Data Structures

IT 205

Dr. Manish Khare

Lecture – 24,25
29-Mar-2018

Graph

Introduction

- Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is defined as follows...
  - **Graph is a collection of nodes and edges which connects nodes in the graph**

- Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

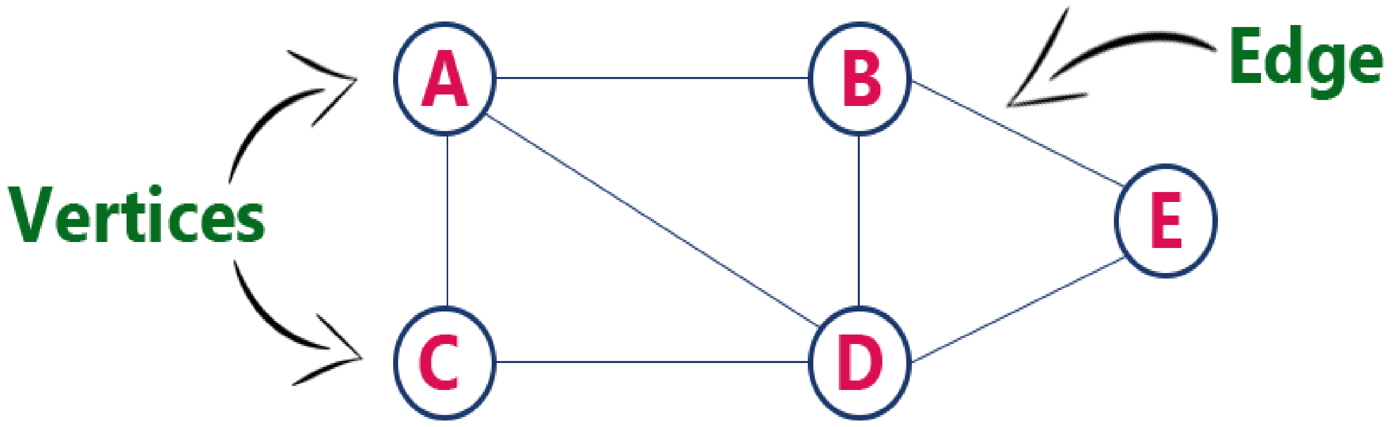
Introduction

➢ Example

The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology

- **Vertex**
  - A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Graph Terminology

➢ Edge

- An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

➢ **Undirected Edge -** An undirected edge is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

➢ **Directed Edge -** A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

➢ **Weighted Edge -** A weighted edge is an edge with cost on it.

Graph Terminology

- **Undirected Graph**
  - A graph with only undirected edges is said to be undirected graph.
- **Directed Graph**
  - A graph with only directed edges is said to be directed graph.
- **Mixed Graph**
  - A graph with undirected and directed edges is said to be mixed graph.
- **End vertices or Endpoints**
  - The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Graph Terminology

- **Origin**
  - If an edge is directed, its first endpoint is said to be origin of it.
- **Destination**
  - If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.
- **Adjacent**
  - If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.
- **Incident**
  - An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Graph Terminology

- **Outgoing Edge**
  - A directed edge is said to be outgoing edge on its origin vertex.
- **Incoming Edge**
  - A directed edge is said to be incoming edge on its destination vertex.
- **Degree**
  - Total number of edges connected to a vertex is said to be degree of that vertex.
- **Indegree**
  - Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
- **Outdegree**
  - Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Graph Terminology

- **Parallel edges or Multiple edges**
  - If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.
- **Self-loop**
  - An edge (undirected or directed) is a self-loop if its two endpoints coincide.
- **Simple Graph**
  - A graph is said to be simple if there are no parallel and self-loop edges.
- **Path**
  - A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Graph Terminology

- ***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.

Graph Terminology

- ***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.

Complete Graph

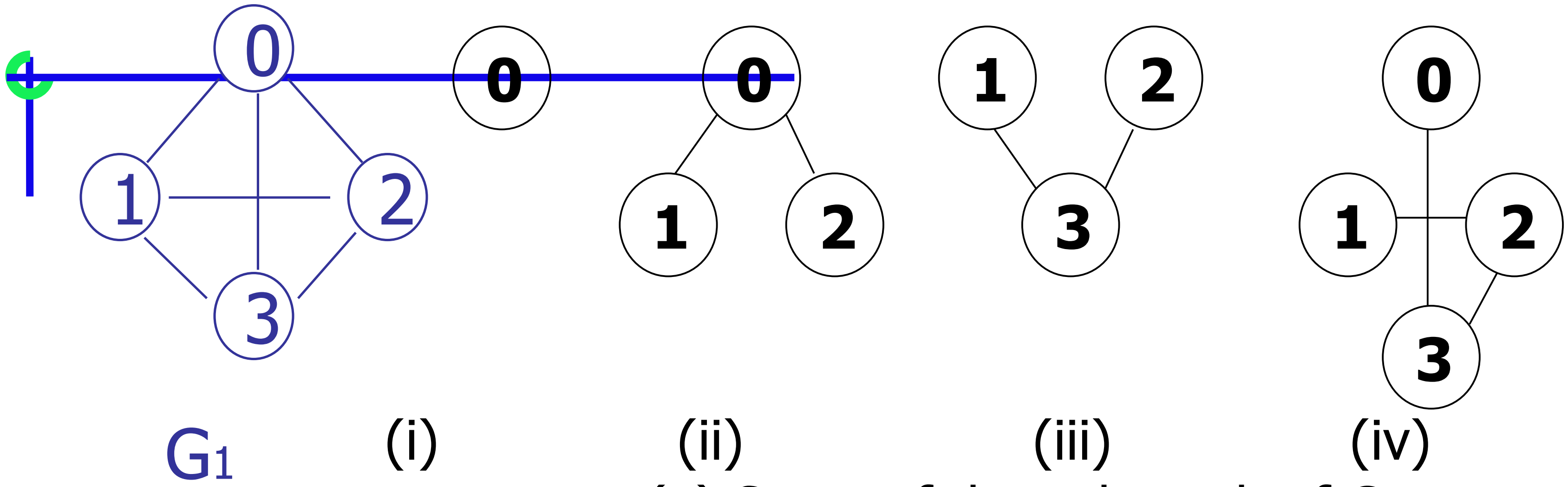
- A complete graph is a graph that has the maximum number of edges
  - for undirected graph with n vertices, the maximum number of edges is n(n-1)/2
  - for directed graph with n vertices, the maximum number of edges is n(n-1)
  - example: G1 is a complete graph

Subgraph and Path

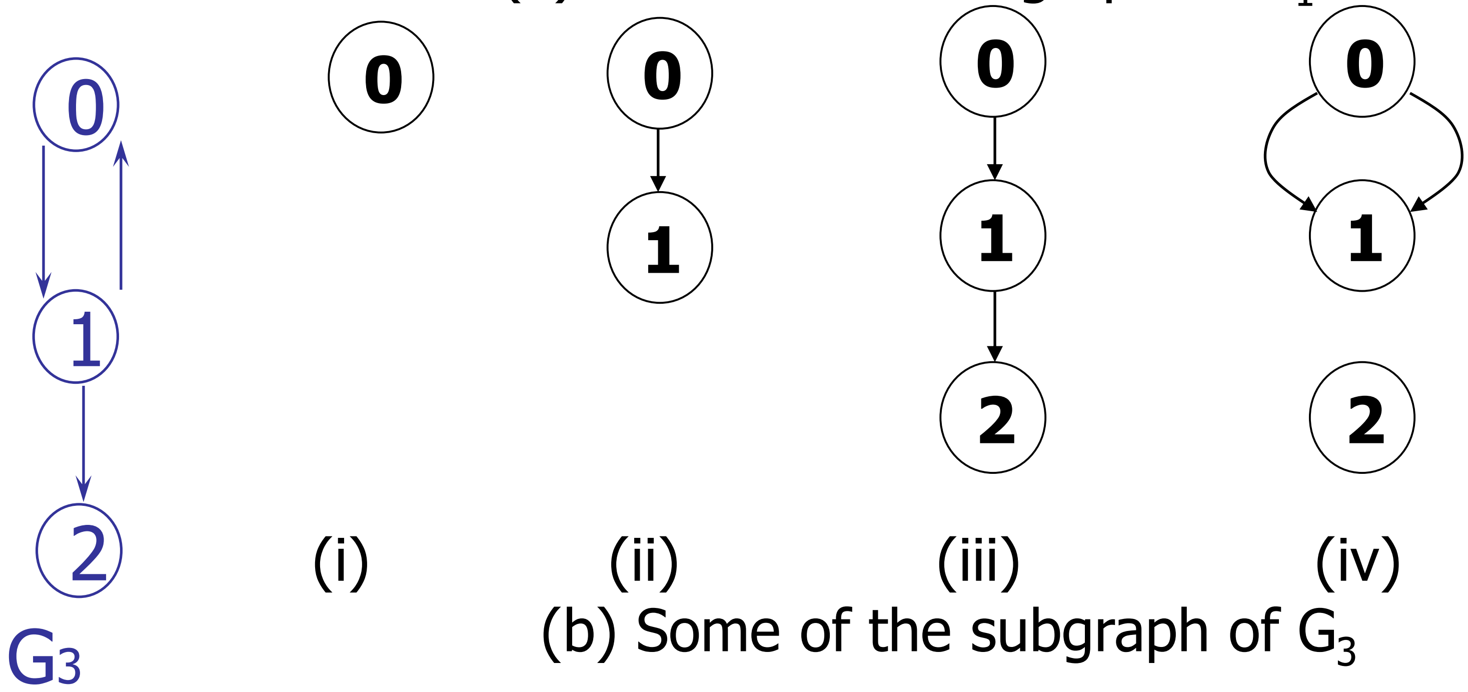
- A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G).

