(1) Unit – IV

## Graph — closed loop but tree is not closed loop

→ A Graph is a non linear ds, which consist of set of points known as nodes (vertices) and set of links known as edges (Arcs) which connects the vertices

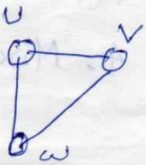
→ Graph G = (V, E) is composed of

V: set of vertices — [repr as circle]

E: set of edges connecting vertices in V

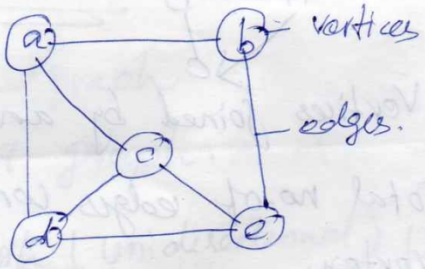
↳ [repr as lines]

→ An edge e = (U, V) is a pair of vertices.




e = (U, V)
e = (V, W)
e = (W, U)

eg:




V = {a, b, c, d, e}

E = {(a, b), (a, c), (a, d), (b, e), (b, a), (c, d), (c, a), (c, e), (d, e)}

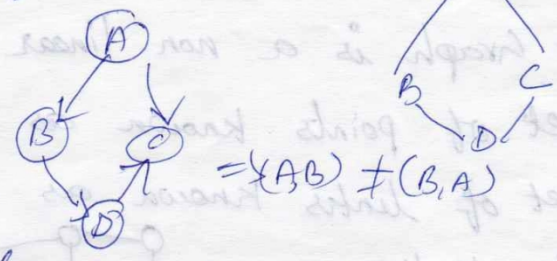
Graph Terminology:


(1) Vertex/node — individual data element of graph is called vertex/node

(2) Edges — It is a connecting link b/w two vertices
↳ 3 types

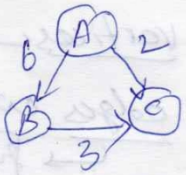
(4) 1. Undirected edge - bidirectional edge (A,B) = (B,A) ↳ same

2. directed edge - Unidirectional edge




$= (A,B) \neq (B,A)$

3. Weighted edge → edge with cost on it
↳ cost

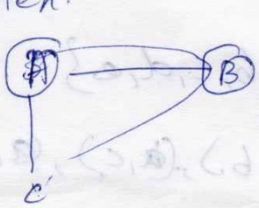



4. Mixed graph -> a graph with undirected & directed edges

5. End Vertices


↳ Two Vertices joined by an edge

(6). Degree → Total no. of edges connected to a ~~node~~ vertex.





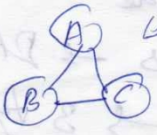
$D(A) = 2$

$D(B) = 2$

$D(C) = 2$

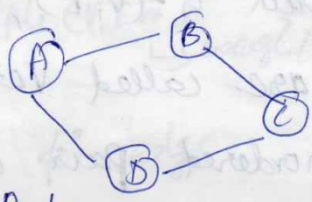
(7) Size of graph - rep. tot no. of edges in graph

size is 3



8. Path → seq of vertices from source node to destination node



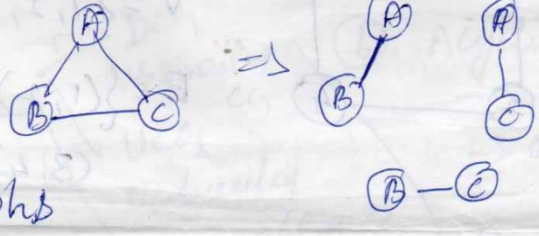



Path from A to C

A → C
src dest

→ Path will be A → B → C / A → D → C.

9. Adjacent nodes. → which connect edges. A&B are AN
A&C " "
B&C " "




Types of Graphs

① Directed graph
② Undirected graph.

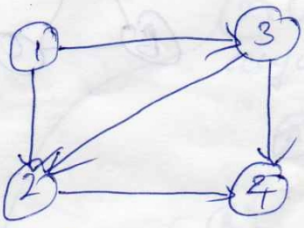
every edge have specific direction

Directed graph (Unidirectional)

A DG or digraph is a pair
G = (V, E) where V is set of elements called Vertices.
and E is set of ordered pair of element
(or)
If the pair (U, V) is ordered then graph is directed or digraphs.
U - origin, V - destination

UnDirected graph

→ No direction specify on the edges.




V = {1, 2, 3, 4}
E = {(1,2), (2,4), (3,4), (3,2), (1,3)}.

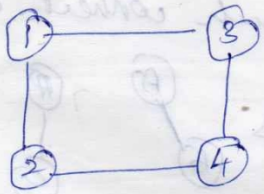
(4)

Q. Undirected graph: (Bidirectional) * No direction specify on the edge.

→ is a pair $G=(V, E)$, where V is set of whose element are called vertices (nodes) & E is a set of unordered pair of distinct elements of V (edges or undirected edges).

→ For undirected edge $\{u, v\}$ rep:

$$u-v = v-u.$$



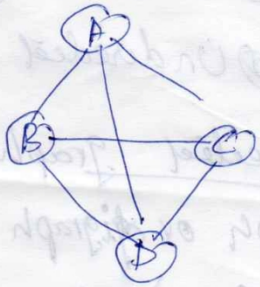

$V = \{1, 2, 3, 4\}$

$E = \{(1,2), (2,1), (1,3), (3,1), (3,4), (4,3), (4,2), (2,4)\}$

* Note

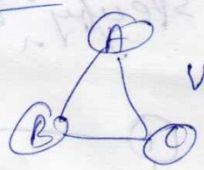
No. of possible edge in * Undirected graph is $n(n-1)/2$

eg:




* Directed graph $n^2$

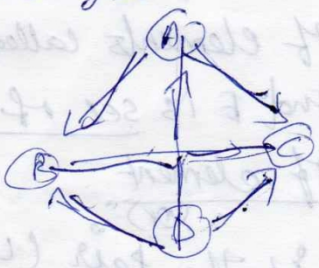
eg.




$V_h = \frac{n(n-1)}{2}$ vertices. $= \frac{3 \times 2}{2} = 3$ edges.

$D_h = n^2$

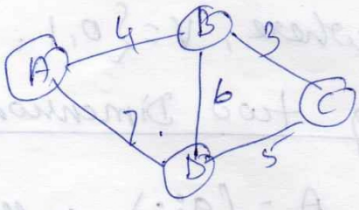
$3^2 = 9$ edges



## 3. Weighted graph

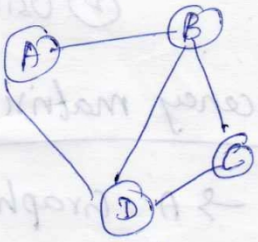
→ WG is a triple (V, E, W)
↳ weight

→ Specify cost for edge.
→ weight is specified for every edge.




## 4. Unweighted graph

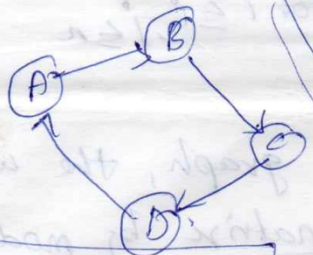
→ There is no any cost for specify that edge.




→ no weight is specified

## (5) cyclic graph

DG → Directed G
UG → Undirected G
Unweighted G
weighted G.




cycle
A → B → C → D → A

→ should perform the loops (if) find path s to D
Starting pt and endpoint should be same vertex it form cycle.

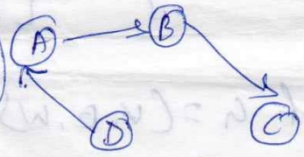
- [cycle in a DG is a simple cycle in which no vertex is repeated except 1st & last are identical].

UDG
→ def of cycle is same for UDG also, but there is a added requirement that if any edge appear more than once

cycle = ABCA

## (6) Acyclic Graph

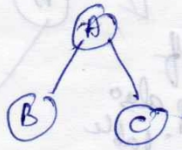
→ doesn't form cycle.




⇒ A → B → ✗

(1) → DAG → Directed acyclic graph (no specific cycle)

UAG.
(2) Undirected acyclic graph




(3) WDAG
(4) WUDAG

# Types of graph

## 1. Directed Graph(unidirectional)

- If a graph contains ordered pair of vertices, is said to be a Directed Graph or digraph.
- Directed Graph or digraph G= (V, E) Where V is the set of elements called verities and E is a set of ordered pair of element.
- If an edge is represented using a pair of vertices ($V_1$, $V_2$), the edge is said to be directed from $V_1$ to $V_2$.
- The first element of the pair $V_1$ is called the start vertex and the second element of the pair $V_2$ is called the end vertex.

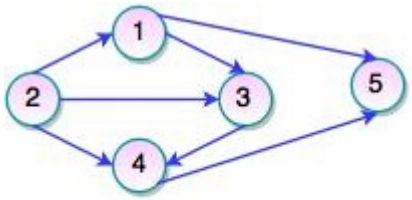



Fig. Directed Graph

**Set of Vertices V = {1, 2, 3, 4, 5}**
**Set of Edges E = {(1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}**

## 2. Undirected Graph(bidirectional)

- Undirected Graph G= (V, E) Where V is the set of elements called verities and E is a set of unordered pair of element.
- If a graph contains unordered pair of vertices, is said to be an Undirected Graph.
- In this graph, pair of vertices represents the same edge.

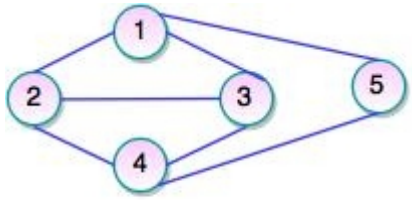


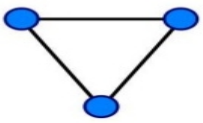

Fig. Undirected Graph

Set of Vertices V = {1, 2, 3, 4, 5}
Set of Edges E = {(1, 2), (1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5) (3,1) (5,1) (4,2) (3 ,2)}

- In an undirected graph, the nodes are connected by undirected arcs.
- It is an edge that has no arrow. Both the ends of an undirected arc are equivalent.

**Note:**

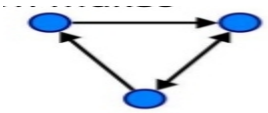
i. Number of possible edges in undirected graph is n (n-1)/2



Undirected graph = n (n-1)/2

=3(3-1)/2=6/2=3 edges

ii. Number of possible edges in directed graph is $n^2$



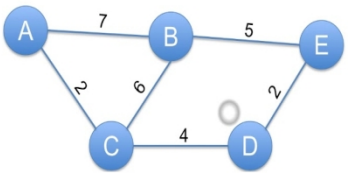
Directed graph = $n^2$ (n is no of vertices)

= $3^2$ =9 edges

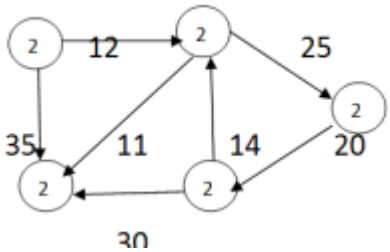
**3. Weighted graph**

- Graphs whose edges or paths have values. All the values seen associated with the edges are called weights. Edges value can represent weight/cost/length.
- G=(V,E,W)

**Weighted undirected graph**




**Weighted directed graph**

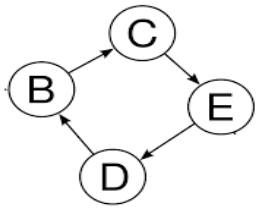



**4. Un weighted Graph**

- Where there is no value or weight associated with the edge. By default, all the graphs are un weighted unless there is a value associated.

