
**Step-03:**



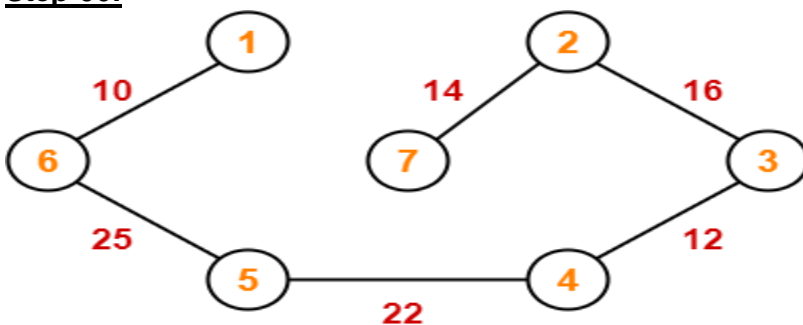
**Step-04:**



**Step-05:**



**Step-06:**



➤ Since all the vertices have been included in the MST, so we stop.

Now, Cost of Minimum Spanning Tree

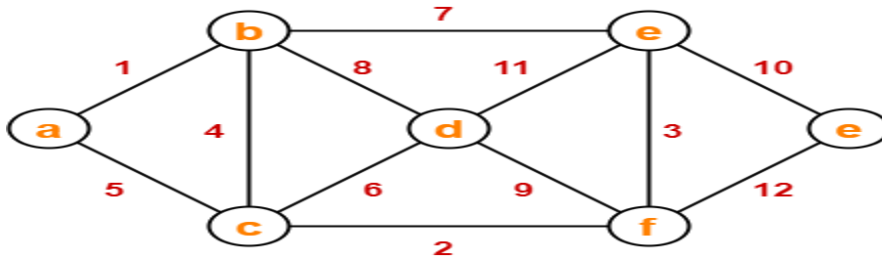
= Sum of all edge weights

=  $10 + 25 + 22 + 12 + 16 + 14$

= 99 units

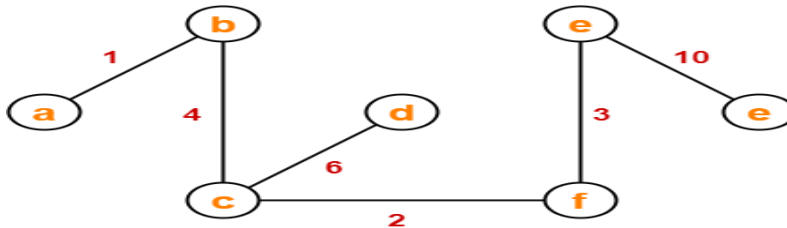
**Example 2**

Using Prim's Algorithm, find the cost of minimum spanning tree (MST) of the given graph-



**Solution-**

The minimum spanning tree obtained by the application of Prim's Algorithm on the given graph is as shown below-



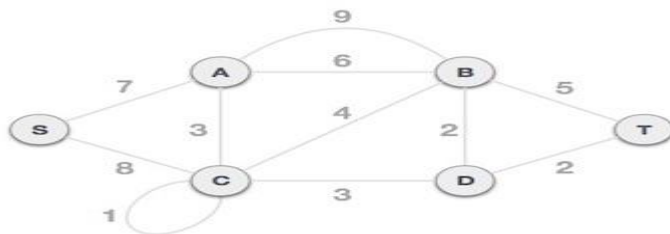
Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