- A path from vertex vp to vertex vq in a graph G, is a sequence of vertices, vp, vi_1, vi_2, ..., vi_n, vq, such that (vp, vi_1), (vi_1, vi_2), ..., (vi_n, vq) are edges in an undirected graph

- The length of a path is the number of edges on it.



(a) Some of the subgraph of G_1



(b) Some of the subgraph of G_3

Degree

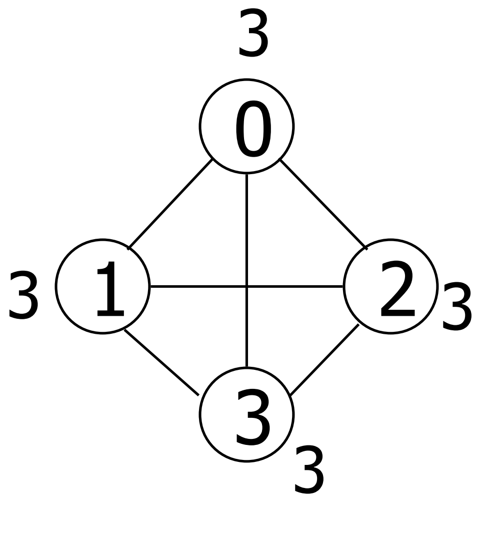
- The degree of a vertex is the number of edges incident to that vertex

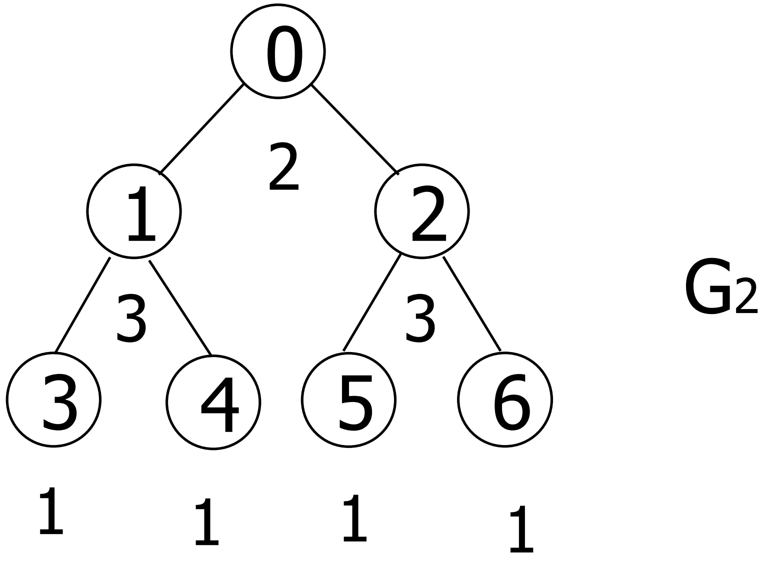
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = (\sum_{0}^{n-1} d_i)/2$$

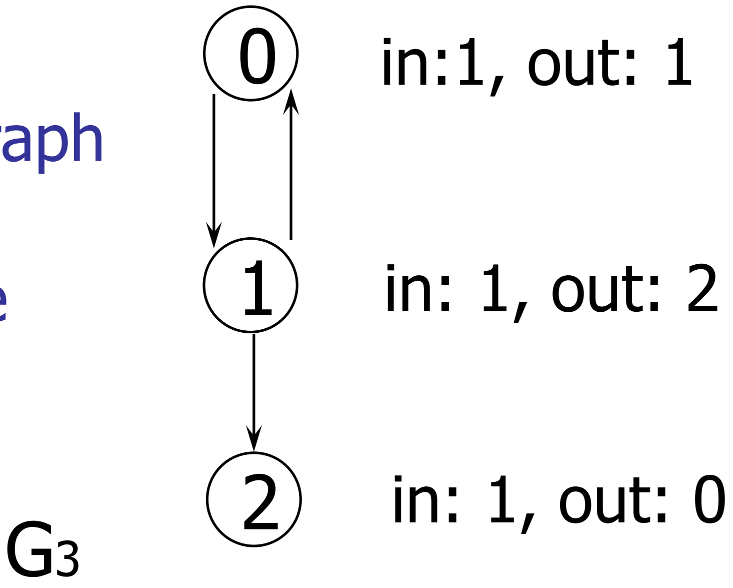
undirected graph

degree





directed graph
in-degree
out-degree



Graph Operations

structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* $\in$ *Graph*, *v*, v_1 and $v_2 \in$ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no incident edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

Graph Representations

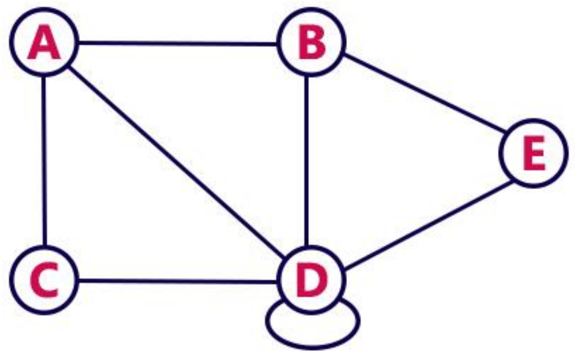
- Graph data structure is represented using following representations...
  - **Adjacency Matrix**
    - **Incidence Matrix**
  - **Adjacency List**
  - **Adjacency multi-list**

Adjacency Matrix

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

Adjacency Matrix

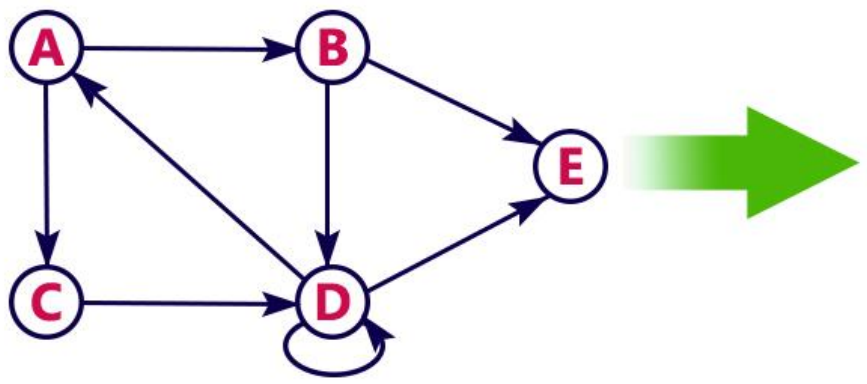
- For example, consider the following undirected graph representation...



$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Adjacency Matrix

- Directed graph representation...



$$
\begin{array}{c|ccccc|}
 & A & B & C & D & E \\
A & 0 & 1 & 1 & 0 & 0 \\
B & 0 & 0 & 0 & 1 & 1 \\
C & 0 & 0 & 0 & 1 & 0 \\
D & 1 & 0 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0
\end{array}
$$

Adjacency Matrix

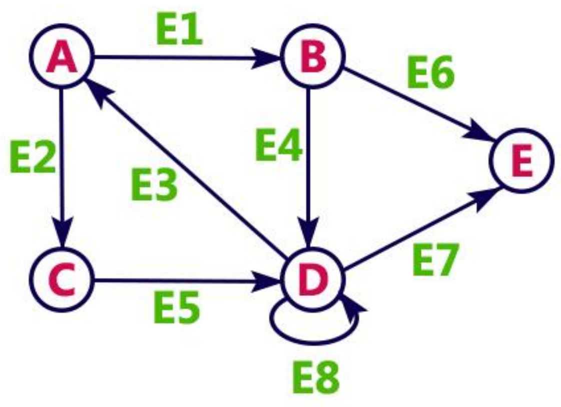
- From the above examples, we can draw the following conclusions:
  - For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
  - The adjacency matrix of an undirected graph is symmetric.
  - The memory use of an adjacency matrix is O(n2), where n is the number of nodes in the graph.
  - Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
  - The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Incidence Matrix

- In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges.
- That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4 x 6 class.
- In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1.
- Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

Incidence Matrix

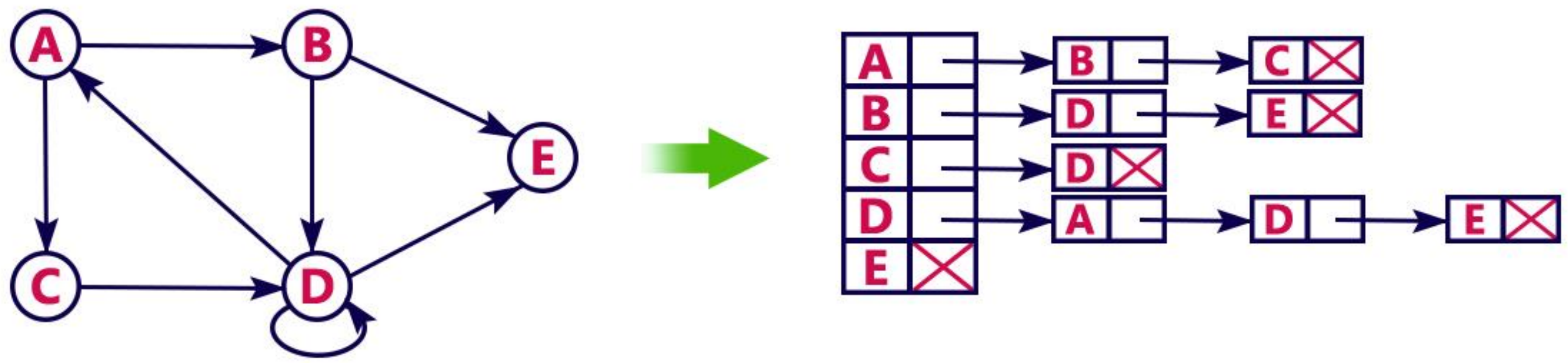
- For example, consider the following directed graph representation...



$$
\begin{array}{c|cccccccc}
 & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\
\hline
A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\
D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0
\end{array}
$$

Adjacency List

- In this representation, every vertex of graph contains list of its adjacent vertices.
- For example, consider the following directed graph representation implemented using linked list...