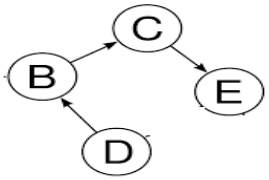
**5. Cyclic and Acyclic Graphs**

A cyclic graph is a directed graph which contains a path from at least one node back to itself. In simple terms cyclic graphs contain a cycle.




**B -> C -> E -> D -> B**

An acyclic graph is a directed graph which contains absolutely no cycle, that is no node can be traversed back to itself.




**D->B -> C -> E**

**Graph representation**

Two common data structures for representing graphs:

1. Adjacency matrix
2. Adjacency lists

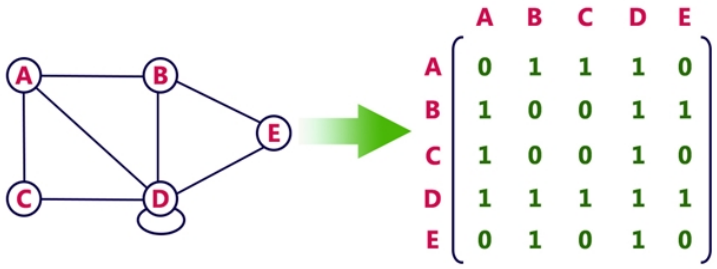
**1. Adjacency Matrix**

- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- In this representation, we have to construct a nxn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a_{i,j} = 1$, otherwise $a_{i,j} = 0$.
- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

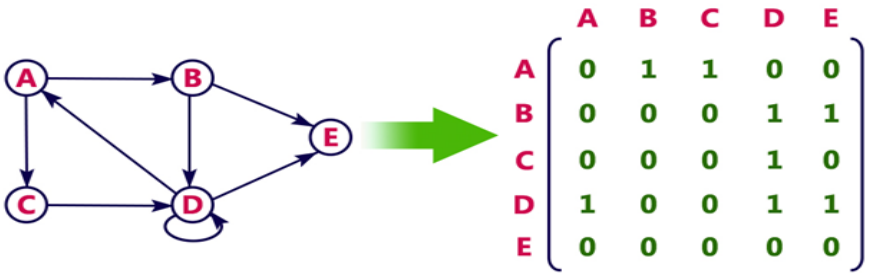
Example

Consider the following **undirected graph representation**:

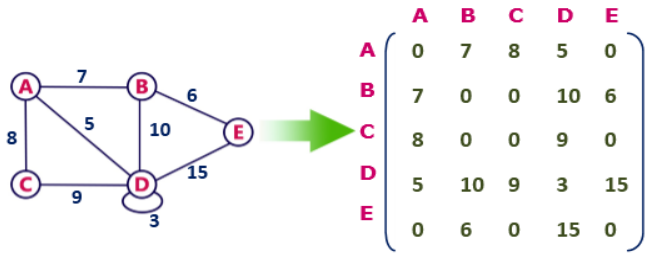
- **Undirected graph representation**




- **Directed graph representation**




- In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.
- **Undirected weighted graph representation**



**Pros:** Representation is easier to implement and follow.

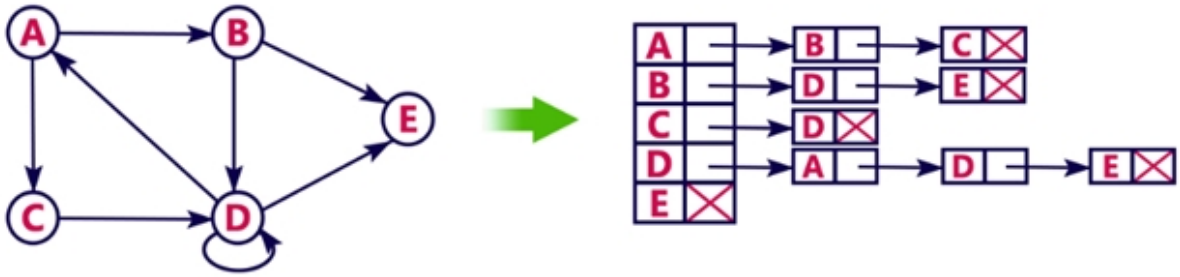
**Cons:** It takes a lot of space and time to visit all the neighbors of a vertex, we have to traverse all the vertices in the graph, which takes quite some time.

**2. Adjacency List**

- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a singly linked list of neighbors of v.

Example

Let's see the following directed graph representation implemented using linked list:




**Pros:**

- Adjacency list saves lot of space.
- We can easily insert or delete as we use linked list.
- Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

**Cons:**

- The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.

## 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)**

## 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

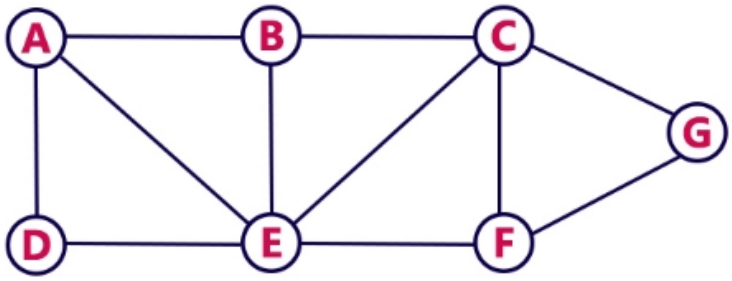
---

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

---

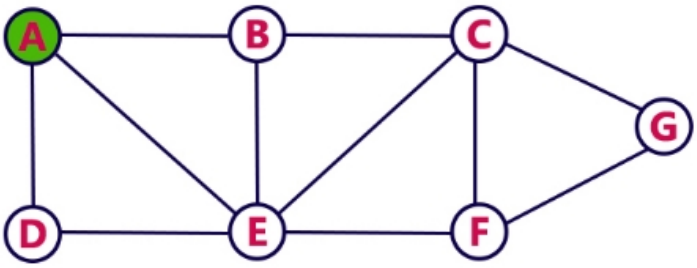
Example

Consider the following example graph to perform DFS traversal




**Step 1:**

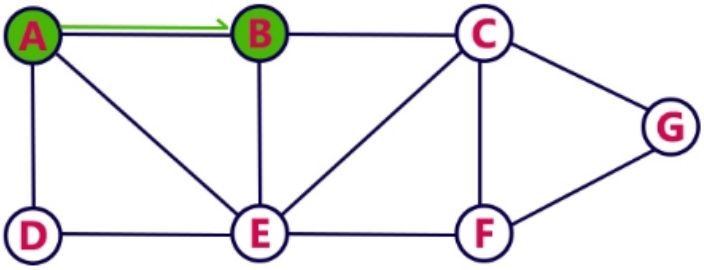
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.





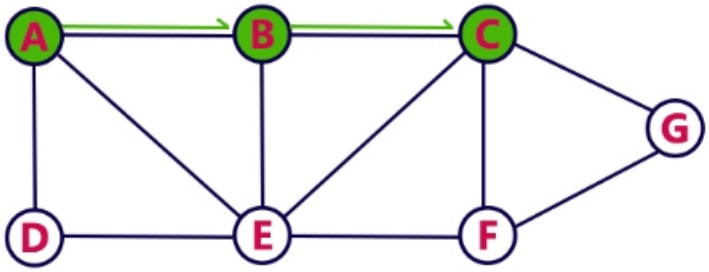
**Step 2:**

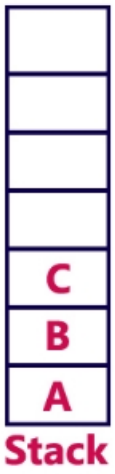
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.




**Step 3:**

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

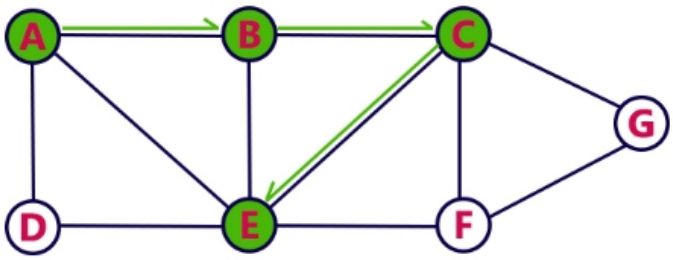



Stack

**Step 4:**

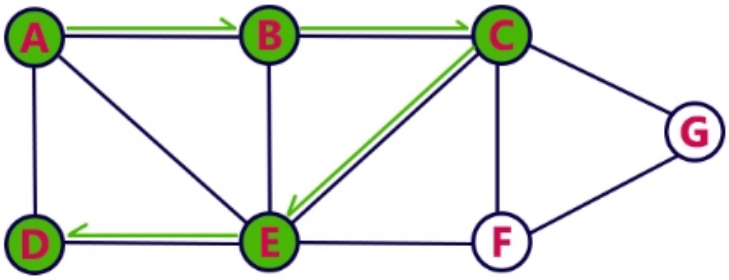
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

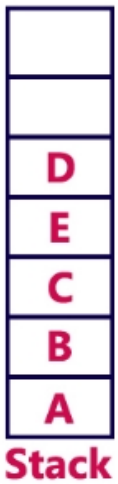




**Step 5:**

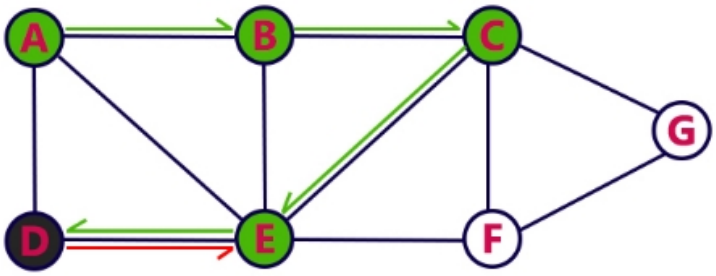
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

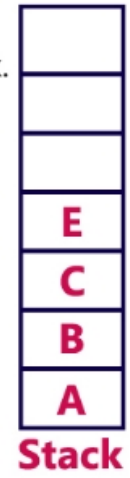


**Step 6:**

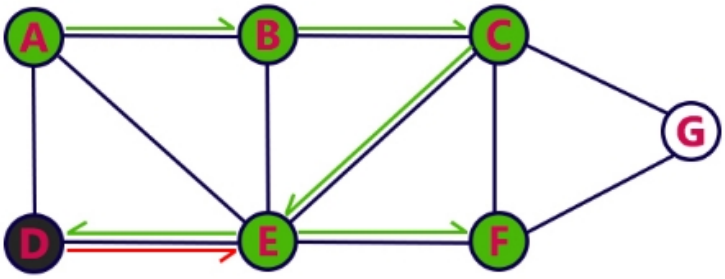
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.




**Step 7:**

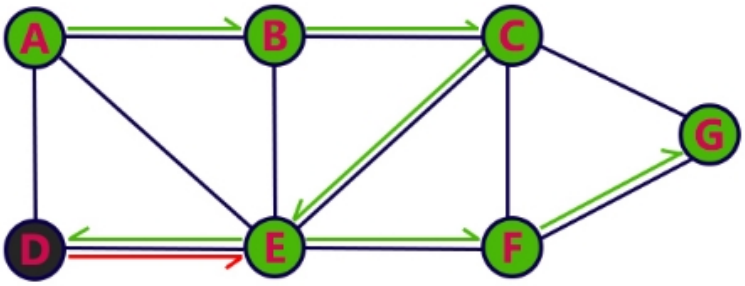
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.