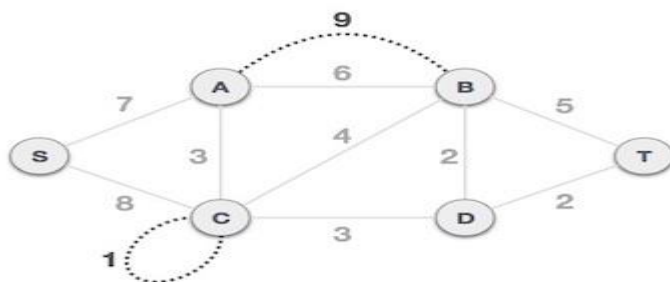
=  $1 + 4 + 2 + 6 + 3 + 10$

= 26 units

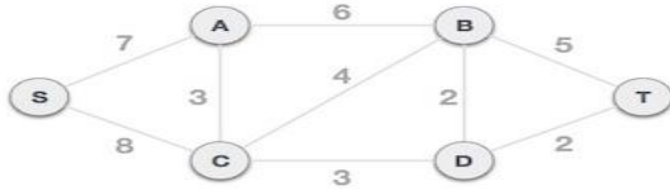
**Example:**



**Step 1 - Remove all loops and parallel edges**



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

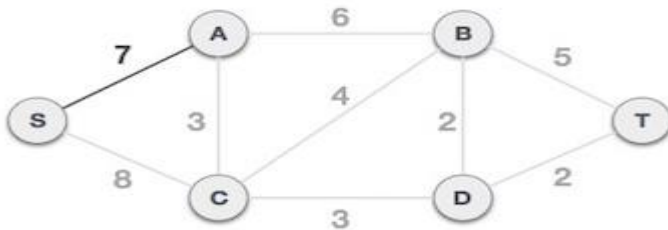


**Step 2** - Choose any arbitrary node as root node

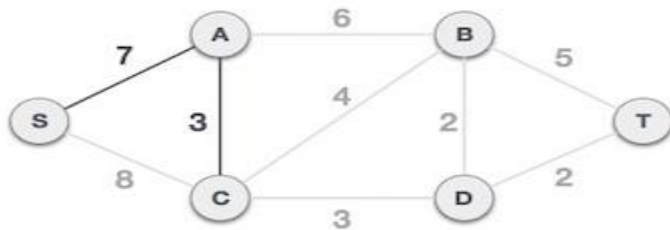
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

**Step 3** - Check outgoing edges and select the one with less cost

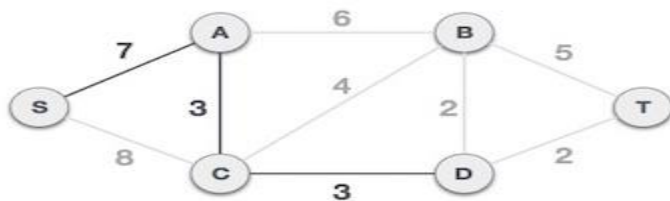
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



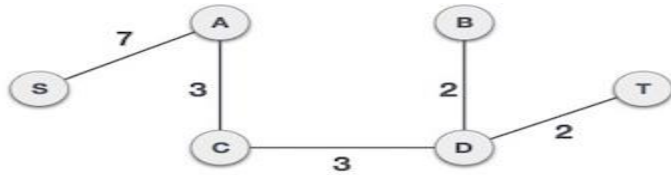
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



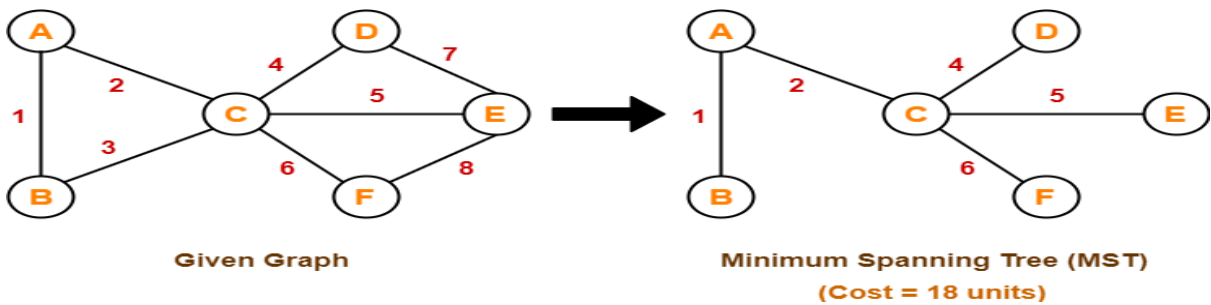
We may find that the output spanning tree of the same graph using two different algorithms is same.