Adjacency Multi-list Representation

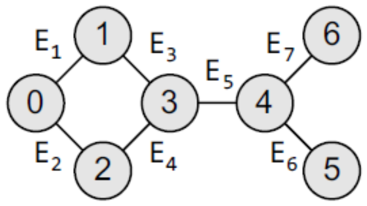
- Graphs can also be represented using multi-lists which can be said to be modified version of adjacency lists. Adjacency multi-list is an edge-based rather than a vertex-based representation of graphs.
- A multi-list representation basically consists of two parts - a directory of node's information and a set of linked lists storing information about edges.
- While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge).
- For example, the directory entry for node *i* points to the adjacency list for node *i*. This means that the nodes are shared among several lists.

Adjacency Multi-list Representation

- In a multi-list representation, the information about an edge (v_i, v_j) of an undirected graph can be stored using the following attributes:
  - M: A single bit field to indicate whether the edge has been examined or not.
  - v_i: A vertex in the graph that is connected to vertex v_j by an edge.
  - v_j: A vertex in the graph that is connected to vertex v_i by an edge.
  - Link i for v_i: A link that points to another node that has an edge incident on vi.
  - Link j for v_i: A link that points to another node that has an edge incident on vj.

marked	vertex1	vertex2	link1	link2

Adjacency Multi-list Representation



Edge 1		0	1	Edge 2	Edge 3
Edge 2		0	2	NULL	Edge 4
Edge 3		1	3	NULL	Edge 4
Edge 4		2	3	NULL	Edge 5
Edge 5		3	4	NULL	Edge 6
Edge 6		4	5	Edge 7	NULL
Edge 7		4	6	NULL	NULL

Adjacency Multi-list Representation

- Using the adjacency multi-list given above, the adjacency list for vertices can be constructed as shown below

VERTEX	LIST OF EDGES
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7

Graph Traversals

- Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit 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 vertices of graph without getting into looping path.

- There are two graph traversal techniques and they are as follows...
  - **DFS (Depth First Search)**
  - **BFS (Breadth First Search)**

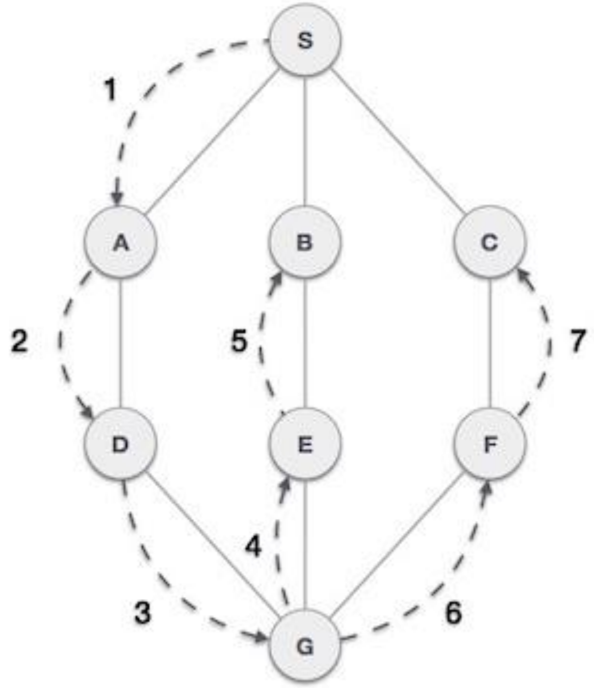
DFS (Depth First Search)

- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

- DFS can be more than one solution for one problem.

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

DFS (Depth First Search)



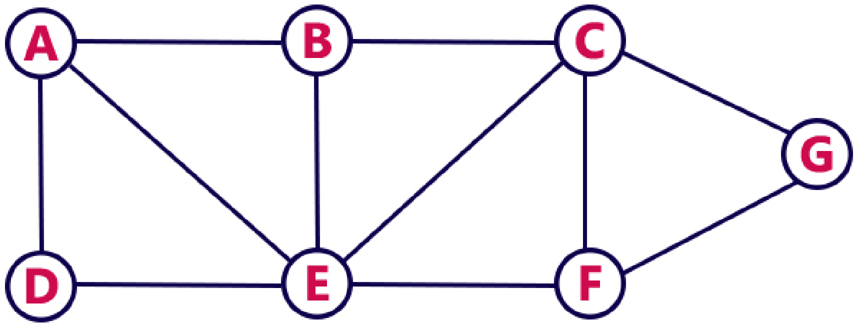
- As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

DFS (Depth First Search)

- 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 **adjacent** vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
  - **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
  - **Step 5:** When there is no new vertex to be 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

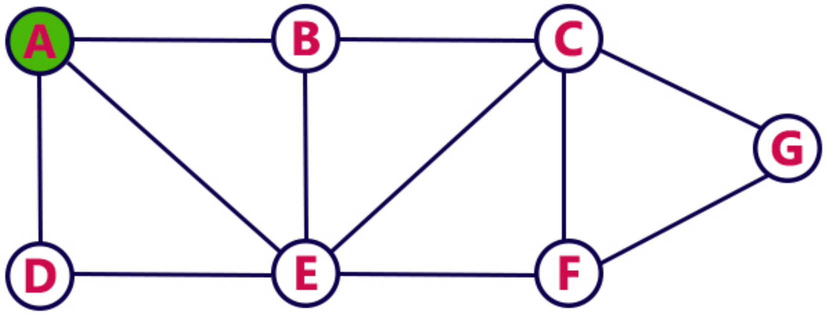
DFS (Depth First Search)

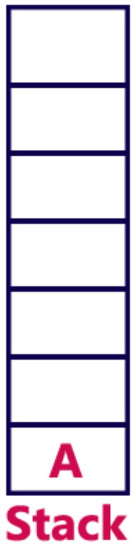
Consider the following example graph to perform DFS traversal



Step 1:

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

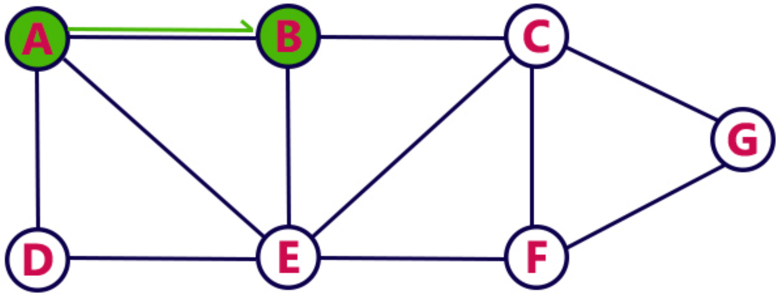


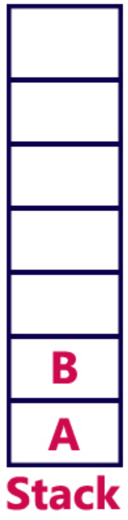


DFS (Depth First Search)

Step 2:

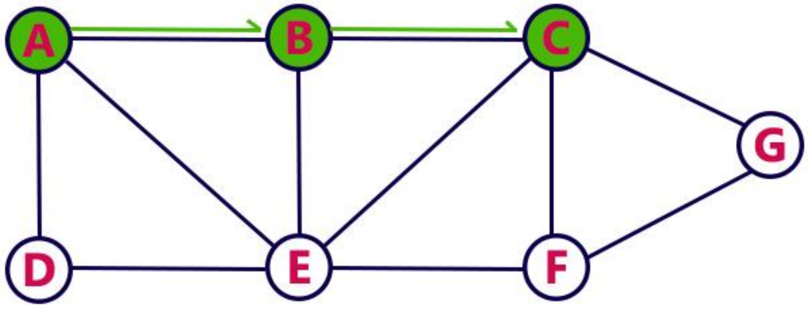
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

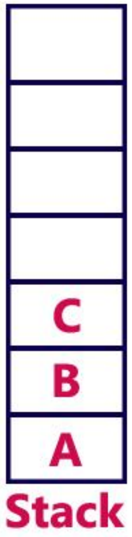




Step 3:

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

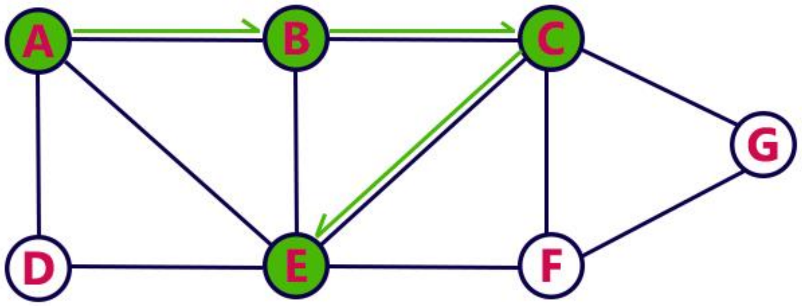


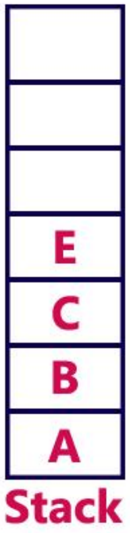


DFS (Depth First Search)

Step 4:

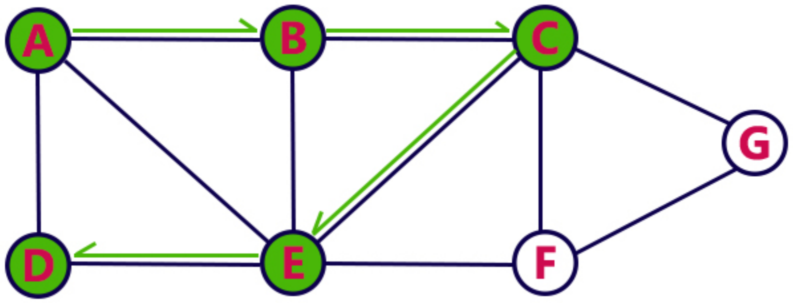
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

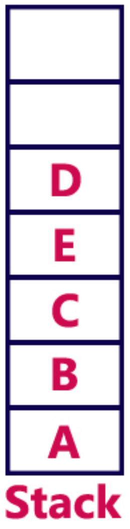




Step 5:

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

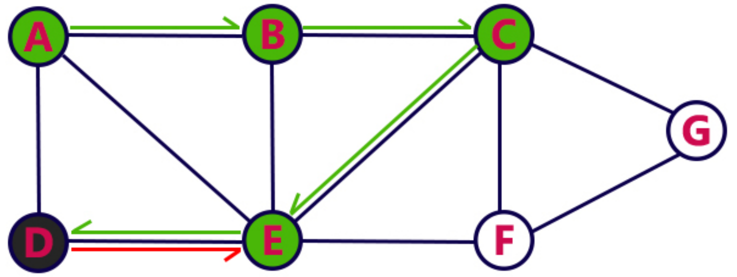


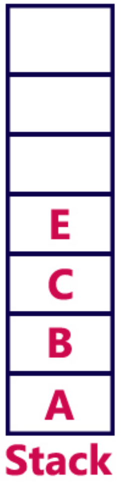


DFS (Depth First Search)

Step 6:

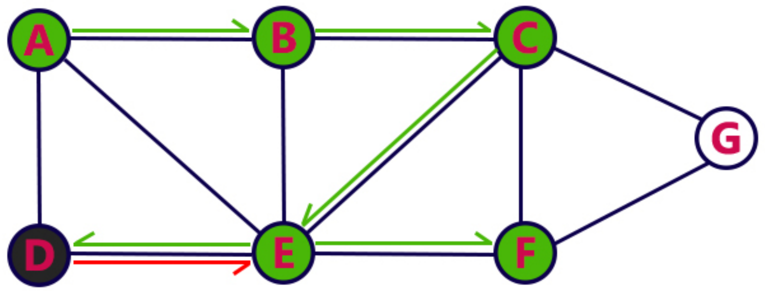
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

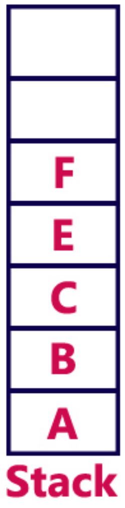




Step 7:

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

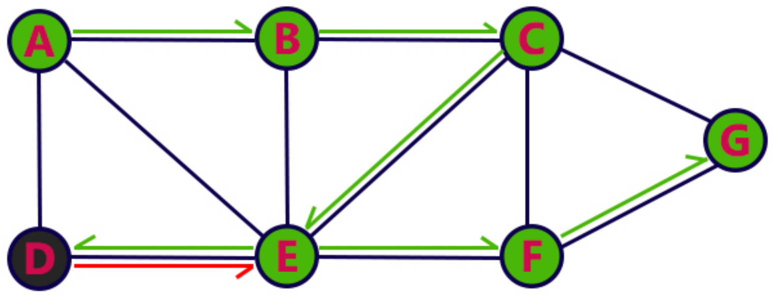


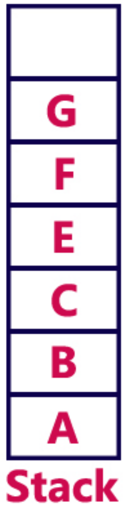


DFS (Depth First Search)

Step 8:

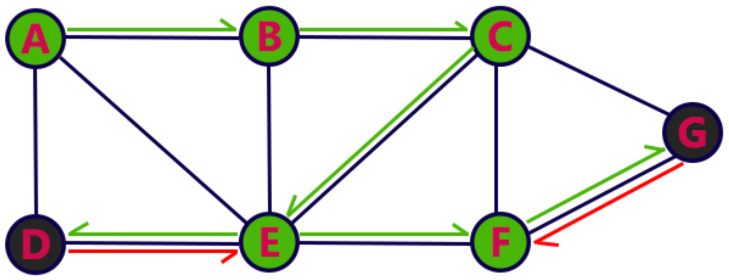
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

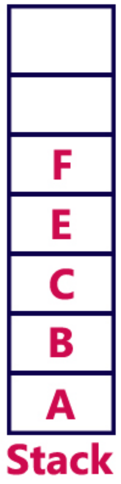




Step 9:

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

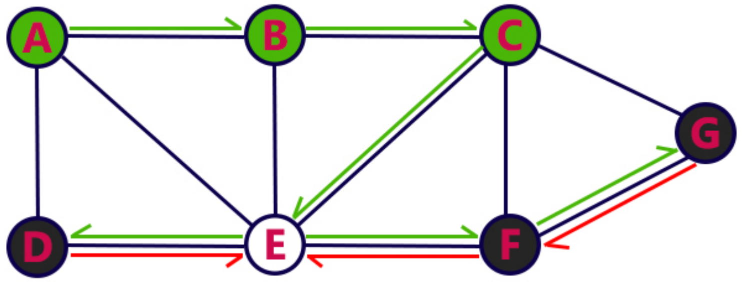


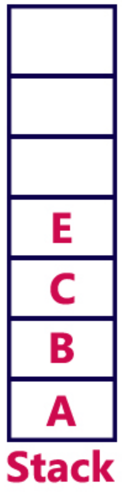


DFS (Depth First Search)

Step 10:

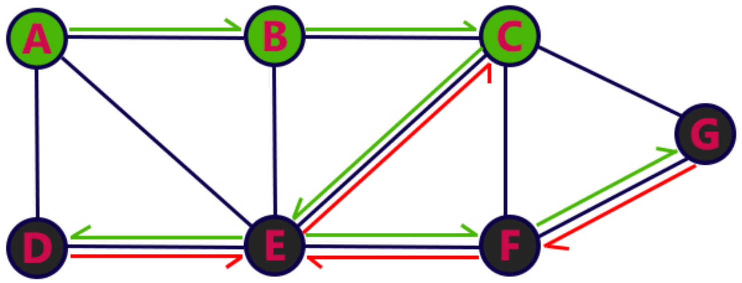
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

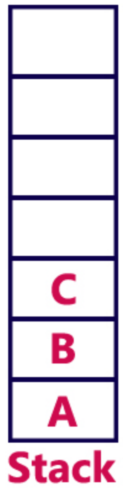




Step 11:

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

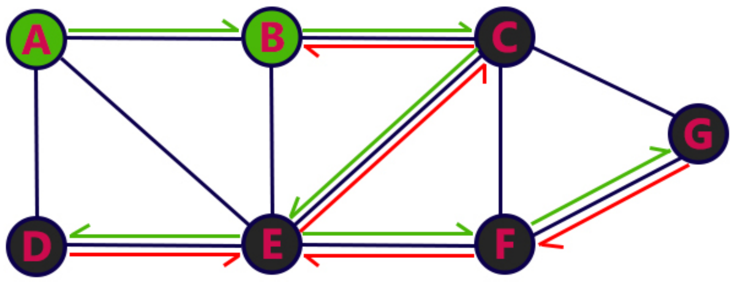


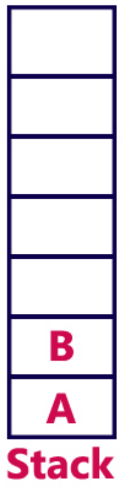


DFS (Depth First Search)

Step 12:

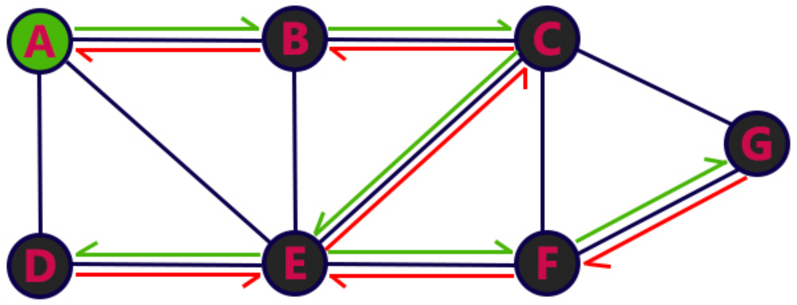
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

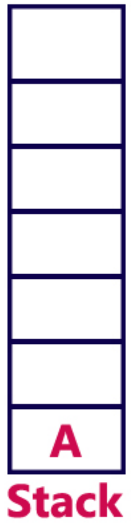




Step 13:

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

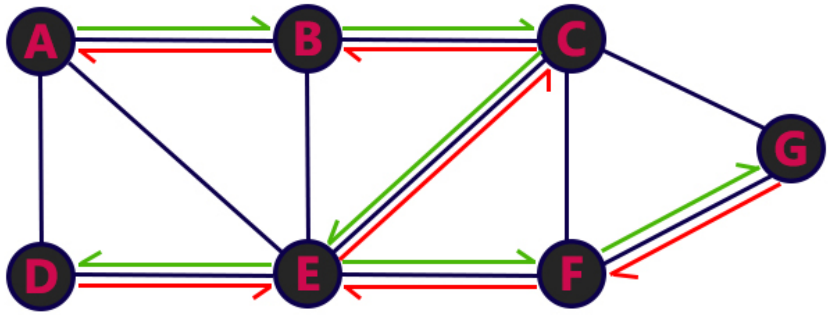


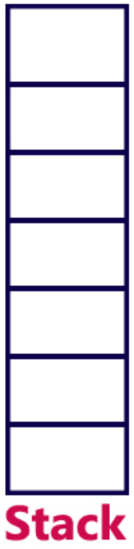


DFS (Depth First Search)