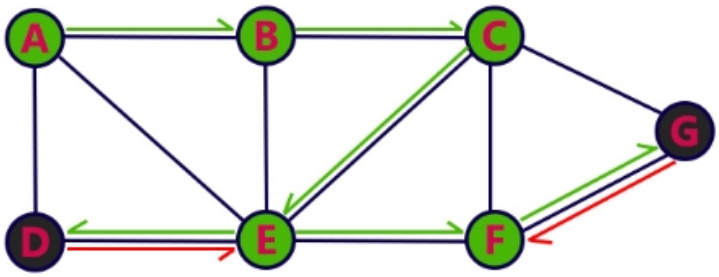
**Step 8:**

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




**Step 9:**

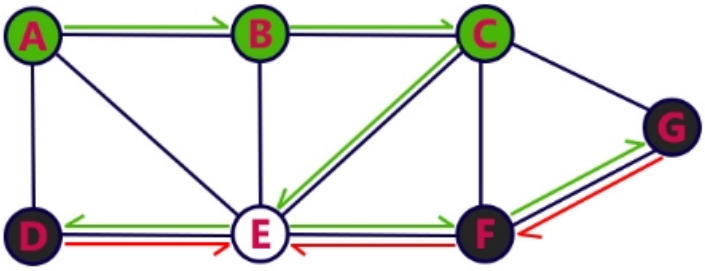
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.





**Step 10:**

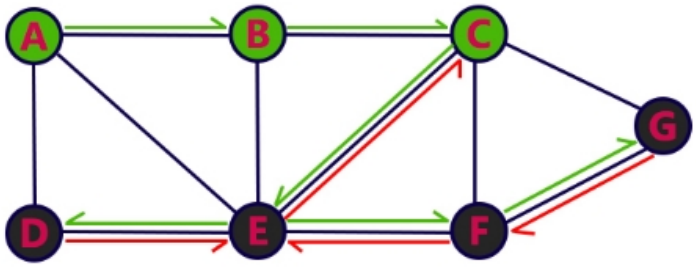
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

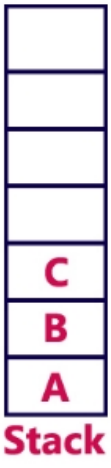




**Step 11:**

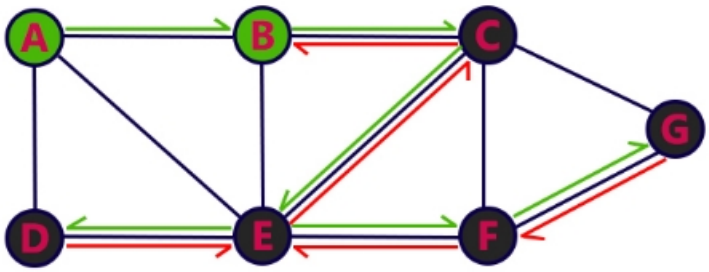
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



**Step 12:**

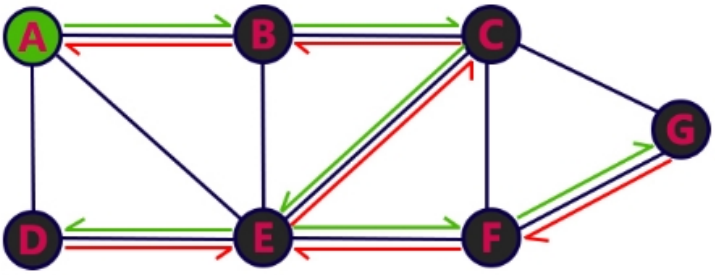
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.





**Step 13:**

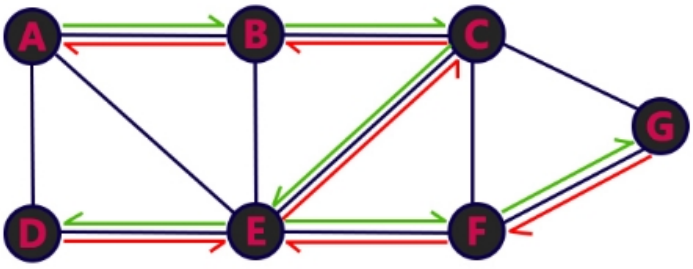
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



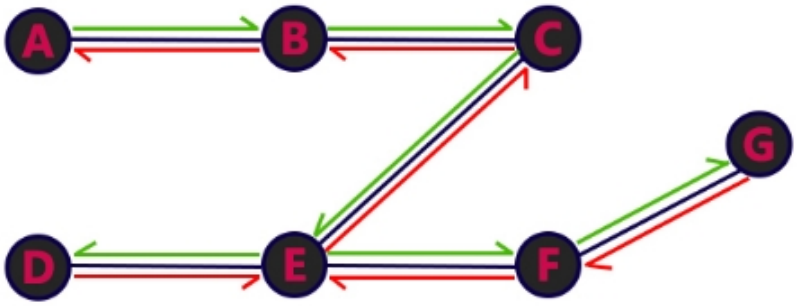


**Step 14:**

- 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.




# BFS (Breadth First Search)

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.

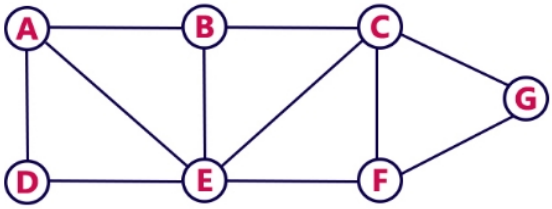
**Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

**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

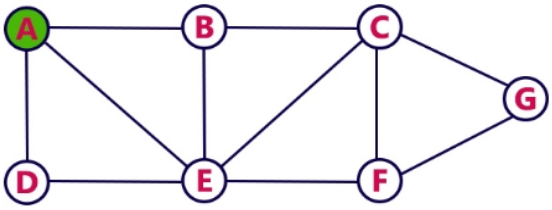
Example

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.

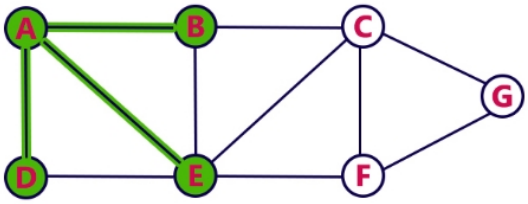



**Queue**

| A | | | | | | |
|---|---|---|---|---|---|---|

**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..

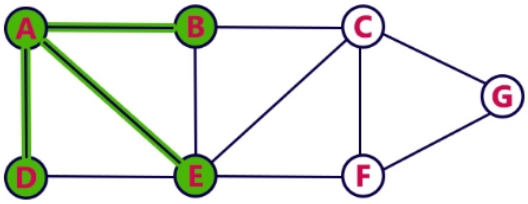



**Queue**

| | D | E | B | | | |
|---|---|---|---|---|---|---|

**Step 3:**

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

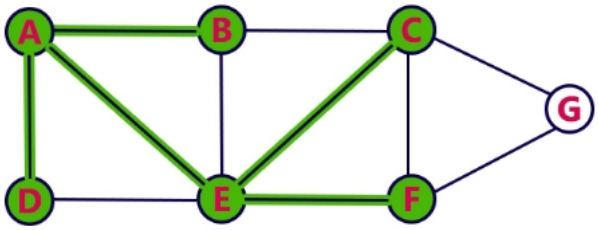



**Queue**

| | | E | B | | | |
|---|---|---|---|---|---|---|

**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.

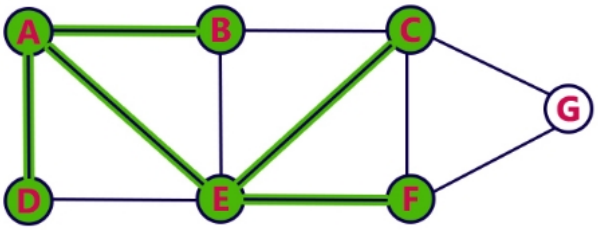



**Queue**

| | | | B | C | F | |
|---|---|---|---|---|---|---|

**Step 5:**

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

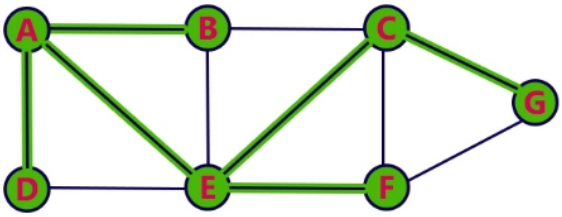



**Queue**

| | | | | C | F | |
|---|---|---|---|---|---|---|

**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.

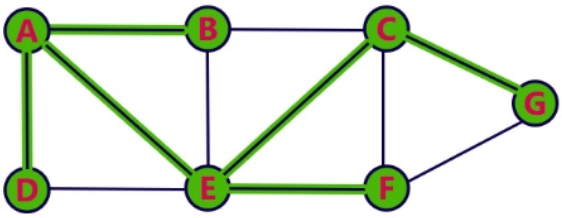



**Queue**

| | | | | | F | G |
|---|---|---|---|---|---|---|

**Step 7:**

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

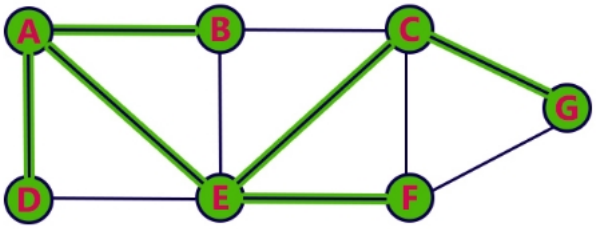



**Queue**

| | | | | | | G |
|---|---|---|---|---|---|---|

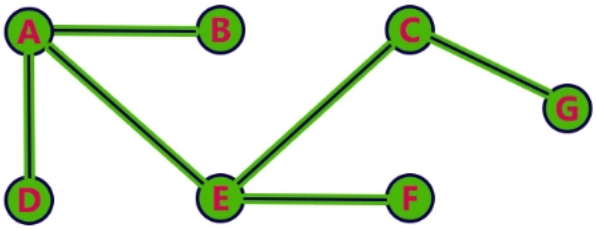
**Step 8:**

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

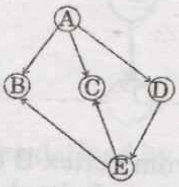
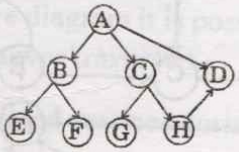



**Queue**

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

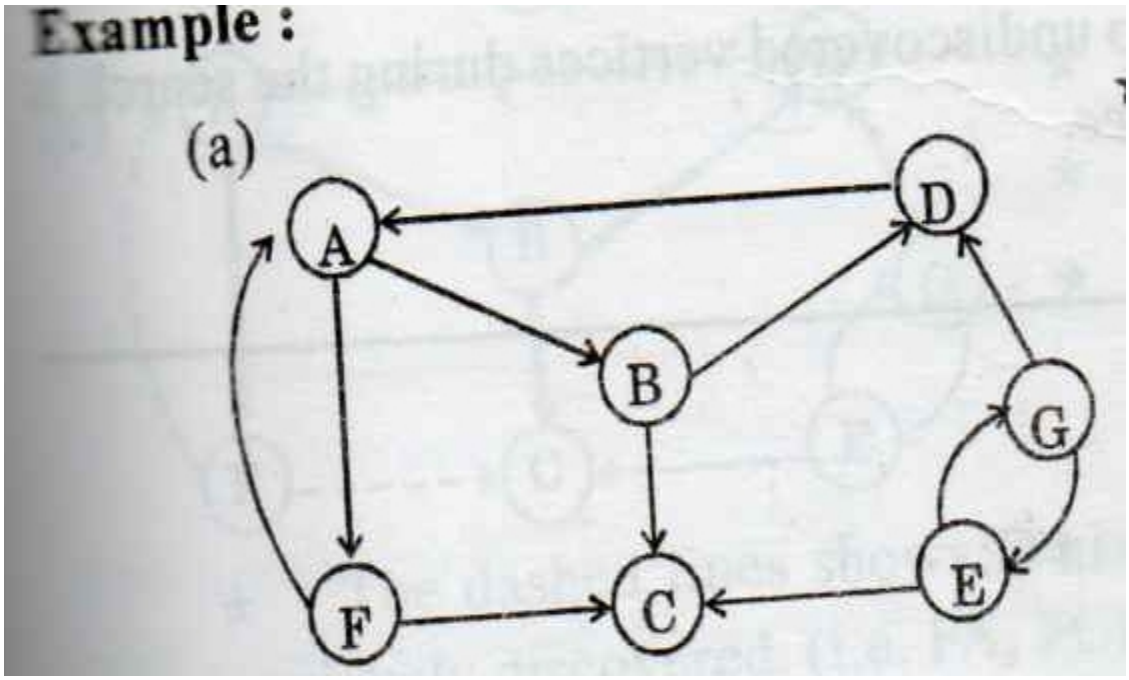



### 6.8.3 Depth first search Vs Breadth first search

| Depth first search | Breadth first search |
|---|---|
| 1. Back tracking is possible from a dead end. | 1. Backtracking is not possible. |
| 2. Vertices from which exploration is incomplete are processed in a LIFO order. | 2. The vertices to be explored are organized as a FIFO queue. |
| 3. Search is done in one particular direction at the time. | 3. The vertices in the same level are maintained parallely. (left to right) (alphabetical ordering) |
| 4. Example :   Order of traversal A → B → C → D → E | 4. Example :   Order of traversal A B C D E F G H |