### Comparison between Prim's and Kruskals

1. If all the edge weights are distinct, then both the algorithms are guaranteed to find the same MST.

#### Example-

Consider the following example-



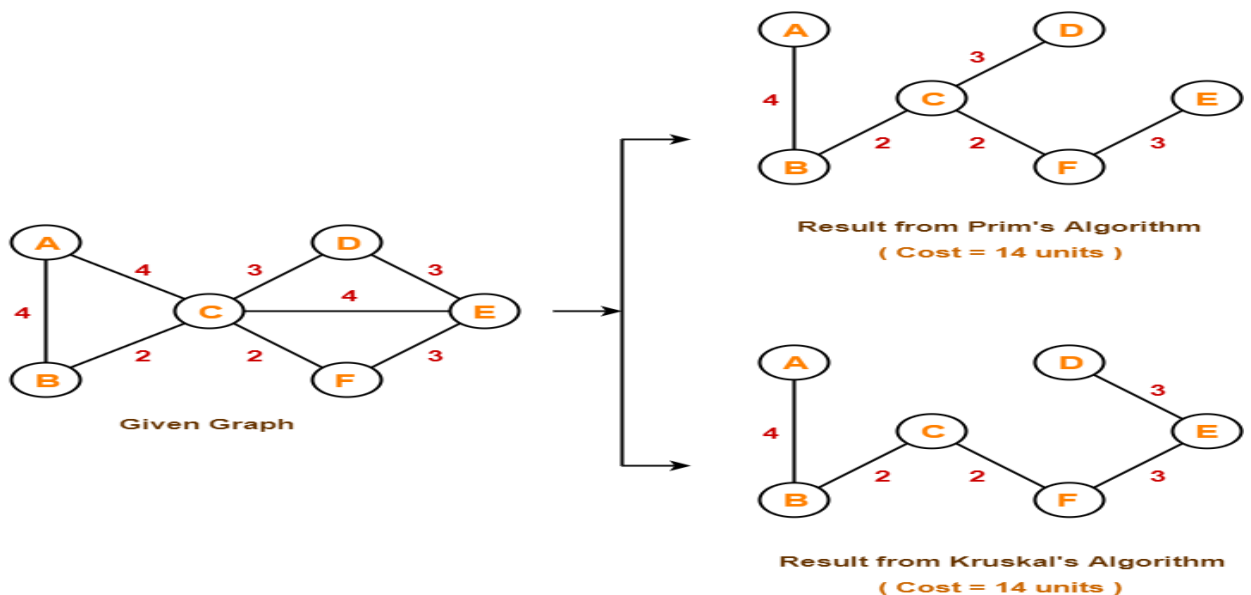
Here, both the algorithms on the above given graph produces the same MST as shown.

2. If all the edge weights are not distinct, then both the algorithms may not always produce the same MST.

- However, cost of both the MSTs would always be same in both the cases.

#### Example-

Consider the following example-



3. Kruskal's Algorithm is preferred when-

- The graph is sparse.
- There are less number of edges in the graph like  $E = O(V)$
- The edges are already sorted or can be sorted in linear time.

Prim's Algorithm is preferred when-

- The graph is dense.
- There are large number of edges in the graph like  $E = O(V^2)$ .

4. Difference between Prim's Algorithm and Kruskal's Algorithm-

<b>Prim's Algorithm</b>	<b>Kruskal's Algorithm</b>
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

### Definition

This algorithm was created and published by Dr. Edsger W. Dijkstra, a brilliant Dutch computer scientist and software engineer.

The Dijkstra's algorithm finds the shortest path from a particular node, called the source node to every other node in a connected graph. It produces a shortest path tree with the source node as the root. It is profoundly used in computer networks to generate optimal routes with the aim of minimizing routing costs.