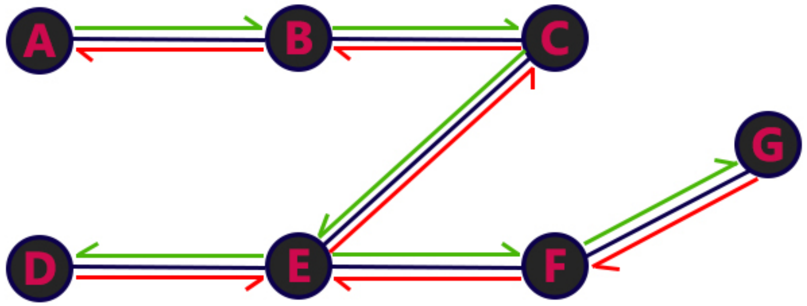
Step 14:

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

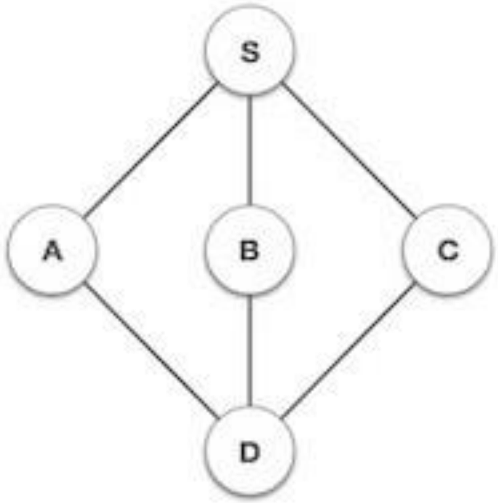




- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



DFS (Depth First Search)

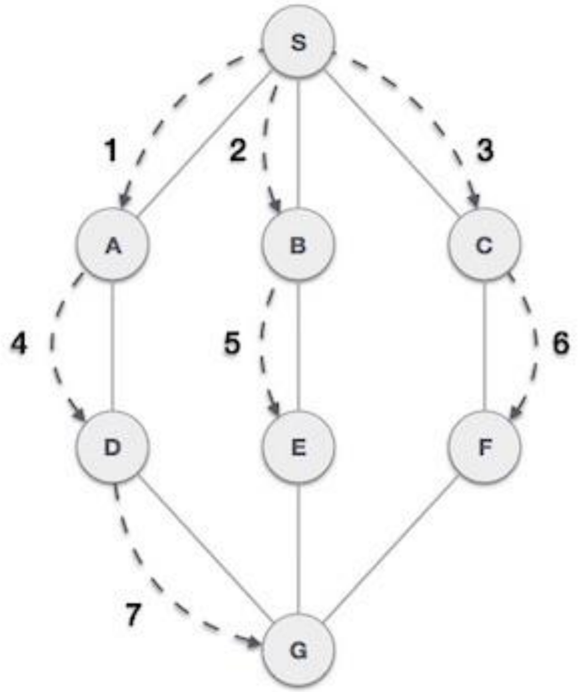


BFS (Breadth First Search)

- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

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

BFS (Breadth First Search)



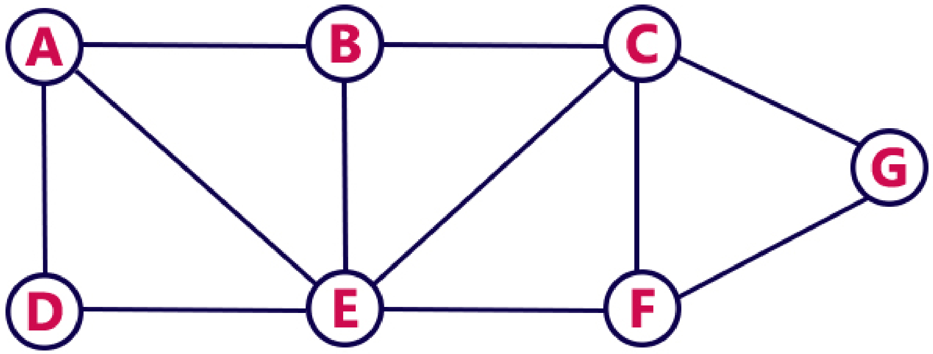
- As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

BFS (Breadth First Search)

- 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 **adjacent** vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
  - **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
  - **Step 5:** Repeat step 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

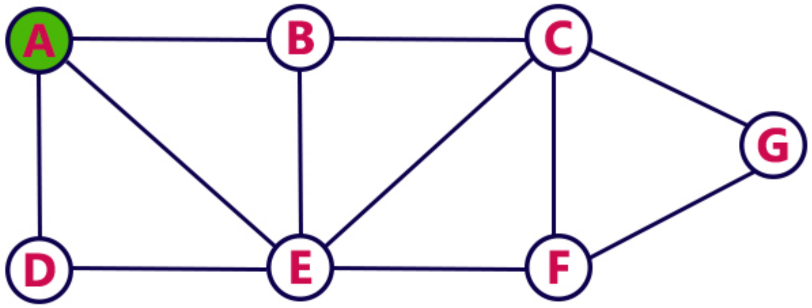
BFS (Breadth First Search)

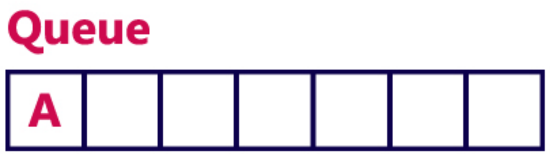
Consider the following example graph to perform BFS traversal



Step 1:

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

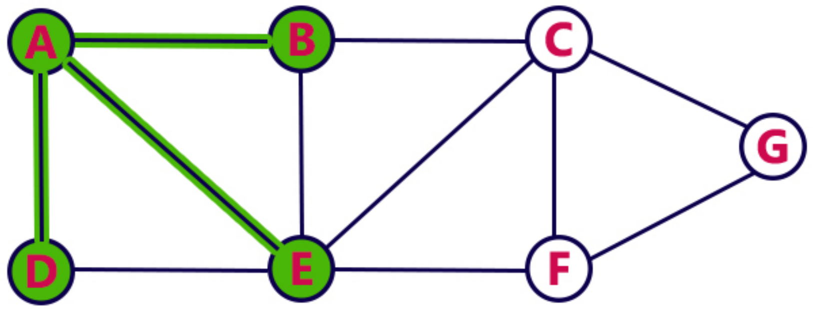




BFS (Breadth First Search)

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..

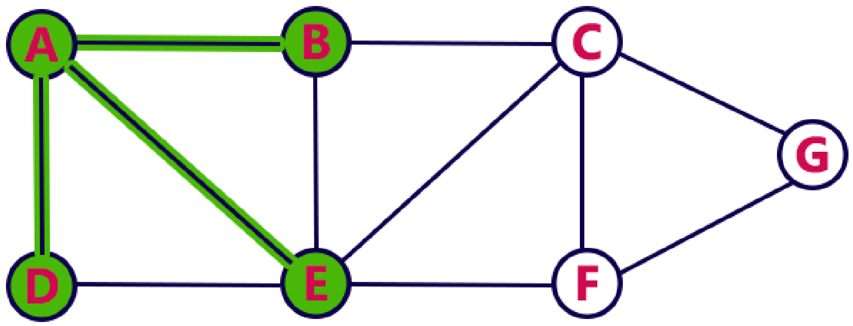


Queue

	D	E	B			

Step 3:

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



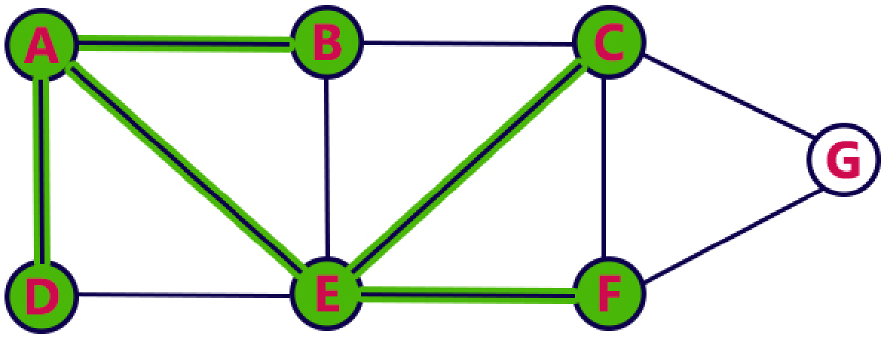
Queue

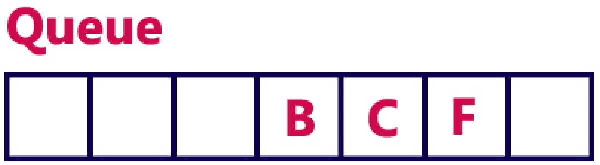
		E	B			

BFS (Breadth First Search)

Step 4:

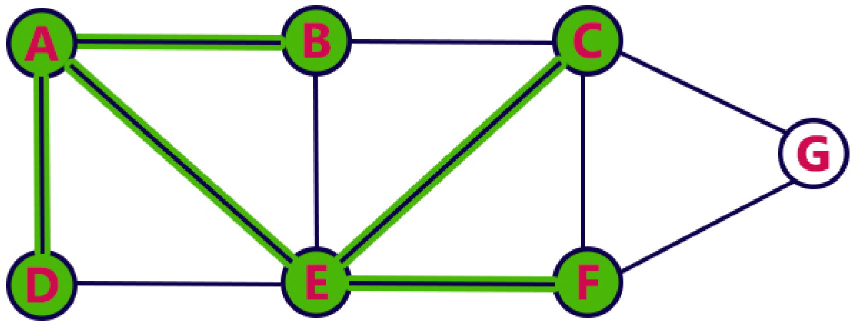
- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