# H.W

# To construct DFS and BFS

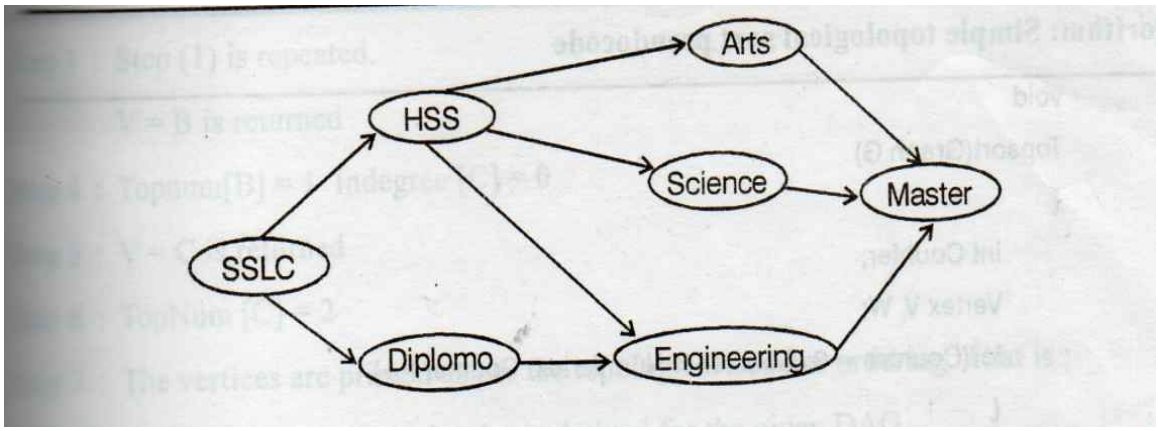



# Topological Sort

- Topological Sort is a linear ordering of the vertices in a Directed Acyclic Graph(DAG) such that if there is a path from $v_i$ to $v_j$ ,then $v_j$ appear $v_i$ in the ordering.
- Topological Sorting is possible if and only if the graph is a Directed Acyclic Graph.
- There may exist multiple different topological orderings for a given directed acyclic graph.

**Example :**

A course prerequistic structure is shown as a graph in this example. A directed edge between (SSLC, HSS) indicates that course SSLC must be completed before attempting to course HSS.



A topological ordering of these courses is any course sequence that does not violate the prerequistic requirement. The legal topological orderings are:

(i) SSLC, HSS, Arts, Master

(ii) SSLC, HSS, Science, Master

(iii) SSLC, HSS, Engineering, Master

(iv) SSLC, Diplomo, Engineering, Master

From the example, it is clear that the ordering is not necessarily unique, any legal ordering is valid. Topological ordering is not possible if the graph has a cycle, since for two vertices $v$ and $w$ on the cycle, $v$ preceedes $w$ and $w$ preceeds $v$.

**The Strategy**

(i) Find any vertex with no incoming edges (i.e.) indegree = 0. If such vertex found, print the vertex and remove it along with its edges from the graph.

(ii) Repeat step (i) on the rest of the graph.

**Simple Topological Sort**

The steps to perform simple topological sort is:

(i) Assuming that the Indegree array is initialised and the graph is read into an adjacency list.

(ii) The function FindNewVertexOfIndegreeZero scans the indegree array, to find a vertex with indegree 0, that has not already been assigned a topological number.

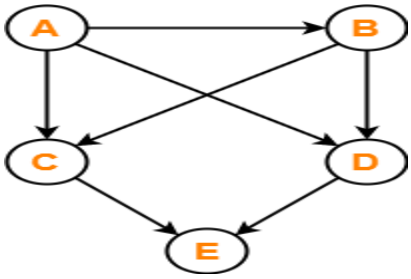
(a) It returns NotAVertex if no such vertex exists, that indicates that the graph has a cycle.

(b) If vertex (v) is returned, the topological number is assigned to $v$, then the indegree of vertices (w) adjacent to vertex (v) are decremented.

(iii) Repeat step 2 for the rest of the graph.

# Problem-1:

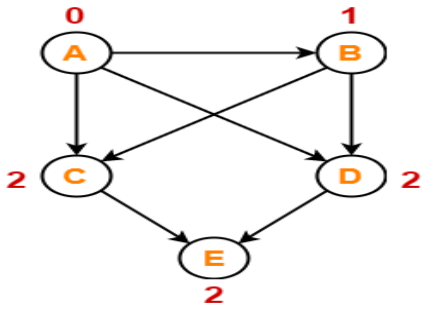
Find the number of different topological orderings possible for the given graph-




# Solution-

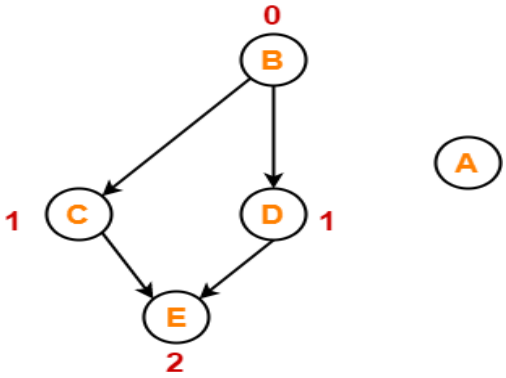
The topological orderings of the above graph are found in the following steps-

**Step1:** Write in-degree of each vertex-



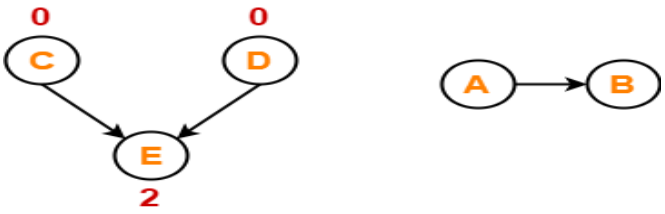

**Step-2:**

- Vertex-A has the least in-degree.
- So, remove vertex-A and its associated edges.
- Now, update the in-degree of other vertices.



**Step-3:**

- Vertex-B has the least in-degree.
- So, remove vertex-B and its associated edges.
- Now, update the in-degree of other vertices.




**Step-4:**

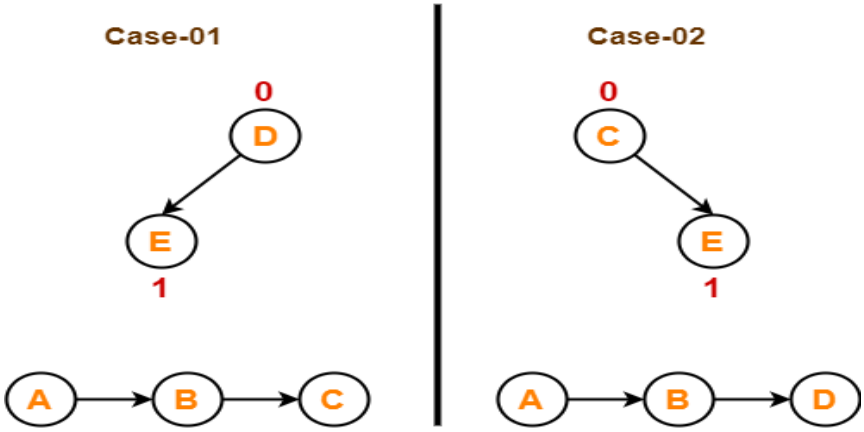
- There are two vertices with the least in-degree. So, following 2 cases are possible-

  Case-1:

  - Remove vertex-C and its associated edges.
  - Then, update the in-degree of other vertices.

  Case-2:

  - Remove vertex-D and its associated edges.
  - Then, update the in-degree of other vertices.




**Step-5:**

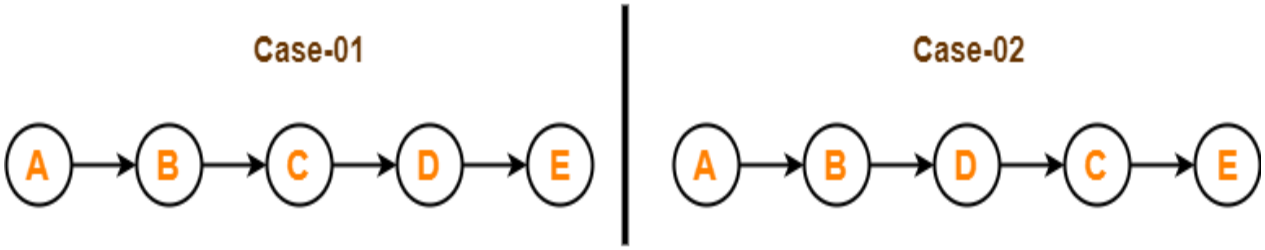
- Now, the above two cases are continued separately in the similar manner.

  Case-1

  - Remove vertex-D since it has the least in-degree.
  - Then, remove the remaining vertex-E.

Case-2:

- Remove vertex-C since it has the least in-degree.
- Then, remove the remaining vertex-E.



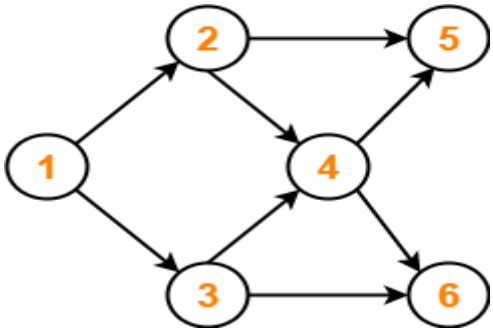

**Conclusion**

For the given graph, following **2** different topological orderings are possible-

- **A B C D E**
- **A B D C E**

# Problem-2:

Find the number of different topological orderings possible for the given graph-

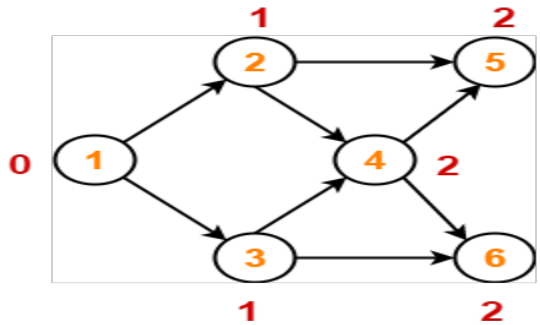



**Solution**

The topological orderings of the above graph are found in the following steps-

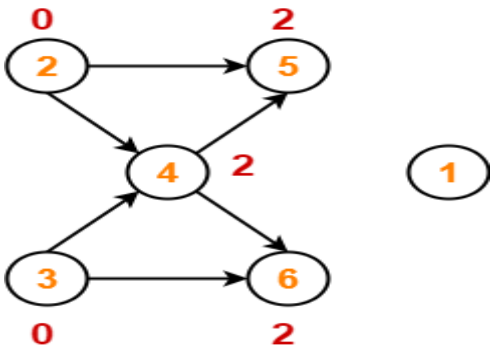
**Step-1:**

Write in-degree of each vertex-




**Step-2:**

- Vertex-1 has the least in-degree.
- So, remove vertex-1 and its associated edges.
- Now, update the in-degree of other vertices.