### Basics of 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.

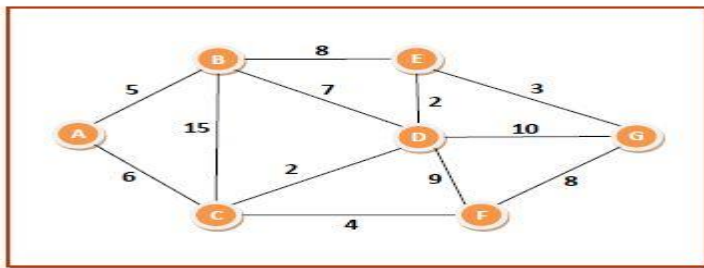
## Dijkstra's Algorithm

Input – A graph representing the network; and a source node,  $s$

Output – A shortest path tree,  $spt[]$ , with  $s$  as the root node.

1. Select the source node also called the initial node
2. Define an empty set  $N$  that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with  $0$ , and insert it into  $N$ .
4. Repeat Steps 5 to 7 until the destination node is in  $N$  or there are no more labelled nodes in  $N$ .
5. Consider each node that is not in  $N$  and is connected by an edge from the newly inserted node.
  6. (a) If the node that is not in  $N$  has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.
  - (b) Else if the node that is not in  $N$  was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)
7. Pick a node not in  $N$  that has the smallest label assigned to it and add it to  $N$ .

Example



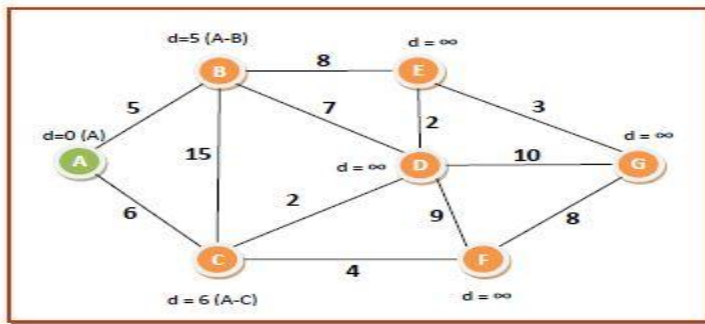
The initializations will be as follows –

- $dist[7] = \{0, \infty, \infty, \infty, \infty, \infty, \infty\}$
- $Q = \{A, B, C, D, E, F, G\}$
- $SS = \emptyset$

**Pass 1** – We choose node **A** from **Q** since it has the lowest **dist[]** value of **0** and put it in **S**. The neighbouring nodes of **A** are **B** and **C**. We update **dist[]** values corresponding to **B** and **C** according to the algorithm. So the values of the data structures become –

- $dist[7] = \{0, 5, 6, \infty, \infty, \infty, \infty\}$
- $Q = \{B, C, D, E, F, G\}$
- $S = \{A\}$

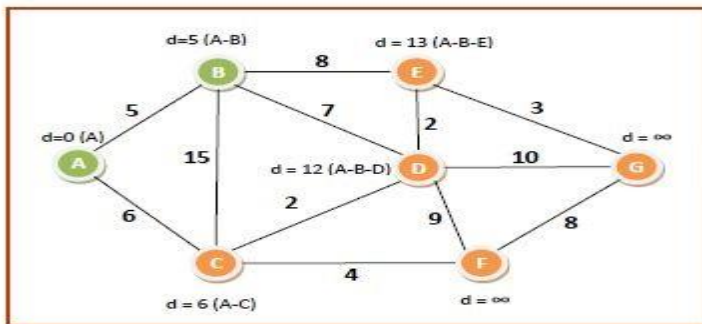
The distances and shortest paths after this pass are shown in the following graph. The green node denotes the node already added to  $S$  –



**Pass 2** – We choose node **B** from **Q** since it has the lowest **dist[]** value of **5** and put it in **S**. The neighbouring nodes of **B** are **C, D** and **E**. We update **dist[]** values corresponding to **C, D** and **E** according to the algorithm. So the values of the data structures become –