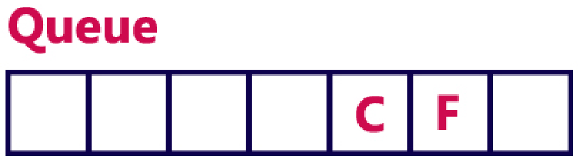




Step 5:

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

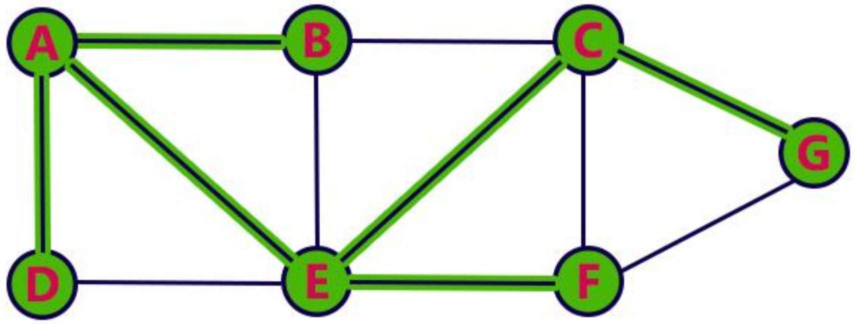


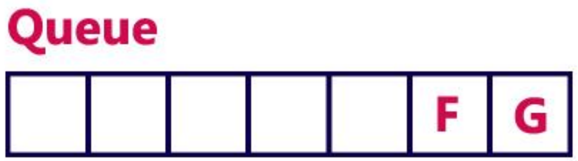


BFS (Breadth First Search)

Step 6:

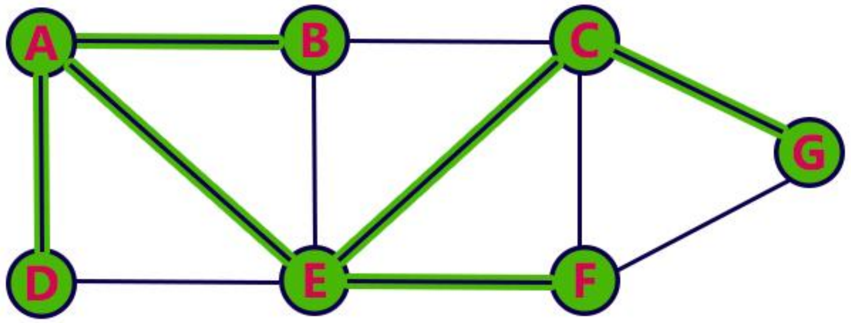
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

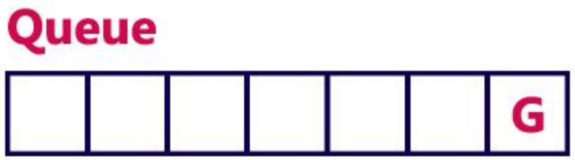




Step 7:

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

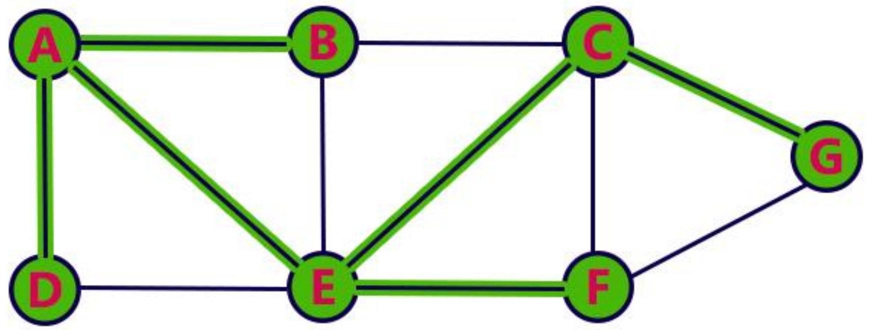




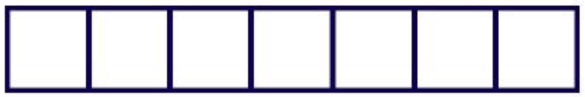
BFS (Breadth First Search)

Step 8:

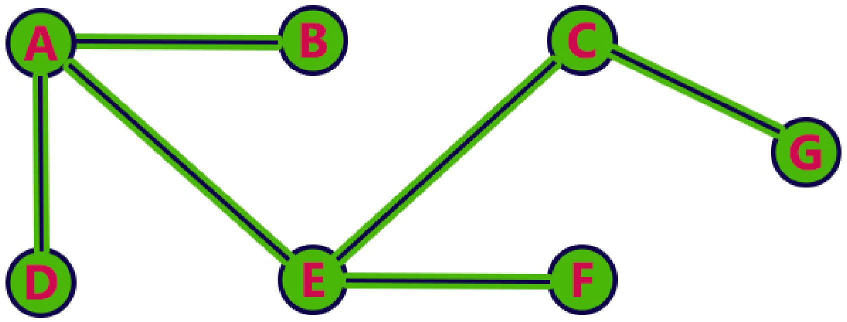
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



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

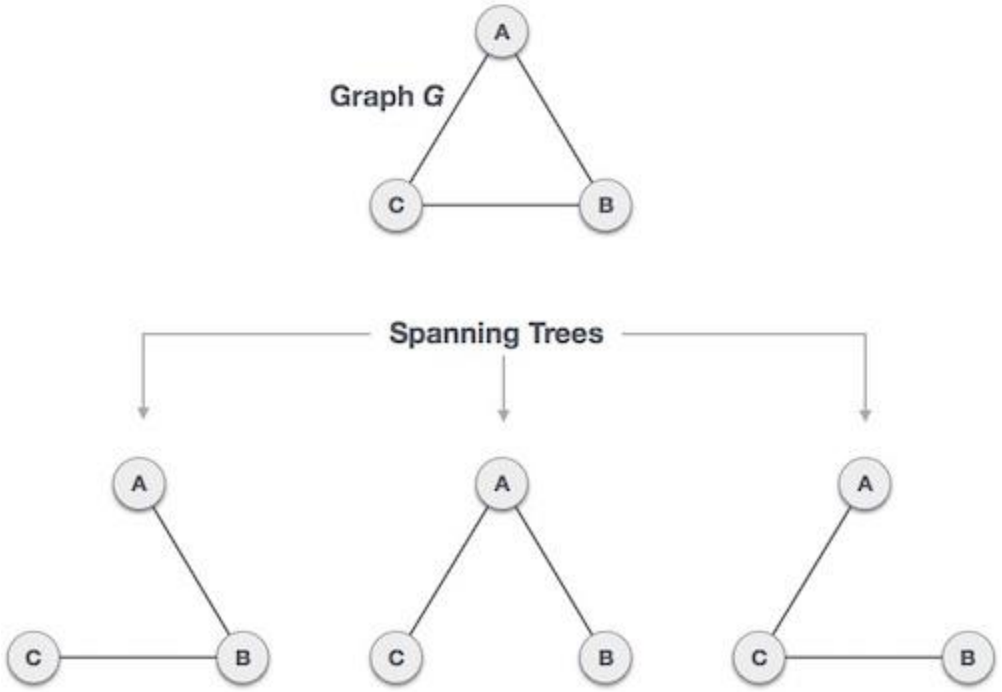


Spanning Tree

- 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.

- By this definition, we can draw a conclusion that 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.

Spanning Tree



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

Spanning Tree

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**.

Spanning Tree

Mathematical Properties of Spanning Tree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Minimum Spanning Tree (MST)

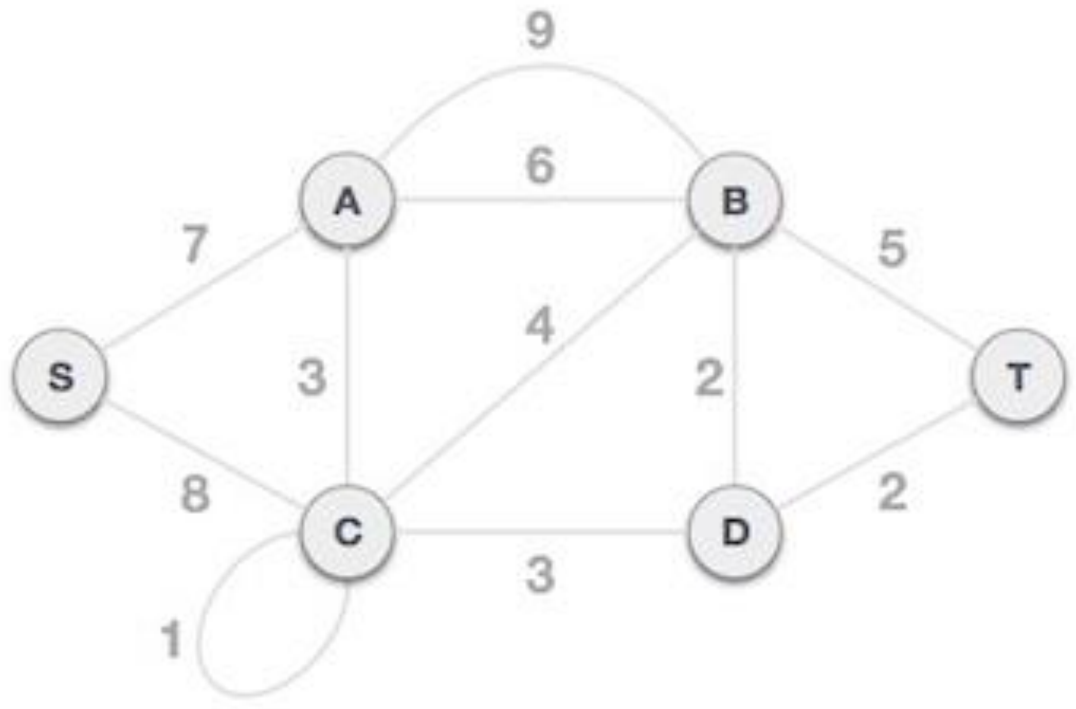
- In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.
- In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.
- Two most important spanning tree algorithms here –
  - Kruskal's Algorithm
  - Prim's Algorithm

Kruskal's 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.