**Step-3:**

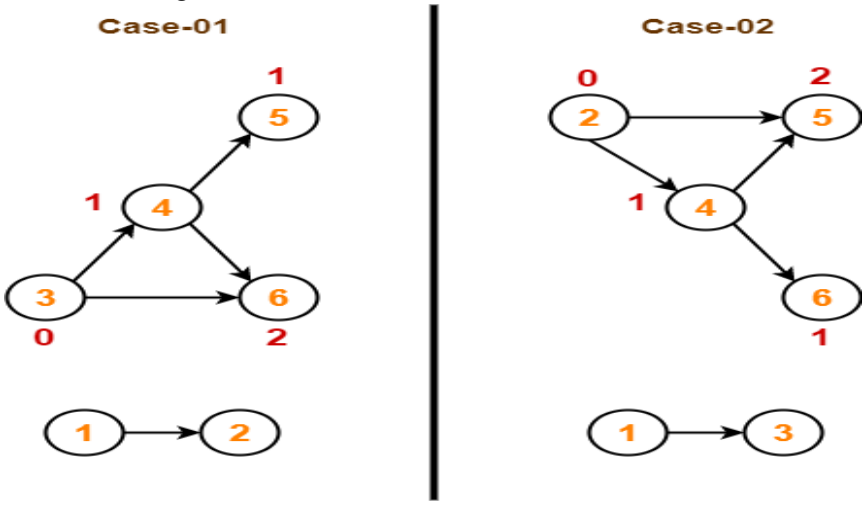
There are two vertices with the least in-degree. So, following 2 cases are possible-

case-1

- Remove vertex-2 and its associated edges.
- Then, update the in-degree of other vertices.

case-2:

- Remove vertex-3 and its associated edges.
- Then, update the in-degree of other vertices.




**Step-4:**

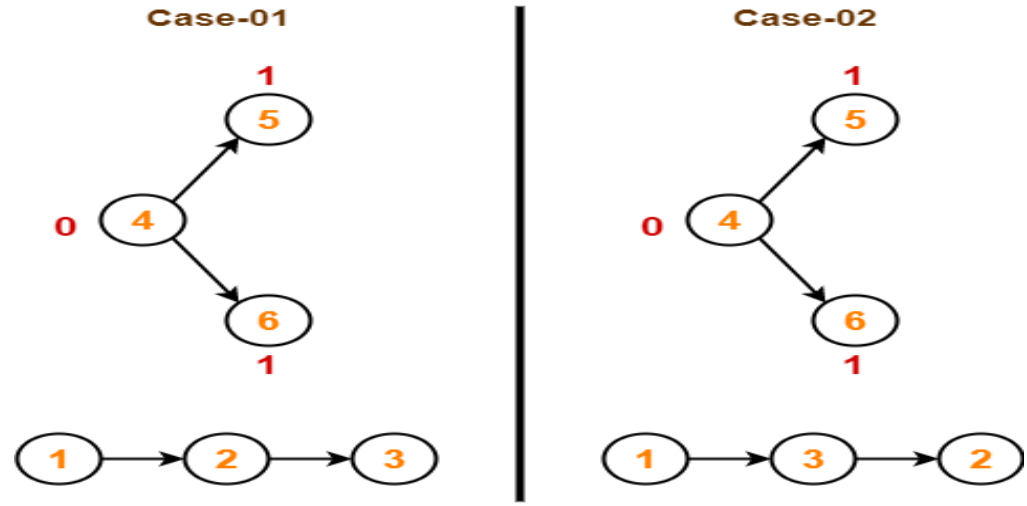
Now, the above two cases are continued separately in the similar manner.

Case-1:

- Remove vertex-3 since it has the least in-degree.
- Then, update the in-degree of other vertices.

Case-2:

- Remove vertex-2 since it has the least in-degree.
- Then, update the in-degree of other vertices.



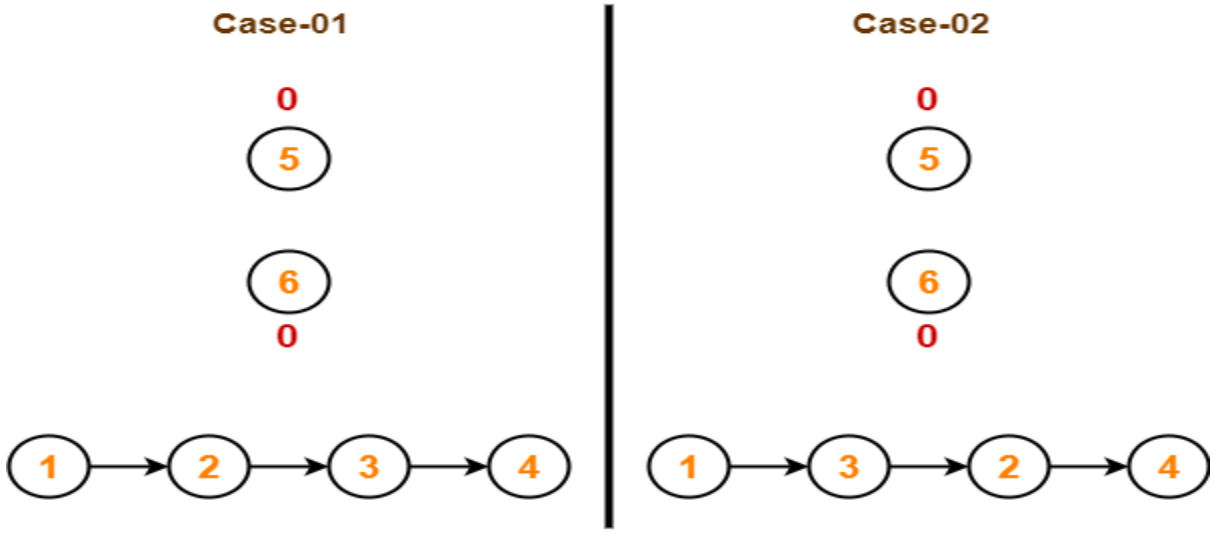

**Step-5:**

Case-1:

- Remove vertex-4 since it has the least in-degree.
- Then, update the in-degree of other vertices.

Case-2:

- Remove vertex-4 since it has the least in-degree.
- Then, update the in-degree of other vertices.

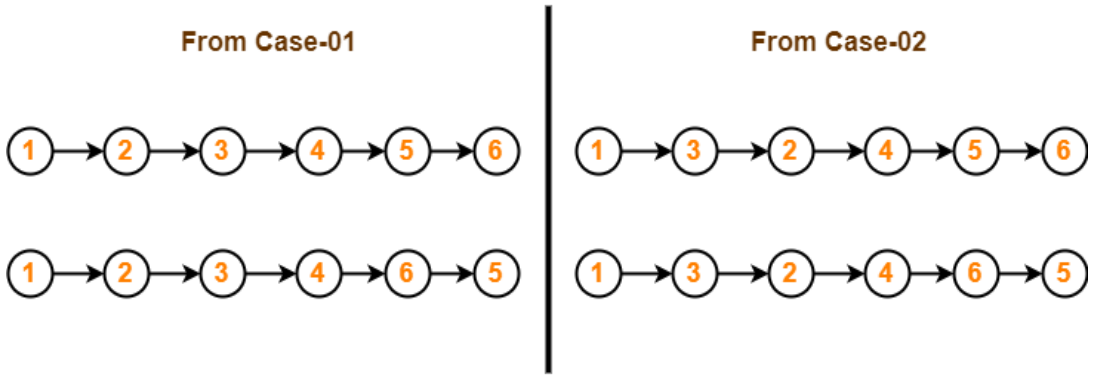



**Step-6:**

In case-01,

- There are 2 vertices with the least in-degree.
- So, 2 cases are possible.
- Any of the two vertices may be taken first.

Same is with case-02.



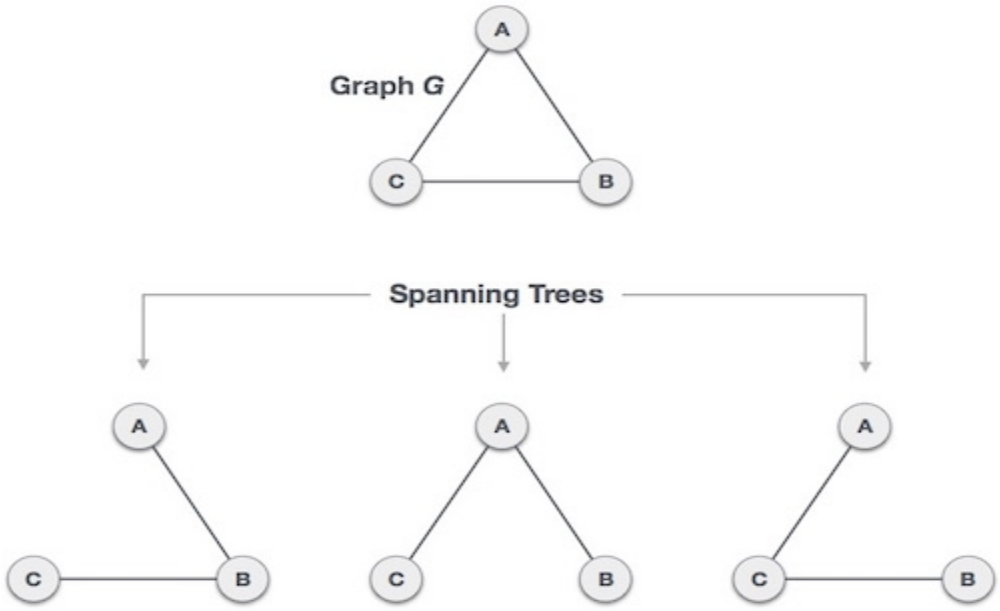

**Conclusion:**

For the given graph, following **4** different topological orderings are possible-

- **1 2 3 4 5 6**
- **1 2 3 4 6 5**
- **1 3 2 4 5 6**
- **1 3 2 4 6 5**

## 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.




- 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.

**General Properties of Spanning Tree**

- 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 are −
    - Civil Network Planning
    - Computer Network Routing Protocol
    - Cluster Analysis
- Let us understand this through a small example.
- Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes.
- This is where the spanning tree comes into picture.

**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**

- We shall learn about two most important spanning tree algorithms here −
    - Kruskal's Algorithm
    - Prim's Algorithm
- Both are greedy algorithms.

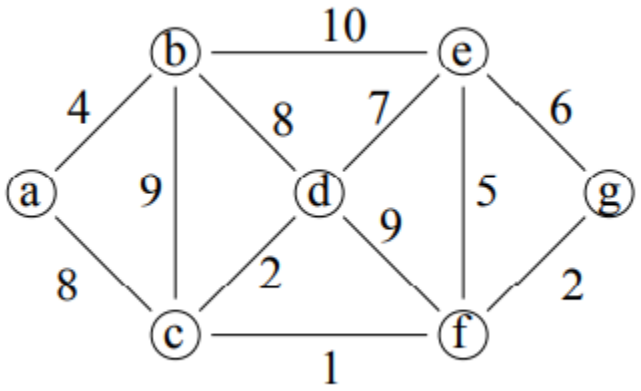
**Prim's Algorithm**

- Prim's Algorithm is used to find the minimum spanning tree from a graph.
- Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
- Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step.
- The edges with the minimal weights causing no cycles in the graph got selected.