- $\text{dist}[7]=\{0,5,6,12,13,\infty,\infty\}$
- $Q=\{C,D,E,F,G\}$
- $S=\{A,B\}$

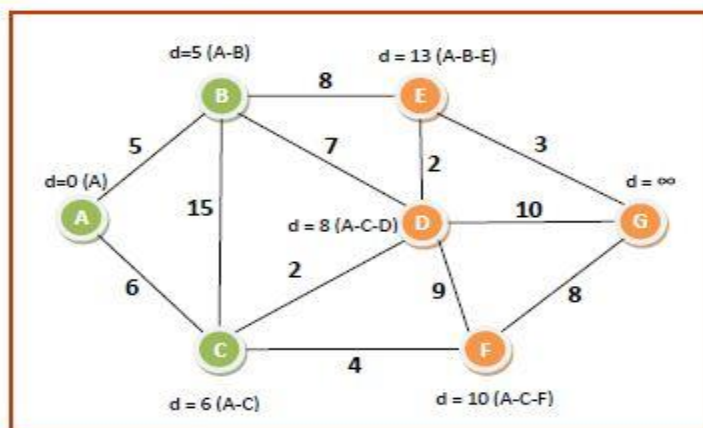
The distances and shortest paths after this pass are –



**Pass 3** – We choose node **C** from **Q** since it has the lowest **dist[]** value of **6** and put it in **S**. The neighbouring nodes of **C** are **D** and **F**. We update **dist[]** values corresponding to **D** and **F**. So the values of the data structures become –

- $\text{dist}[7]=\{0,5,6,8,13,10,\infty\}$
- $Q=\{D,E,F,G\}$
- $S=\{A,B,C\}$

The distances and shortest paths after this pass are –

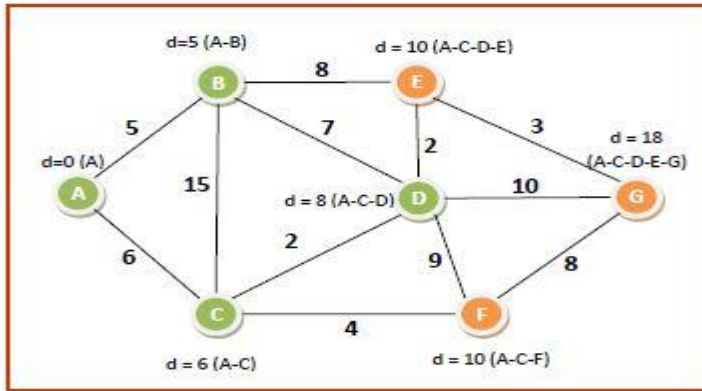




**Pass 4** – We choose node **D** from **Q** since it has the lowest **dist[]** value of 8 and put it in **S**. The neighbouring nodes of **D** are **E**, **F** and **G**. We update **dist[]** values corresponding to **E**, **F** and **G**. So the values of the data structures become –

- $\text{dist}[7]=\{0,5,6,8,10,10,18\}$
- $Q=\{E,F,G\}$
- $S=\{A,B,C,D\}$

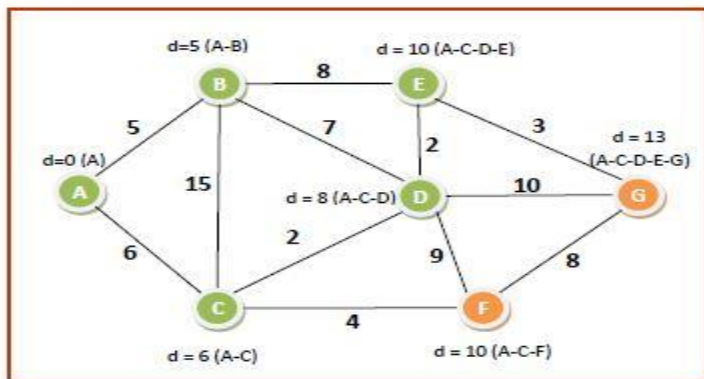
The distances and shortest paths after this pass are –