Kruskal's Spanning Tree Algorithm

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

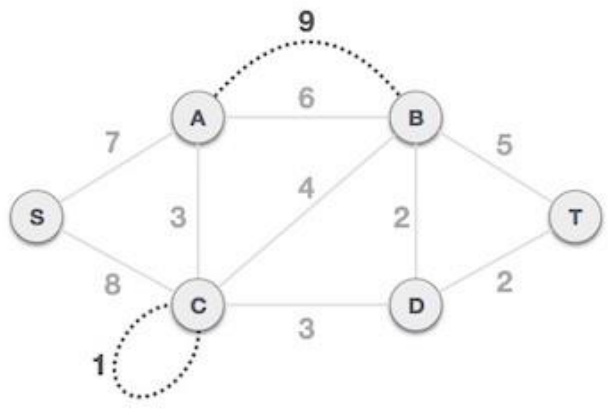


Kruskal's Spanning Tree Algorithm

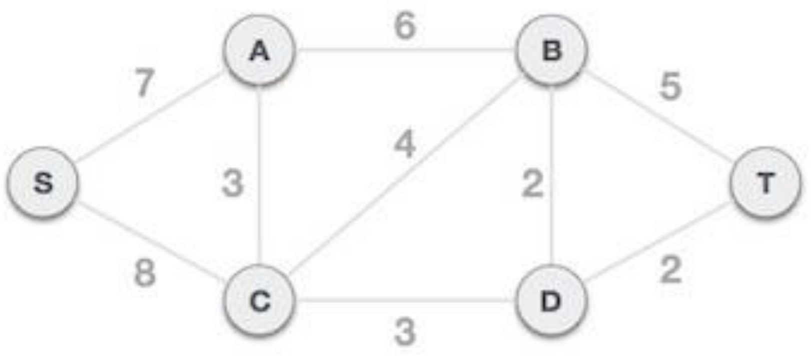
- **Algorithm Steps**
  - Sort the graph edges with respect to their weights.
  - Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
  - Only add edges which doesn't form a cycle , edges which connect only disconnected components.

Kruskal's Spanning Tree Algorithm - Solution

- Step 1 - Remove all loops and Parallel Edges
  - 1. Remove all loops and parallel edges from the given graph.
  - 2. In case of parallel edges, keep the one which has the least cost associated and remove all others.



(i)

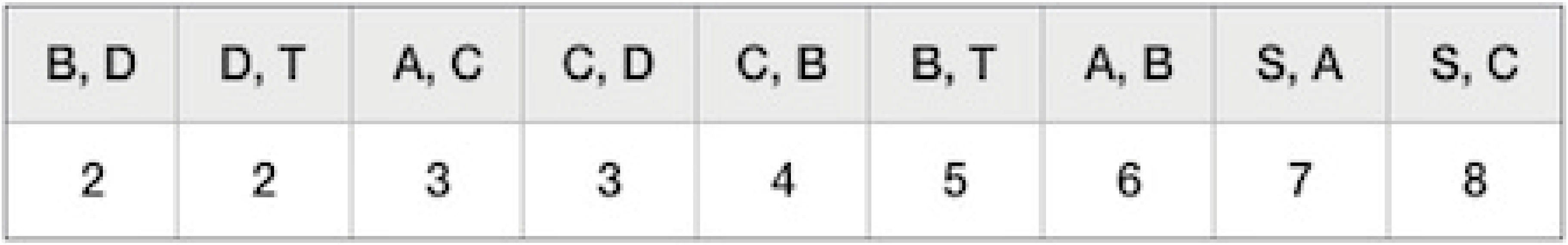


(ii)

Kruskal's Spanning Tree Algorithm - Solution

- 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).



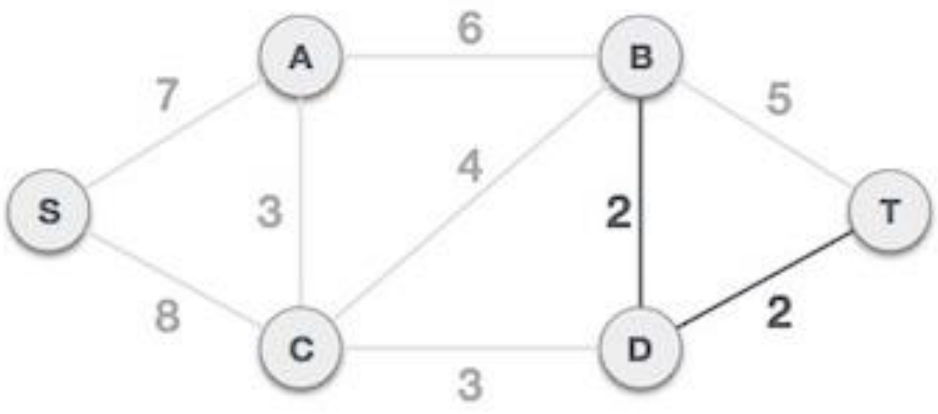
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Kruskal's Spanning Tree Algorithm - Solution

- 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.

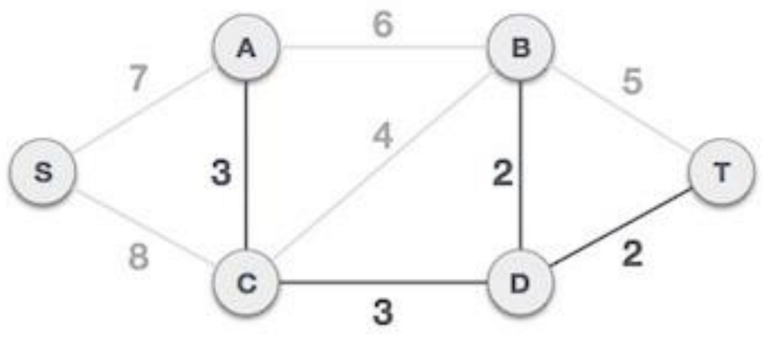
Kruskal's Spanning Tree Algorithm - Solution

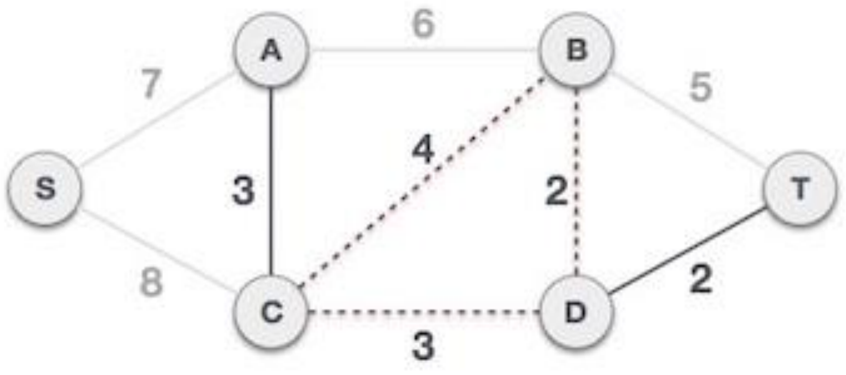
- The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.



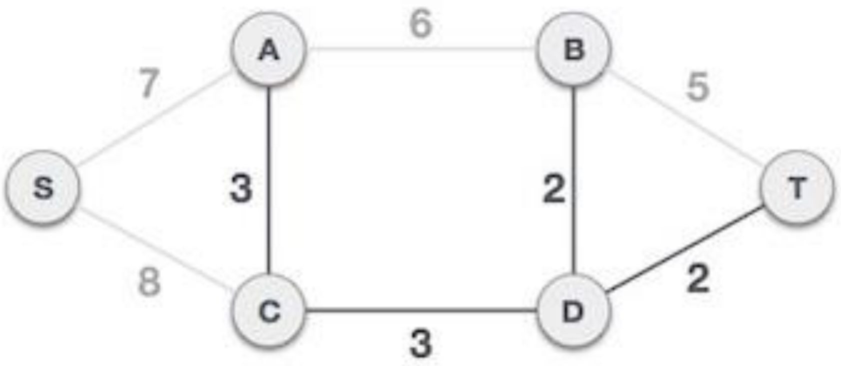
- Next cost is 3, and associated edges are A,C and C,D. We add them again –