**The algorithm is given as follows.**

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

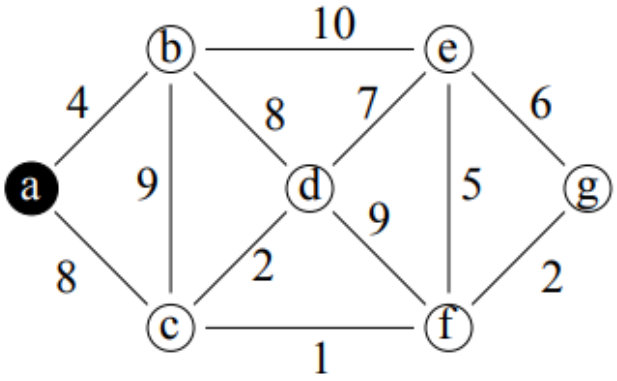
EXAMPLE:




Connected graph

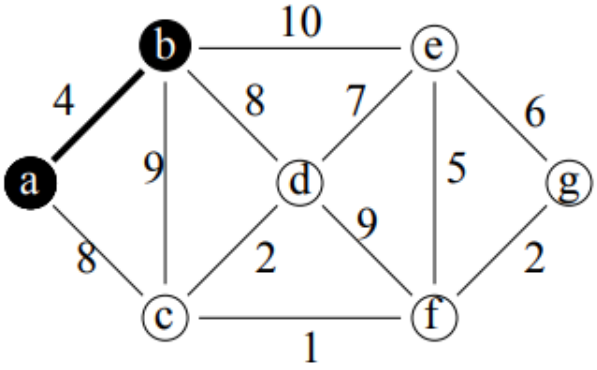
SOLUTION:

**STEP 1:**



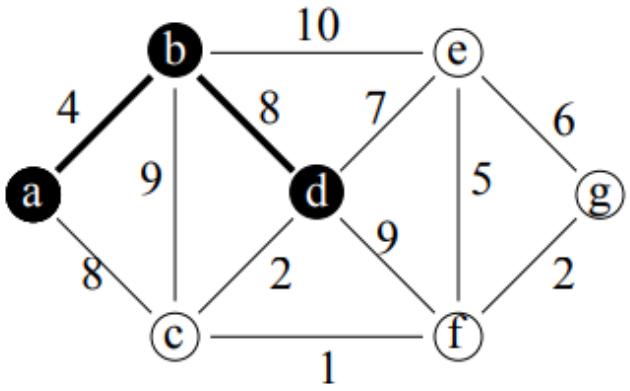

S={a}
V \ S = {b,c,d,e,f,g}
A={}
lightest edge = {a,b}

**STEP 2:**



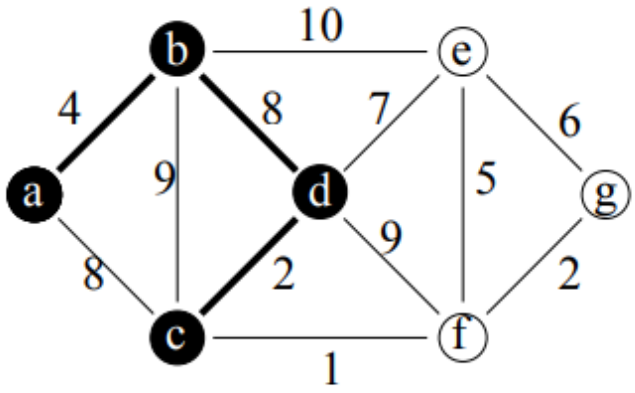

S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}

**STEP 3:**



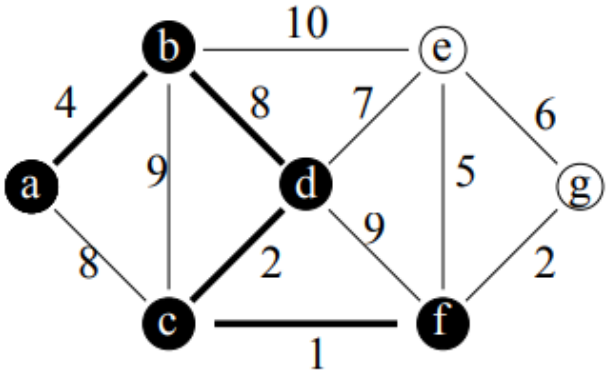

S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}

**STEP 4:**



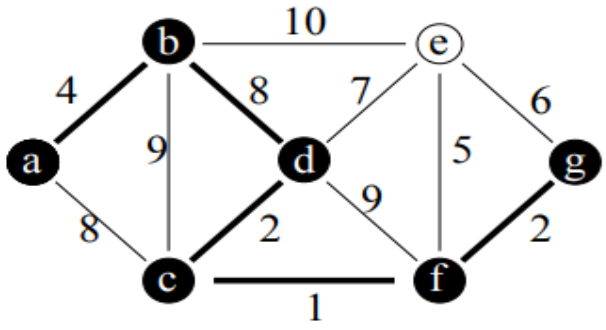

S={a,b,c,d}
V \ S = {e,f,g}
A={{a,b},{b,d},{c,d}}
lightest edge = {c,f}

**STEP 5:**




S={a,b,c,d,f}
V \ S = {e,g}
A={{a,b},{b,d},{c,d},{c,f}}
lightest edge = {f,g}

**STEP 6:**




Step 1.5 after

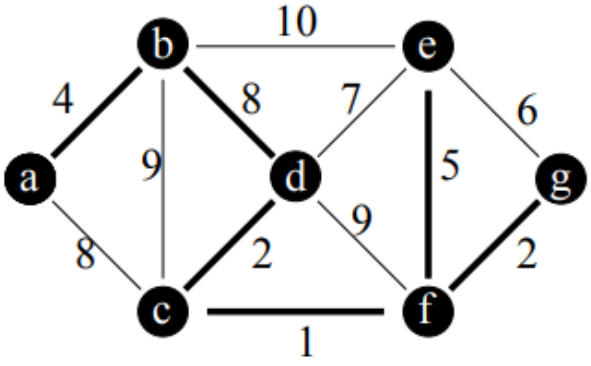
S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}

**STEP 7:**



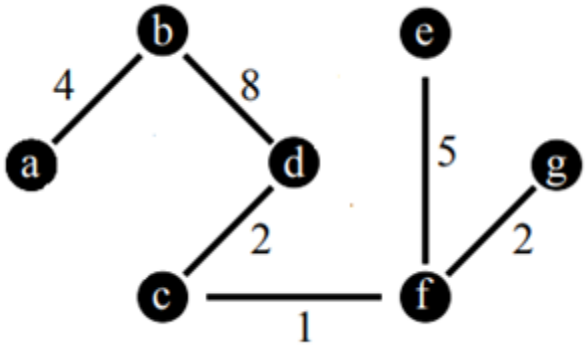

S={a,b,c,d,e,f,g}

V \ S = {}

A={{a,b},{b,d},{c,d},{c,f},
{f,g},{f,e}}

MST completed

**FINAL OUTPUT**

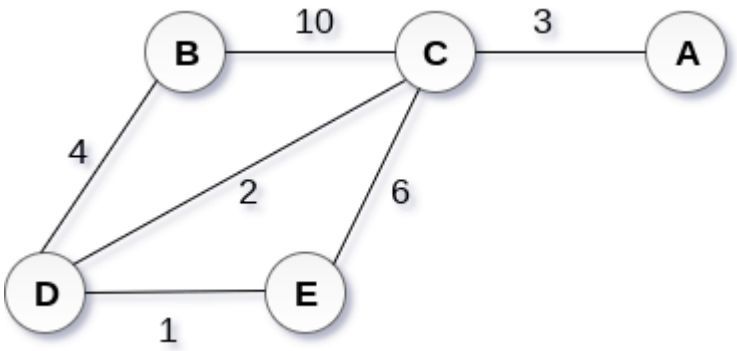



**The cost of MST will be calculated as;**

**cost(MST) = 4 + 8 + 2+ 1 + 5+2 = 22 units.**

**Example 2:**

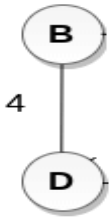
**Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.**



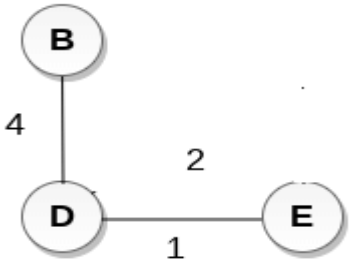

**SOLUTION**

**STEP 1:**

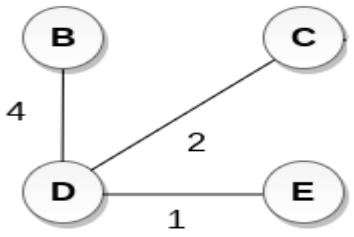

**STEP 2:**



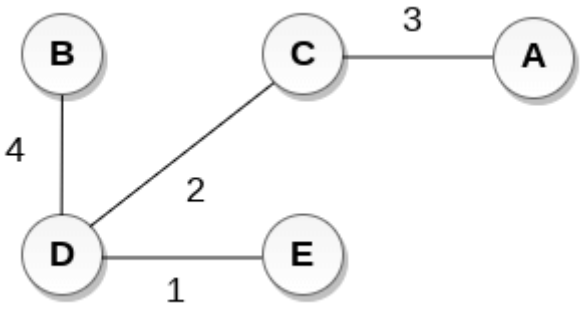

**STEP 3:**




**STEP 4:**




**STEP 5:**




**The cost of MST will be calculated as;**

**cost(MST) = 4 + 2 + 1 + 3 = 10 units.**

**This is the final output.**

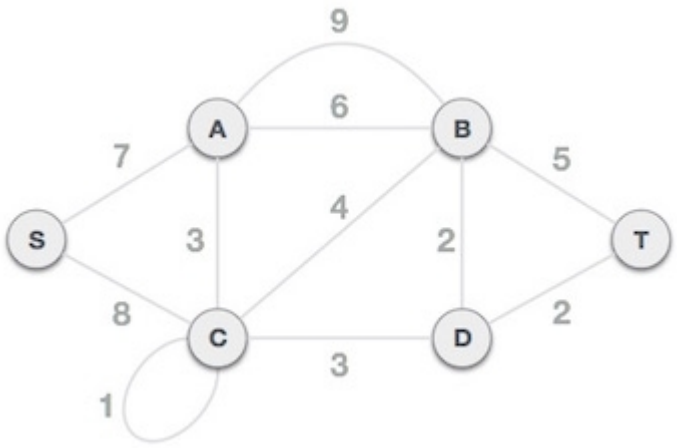
**<u>Kruskal's Algorithm</u>**

- Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph.
- The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph.
- Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

The Kruskal's algorithm is given as follows.

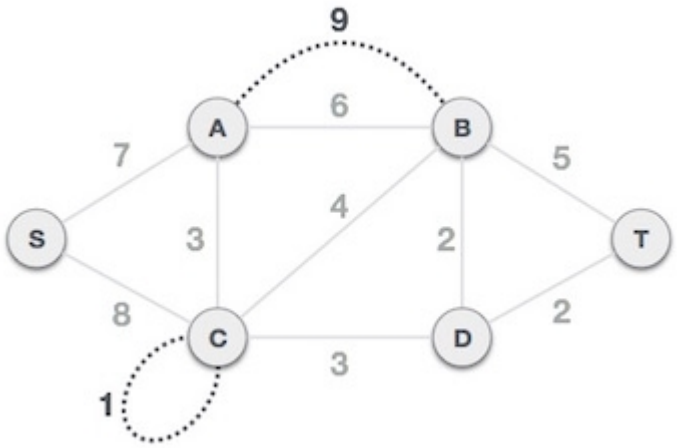
1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

**Example : Apply the Kruskal's algorithm on the graph given as follows.**

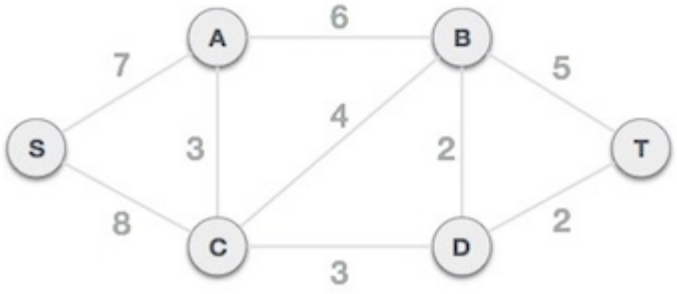



## Step 1 - Remove all loops and Parallel Edges

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.