**Pass 5** – We can choose either node **E** or node **F** from **Q** since both of them have the lowest **dist[]** value of **10**. We select any one of them, say **E**, and put it in **S**. The neighbouring nodes of **D** is **G**. We update **dist[]** values corresponding to **G**. So the values of the data structures become –

- $\text{dist}[7]=\{0,5,6,8,10,10,13\}$
- $Q=\{F,G\}$
- $S=\{A,B,C,D,E\}$

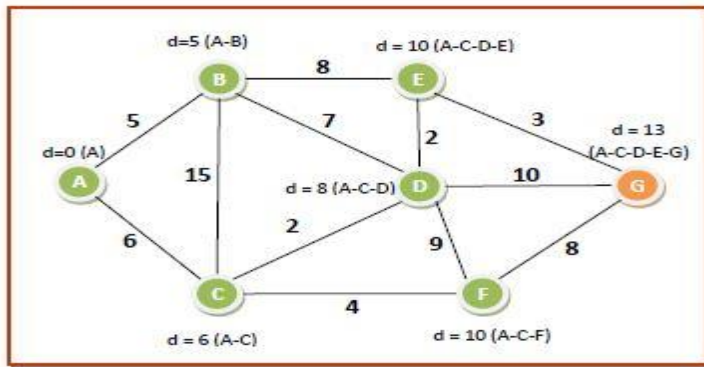
The distances and shortest paths after this pass are –



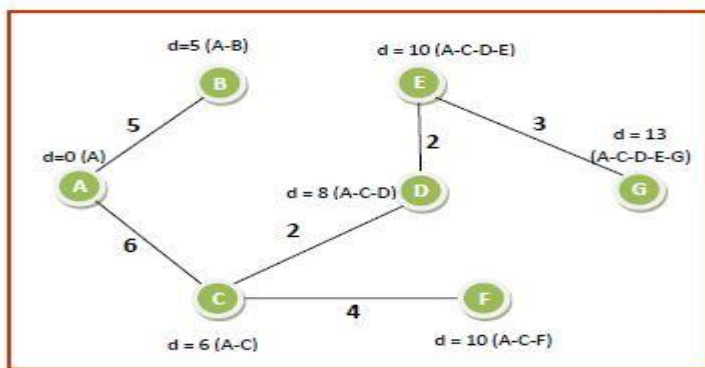
**Pass 6** – We choose node **F** from **Q** since it has the lowest **dist[]** value of **10** and put it in **S**. The neighbouring nodes of **F** is **G**. The **dist[]** value corresponding to **G** is less than that through **F**. So it remains same. The values of the data structures become –

- $\text{dist}[7]=\{0,5,6,8,10,10,13\}$
- $Q=\{G\}$
- $S=\{A,B,C,D,E,F\}$

The distances and shortest paths after this pass are –



**Pass 7** – There is just one node in **Q**. We remove it from **Q** put it in **S**. The `dist[]` array needs no change. Now, **Q** becomes empty, **S** contains all the nodes and so we come to the end of the algorithm. We eliminate all the edges or routes that are not in the path of any route. So the shortest path tree from source node A to all other nodes are as follows –



## Transitive Closure of a Directed Graph

A transitive closure of a graph is constructed to answer reachability questions

### Definition

For a directed graph  $G = (V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, the transitive closure of  $G$  is a graph  $G^* = (V, E^*)$ . In  $G^*$ , for every vertex pair  $v, w$  in  $V$  there is an edge  $(v, w)$  in  $E^*$  if and only if there is a valid path from  $v$  to  $w$  in  $G$ .