Kruskal's Spanning Tree Algorithm - Solution



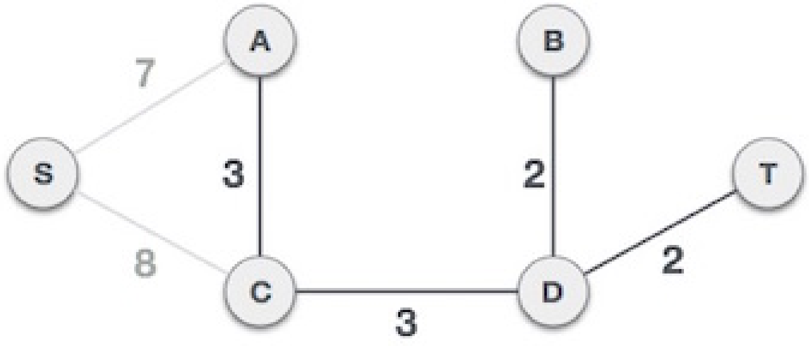


- Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

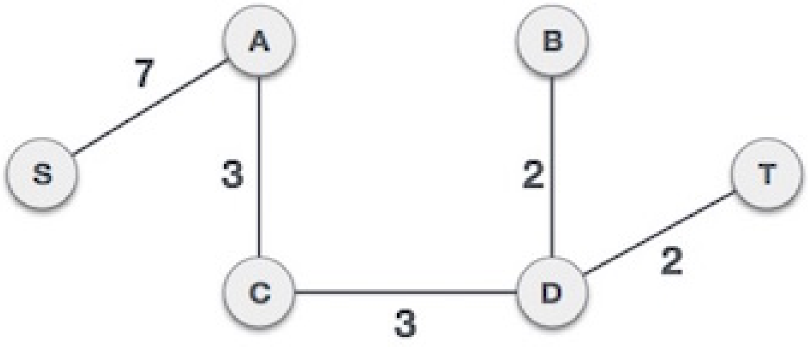


Kruskal's Spanning Tree Algorithm - Solution

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



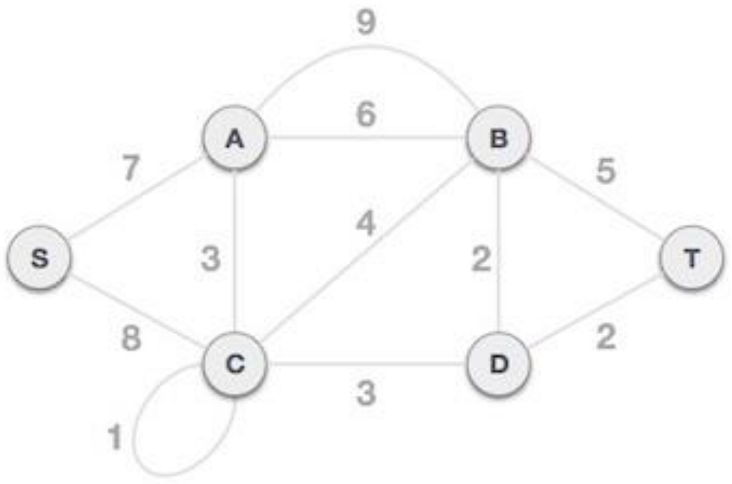
- Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



Prim's Spanning Tree Algorithm

- Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

- Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.



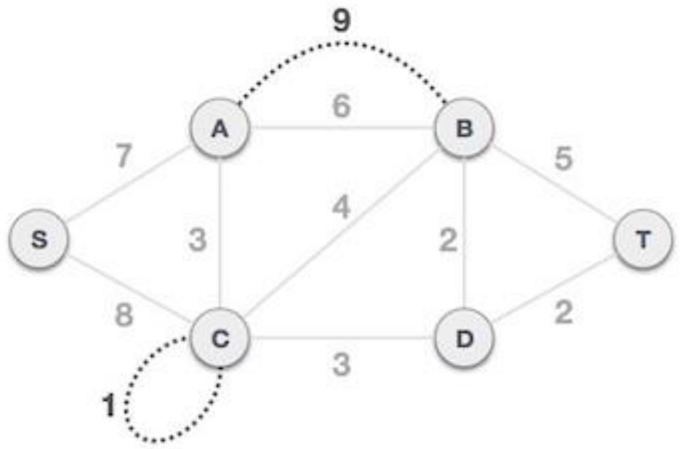
Prim's Spanning Tree Algorithm

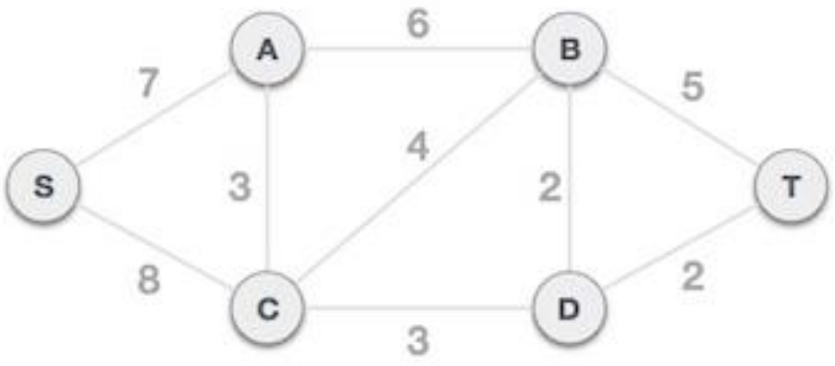
Algorithm Steps

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Prim's Spanning Tree Algorithm

- 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.



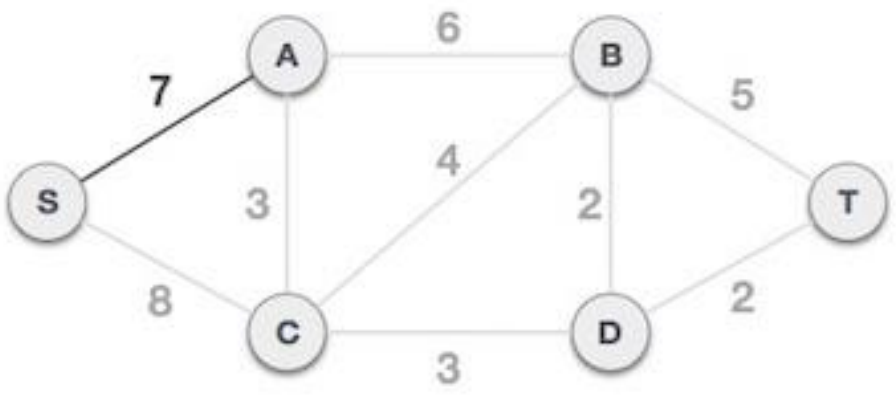


Prim's Spanning Tree Algorithm

- 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.
  - One may wonder why any video can be a root node. So the answer is, 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.

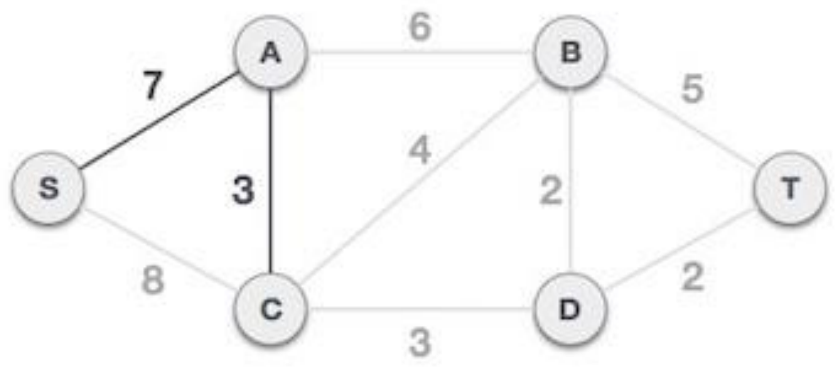
Prim's Spanning Tree Algorithm

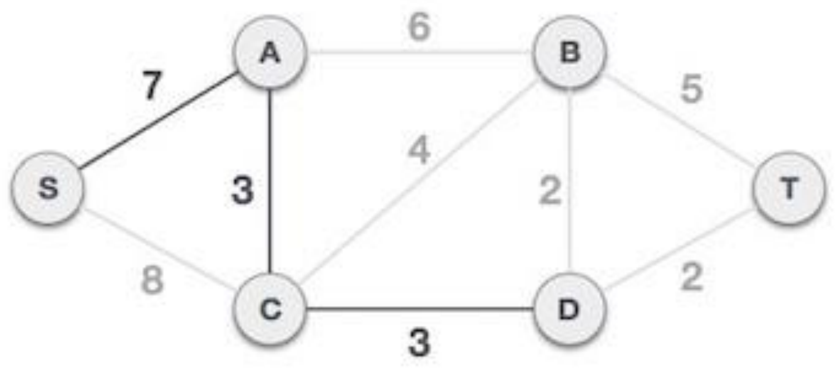
- Step 3 - Check outgoing edges and select the one with less cost
  - After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



Prim's Spanning Tree Algorithm

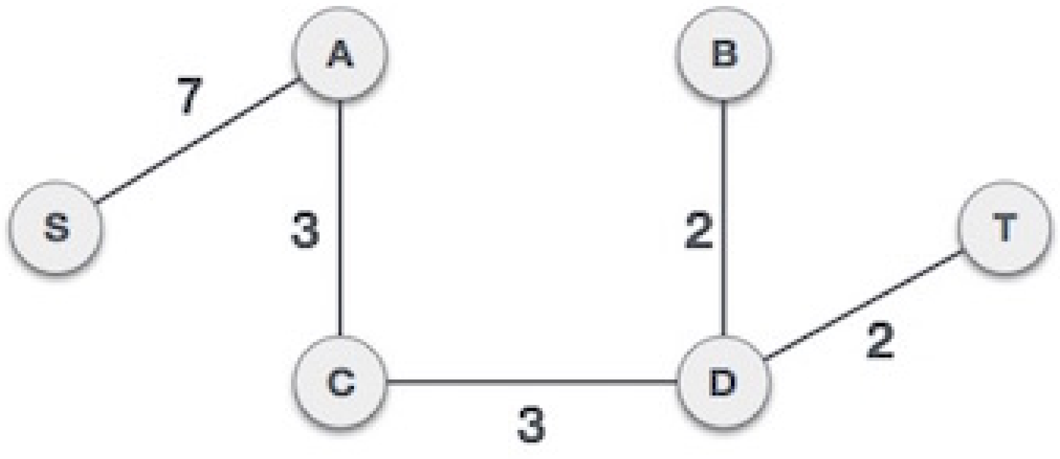
- Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.
- After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose 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.





Prim's Spanning Tree Algorithm

- 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.


A
B
7
S
3
2
T
2
C
D
3