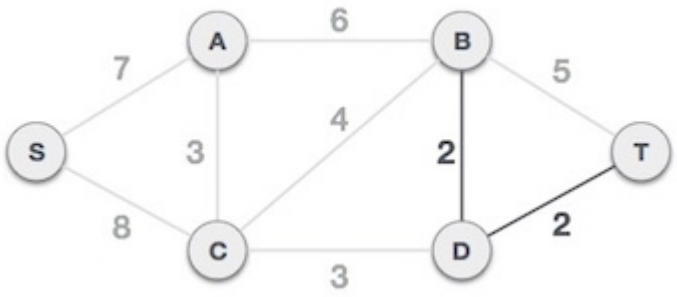

## Step 2 - Arrange all edges in their increasing order of weight

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 |

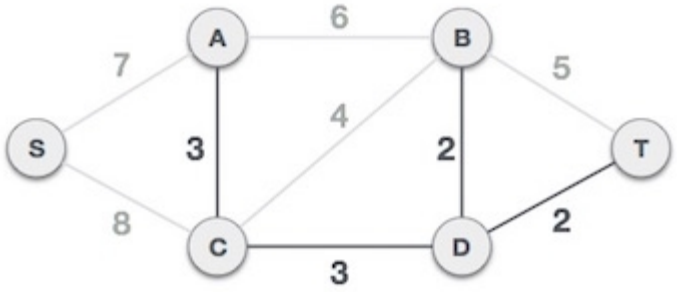
## Step 3 - Add the edge which has the least weightage

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.

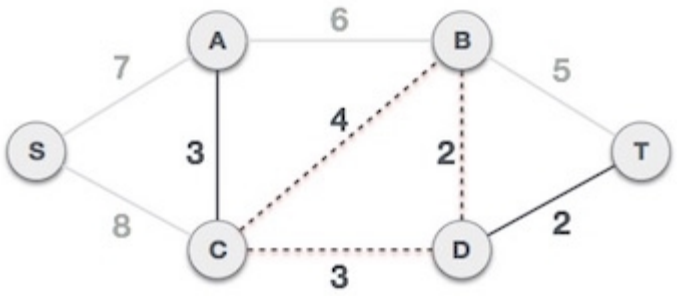



The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

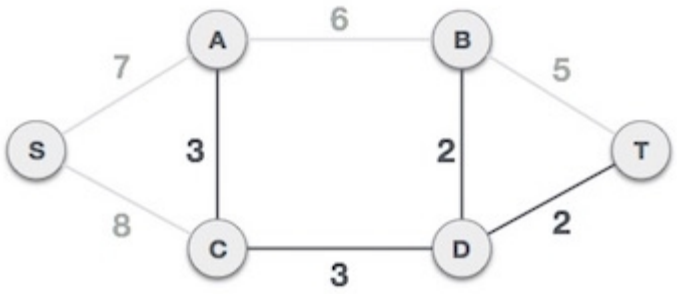
Next cost is 3, and associated edges are A,C and C,D. We add them again −



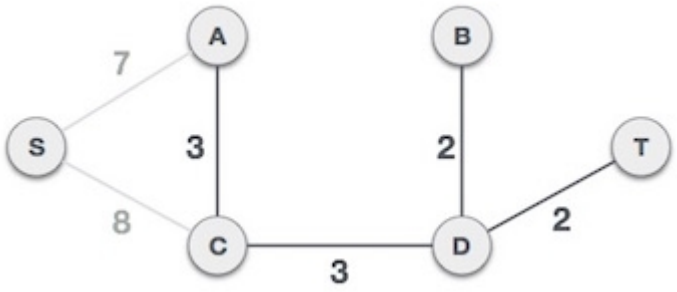

Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



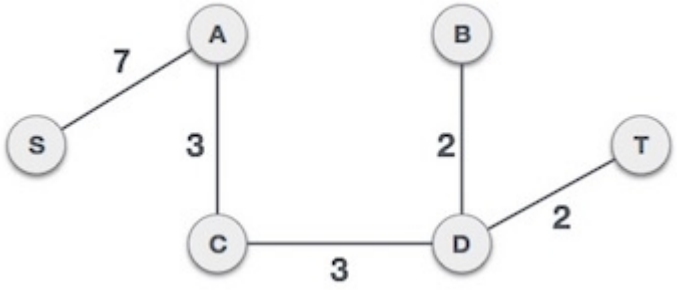

We ignore it. In the process we shall ignore/avoid all edges that create a circuit.




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



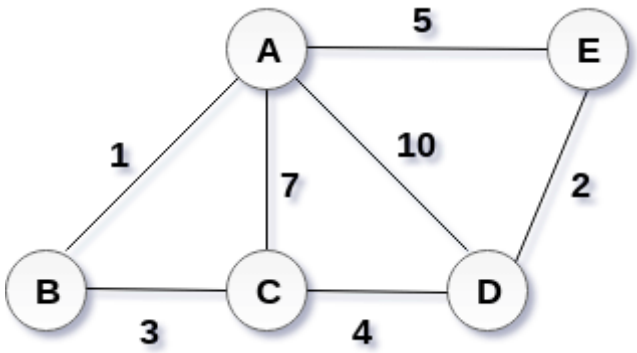

Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.




Hence, the final MST is the one which is shown in the step 4.

The cost of MST = 7+3+3+2+2 = 17.

**Example 2 : Apply the Kruskal's algorithm on the graph given as follows.**




**Solution:**

The weight of the edges given as :

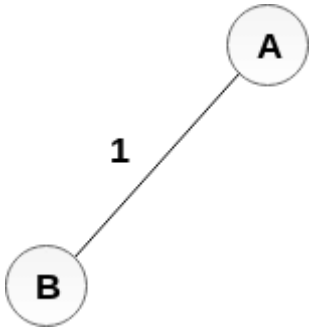
| Edge | AE | AD | AC | AB | BC | CD | DE |
|---|---|---|---|---|---|---|---|
| Weight | 5 | 10 | 7 | 1 | 3 | 4 | 2 |

Sort the edges according to their weights.

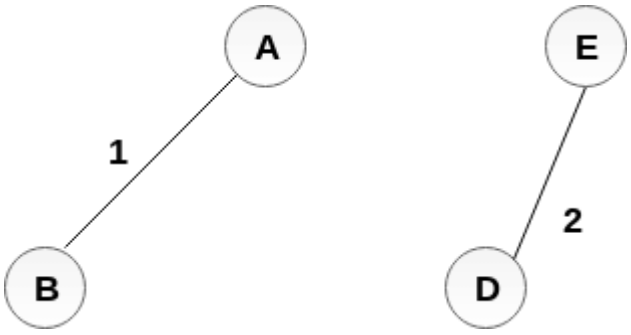
| Edge | AB | DE | BC | CD | AE | AC | AD |
|---|---|---|---|---|---|---|---|
| Weight | 1 | 2 | 3 | 4 | 5 | 7 | 10 |

Start constructing the tree;

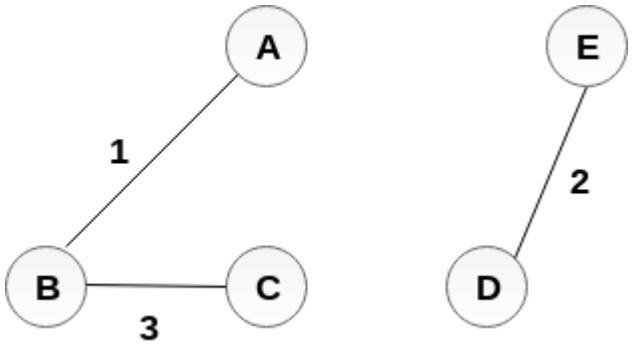
1. Add AB to the MST;



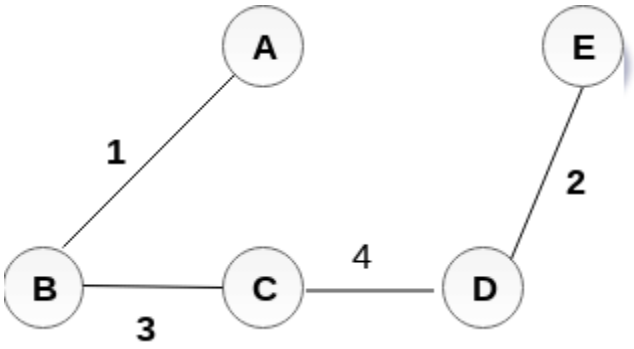

2. Add DE to the MST;




3. Add BC to the MST;



4. Add CD to the MST;




The next step is to add AE, but we can't add that as it will cause a cycle.

The next edge to be added is AC, but it can't be added as it will cause a cycle.

The next edge to be added is AD, but it can't be added as it will contain a cycle.

Hence, the final MST is the one which is shown in the step 4.

The cost of MST = 1 + 2 + 3 + 4 = 10.

- Both **Prim's and Kruskal's algorithm** finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few **major differences between them**.

| PRIM'S ALGORITHM | KRUSKAL'S ALGORITHM |
|---|---|
| It starts to build the Minimum Spanning Tree from any vertex in the graph. | It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph. |
| It traverses one node more than one time to get the minimum distance. | It traverses one node only once. |
| Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to O(E + log V) using Fibonacci heaps. | Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices. |
| Prim's algorithm gives connected component as well as it works only on connected graph. | Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components |
| Prim's algorithm runs faster in dense graphs. | Kruskal's algorithm runs faster in sparse graphs. |

# Shortest path algorithm (SPA) (1)

→ By MST (minimum spanning tree), we are not able to obtain the shortest path between two nodes (source & destination).

→ We can obtain simply minimum cost.

→ But, by using SPA we can obtain minimum dist. blw two nodes.

→ for eg: LAN (Local area network) for all computers

Before designing LAN we should always find out shortest path and obtain economical networking.

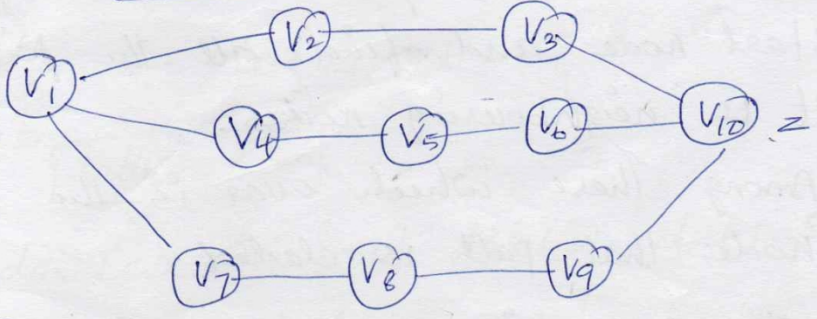
shortest path algorithm categorized into,

(1) weighted SPA

(2) Unweighted SPA.