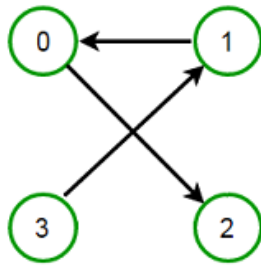
**Where and Why is it Needed?** Finding the transitive closure of a directed graph is an important problem in the following computational tasks

- Transitive closure is used to find the reachability analysis of transition networks representing distributed and parallel systems.
- It is used in the construction of parsing automata in compiler construction
- Recently, transitive closure computation is being used to evaluate recursive database queries

### Algorithm

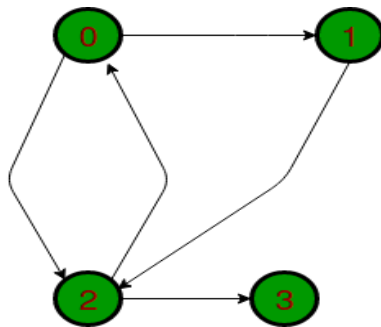
In order to determine the transitive closure of a graph, we define a matrix  $t$  where  $t_{ij} = 1$ , for  $i, j, k = 1, 2, 3, \dots, n$  if there exists a path in  $G$  from the vertex  $i$  to vertex  $j$  with intermediate

vertices in the set  $(1, 2, 3, \dots, k)$  and 0 otherwise. That is,  $G^*$  is constructed by adding an edge  $(i, j)$  into  $E^*$  if and only if  $t_k ij = 1$



Its connectivity matrix  $C$  is

	0	1	2	3
0	1	0	1	0
1	1	1	1	0
2	0	0	1	0
3	1	1	1	1



Transitive closure of graphs is

1	1	1	1
1	1	1	1
1	1	1	1
0	0	0	1

### Floyd Warshall Algorithm-

- Floyd Warshall Algorithm is a famous algorithm.
- It is used to solve All Pairs Shortest Path Problem.
- It computes the shortest path between every pair of vertices of the given graph.
- Floyd Warshall Algorithm is an example of dynamic programming approach.

#### Advantages-

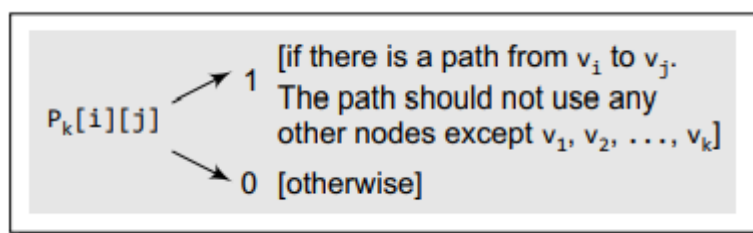
Floyd Warshall Algorithm has the following main advantages-

- It is extremely simple.
- It is easy to implement.

#### When Floyd Warshall Algorithm Is Used?

Floyd Warshall Algorithm is best suited for dense graphs.

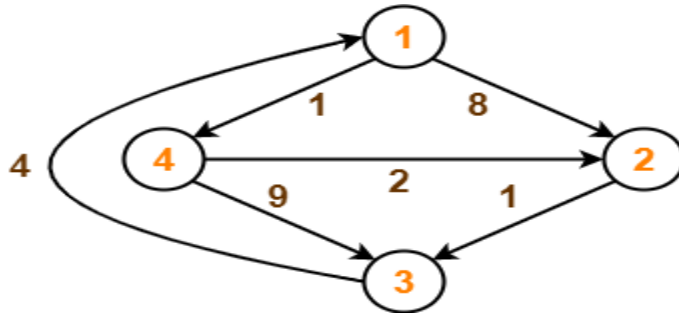
- This is because its complexity depends only on the number of vertices in the given graph.
- For sparse graphs, Johnson's Algorithm is more suitable.



Path matrix entry

**Problem-**

Consider the following directed weighted graph-



Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.

**Solution-**

Step-01:

- Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self edges nor parallel edges.

Step-02:

- Write the initial distance matrix.
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value =  $\infty$ .

Initial distance matrix for the given graph is-

$$D_0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{matrix}$$

Step-03: Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{matrix}$$

$$D_2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{matrix}$$

$$D_3 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{matrix}$$

$$D_4 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{matrix}$$

The last matrix  $D_4$  represents the shortest path distance between every pair of vertices.