## 1). Unweighted SPA

↳ gives a path in Unweighted graph which is equal to no. of edges travelled from source to destination




2525
2275
2275
2275
2275

Path b/w a to z

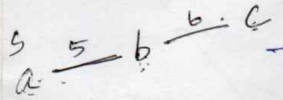
(2)

| S.no. | path | no. of edges |
|---|---|---|
| 1. | $V_1-V_2-V_3-V_{10}$ | 3 |
| 2. | $V_1 V_4 V_5 V_6 V_{10}$ | 4 |
| 3 | $V_1-V_7-V_8-V_9 V_{10}$ | 4. |

→ out of these the path 1 ie $V_1-V_2-V_3-V_{10}$ is shortest one as it consists of only 3 edges from a to z.

2. Dijkstra's shortest path algorithm:

→ used to find shortest path from some source node to some other destination node.




→ source node we start measuring distance – start node.

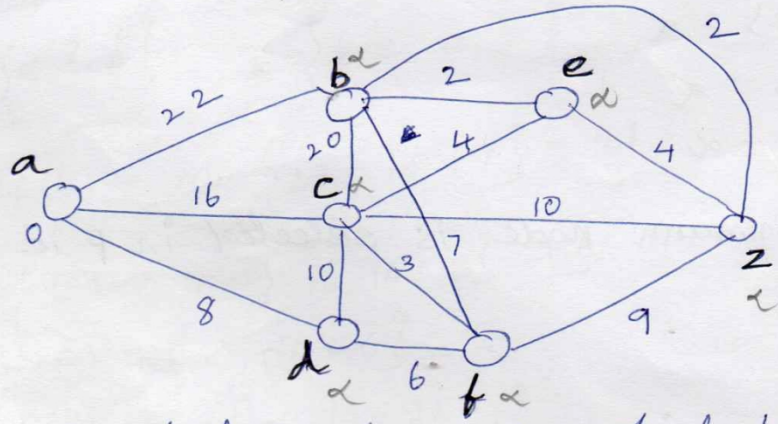
→ destination node – end node

Alg.

1. We start finding the distance from start node and find all the paths from it to neighbouring nodes.
2. Among these which ever is the nearest node that path is selected.
3. This process is repeated until reach the end node. Then whichever is the path that path is called shortest path.

4. all the paths are tried & then choosing the shortest path among them, this alg is solved by a greedy algorithm.
↳ optimum soln

eg:




P = set which is for nodes which have already selected.

T = Remaining node.

a = Source
z = destination

step 1:

V = a (source)

P = {a}, T = {b, c, d, e, f, z}

formula for SPA.

weight or cost

$$dist(b) = \min\{\text{old } dist(b), dist(a) + w(a,b)\}$$

dist blw source to vertex b

$$= \min\{\infty, 0 + 22\}$$

dist(b) = 22

$$dist(c) = \min\{\text{old } dist(c), dist(a) + w(a,c)\}$$

$$= \min\{\infty, 0 + 16\} =$$

dist(c) = 16

$dist(d) = \min\{\text{old } dist(d), dist(a) + w(a,d)\}$

$= \min\{\infty, 0 + 8\}$

$\boxed{dist(d) = 8}$

$dist(e) = \infty$

$dist(f) = \infty$

$dist(z) = \infty$

∴ So minimum node is selected in P ie node d.

step 2

$V = d$

$P = \{a, d\}$, $T = \{b, c, e, f, z\}$

$dist(b) = \min\{\text{old } dist(b), dist(d) + w(b,d)\}$

$= \min\{22, 8 + \infty\}$

$\boxed{dist(b) = 22}$

$dist(c) = \min\{\text{old } dist(c), dist(d) + w(c,d)\}$

$= \min\{16, 22 + 10\}$

$\boxed{dist(c) = 16}$

$dist(e) = \min\{\text{old } dist(e), dist(d) + w(e,d)\}$

$= \min\{\infty, 8 + \infty\}$

$\boxed{dist(e) = \infty}$

$dist(f) = \min\{\text{old } dist(f), dist(d) + w(f,d)\}$

$= \min\{\infty, 8 + 6\} = 14.$

is called shortest path

dist (z) = min{∞, 8+∞} = ∞.

So, min node is selected in P i.e node d

step 3

v = t

P = {a, d, t} , T = {b, c, e, z}

dist(b) = min{22, 14+7} = 21

dist(c) = min{16, 14+3} = 16. min.

dist(e) = min{∞, 14+∞} = ∞

dist(z) = min{∞, 14+9} = 23.

So, min node is c

step 4

v = c

P = {a, d, t, c} , T = {b, e, z}

dist(b) = min{21, 16+20} = 21

dist(e) = min{∞, 16+4} = 20

dist(z) = min{23, 16+10} = 23.

So, min node is e.

step 5

v = e ,

P = {a, d, t, c, e} T = {b, z}

des(b) = {21, 20+2} = 21 ∴ so min node is b

dist(z) = min{23, 20+4} = 23

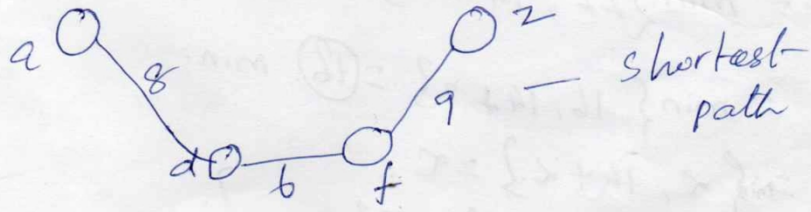
step 6

v = b

P = {a, d, f, c, e, b} , T = {z}

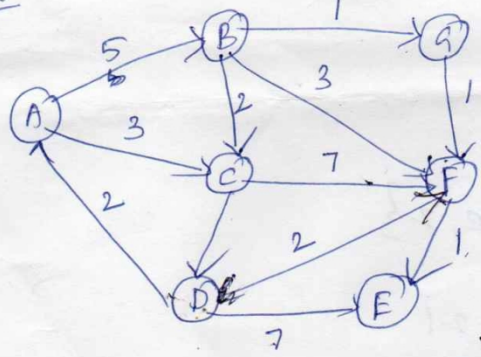
dist(z) = min{23, 21+2} = 23

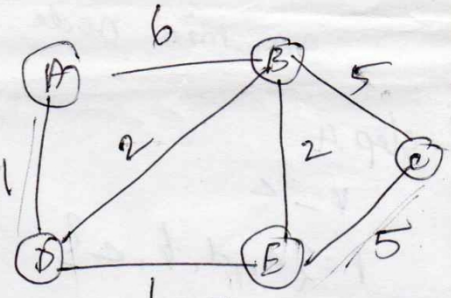
Length of the shortest path from the vertex a to z is 23.




H/W

SPA




Dijkstra's Rule

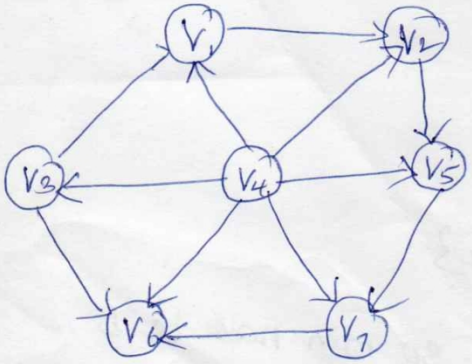
Rule 1 : Make sure there is no -ve edges set dist[v] to source vertex as 0 & set all other distances to infinity

Rule 2 : Relax all vertices adjacent to current vertex

Rule 3 : Choose the closer vertex as next current vertex

Rule 4 : Repeat 2 & 3 until reach the destination.

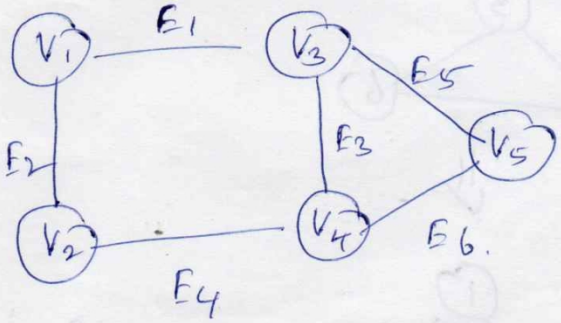
topological sort



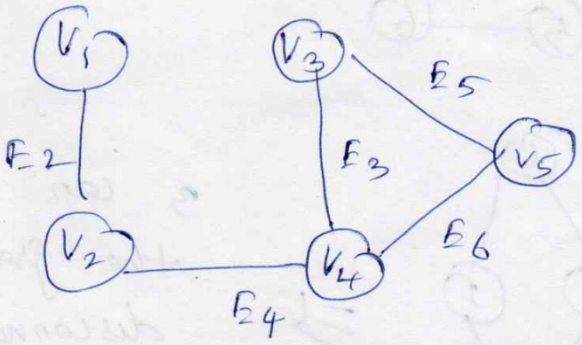
## Biconnectivity:

→ Biconnected graphs are the graphs which can't be broken into two disconnected graphs by connecting single edge. (or) A graph is biconnected if it contains no articulation Point.

eg:

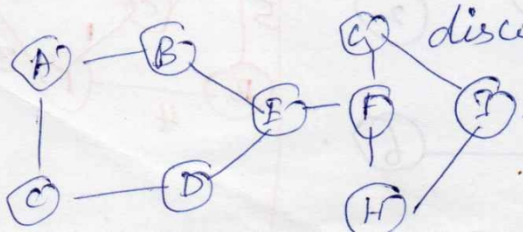



→ if we remove any single edge the graph does't become disconnected (E1 - remove)

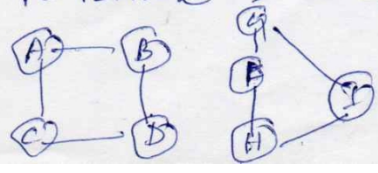



eg:

Articulation pt: is any vertex of a graph whose removal results in disconnected graph.




=>

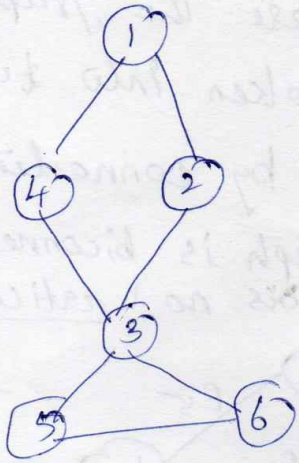



To remove E (graph can be disconnected)

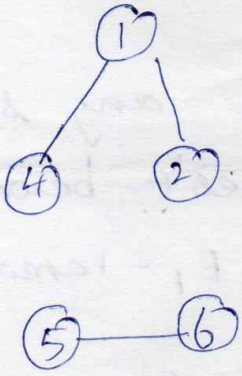
E is AP

eg:

(1)

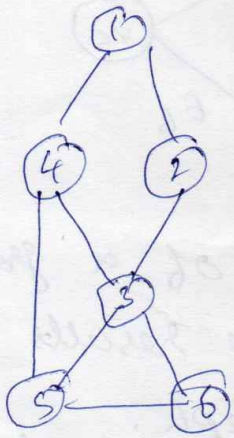



⇒ 3 can be remove the graph can be disconnected.

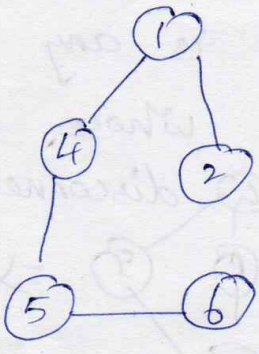



∴ so 3 is articulation point.

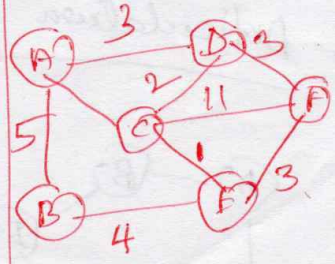
(2).




⇒ 3 can be remove the graph can't be disconnected




MST H/W



## Applications of Graph

- In **Computer science** graphs are used to represent the flow of computation.
- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.
- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm.
- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.