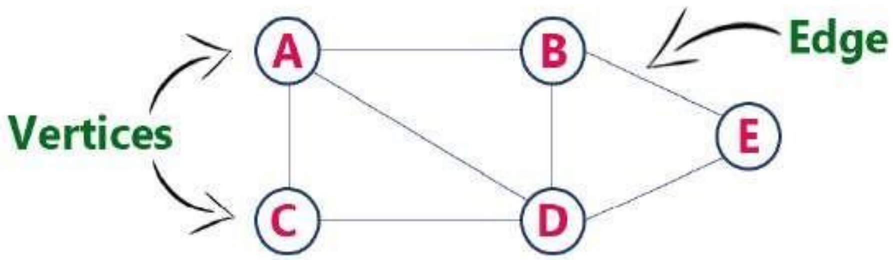
UNIT-IV
GRAPHS: The Graph Abstract Data Type, Introduction, Definition, Graph Representation, Elementary Graph Operation, Depth First Search, Breadth First Search. Connected components, spanning trees. Bi-connected components, Minimum cost spanning trees, Kruskal's algorithm, prims algorithm, Shortest Paths: transitive closure, single source/all destinations, all pairs shortest path

Graph is a data structure which is similar to the tree data structure but it has closed loop where as in tree does not has closed loop

Definition:

A Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is represented as G = (V , E), where V is set of vertices and E is set of edges Example: 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)}. This is a graph with 5 vertices and 6 edges.

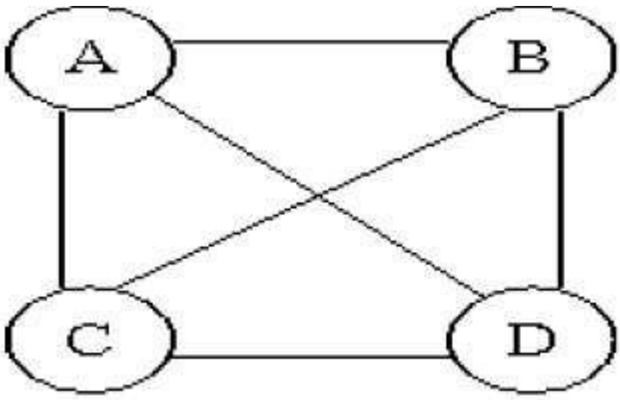


Graph Terminology :

1. **Vertex:** An 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.
2. **Edge** : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex). In above graph, the link between vertices A and B is represented as (A,B). Edges are three types:
1. Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. 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).
3. Weighted Edge - A weighted edge is an edge with cost on it. 3. Weighted Edge - A weighted edge is an edge with cost on it.

Types of Graphs :

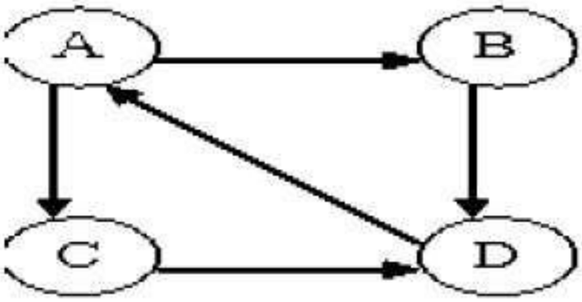
1.Undirected Graph :
A graph with only undirected edges is said to be undirected graph



Undirected Graph.

2. Directed Graph

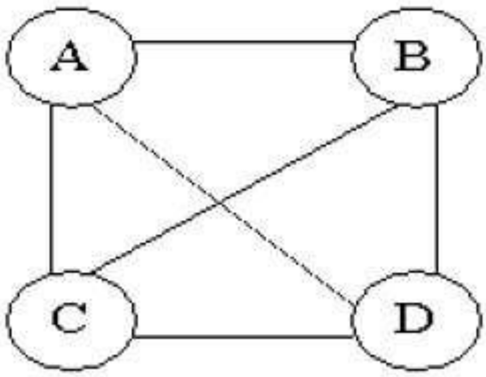
A graph with only directed edges is said to be directed graph.



Directed Graph.

3. Complete Graph

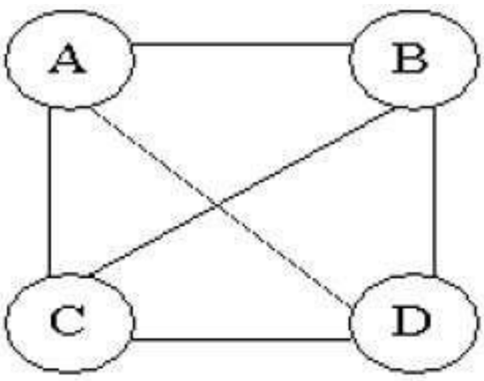
A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

4. Regular Graph

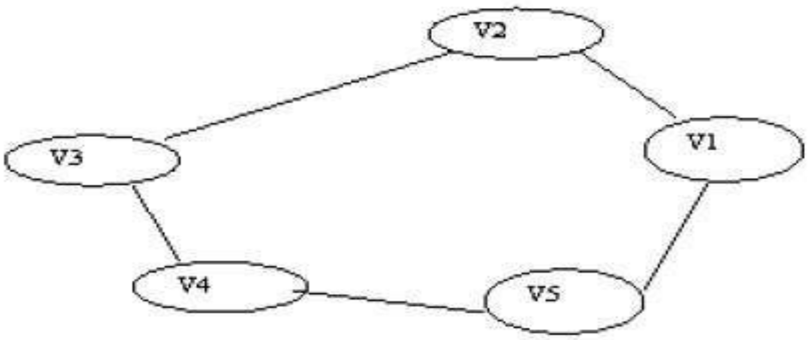
Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



A regular graph

5. Cycle Graph

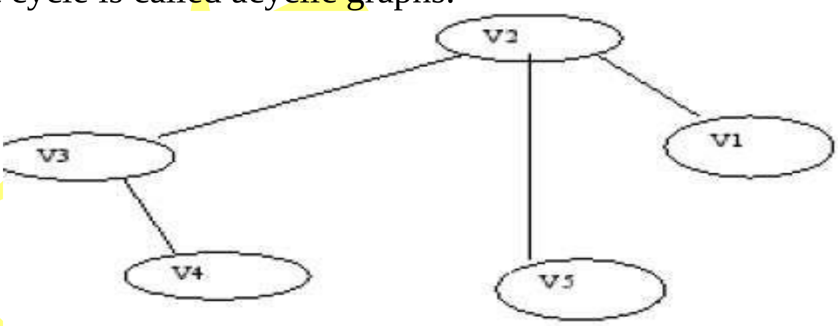
A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



A cycle graph

6. Acyclic Graph

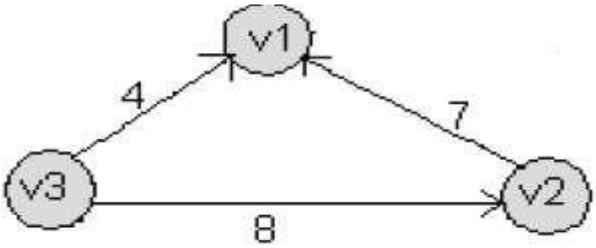
A graph without cycle is called acyclic graphs.



A acyclic graph

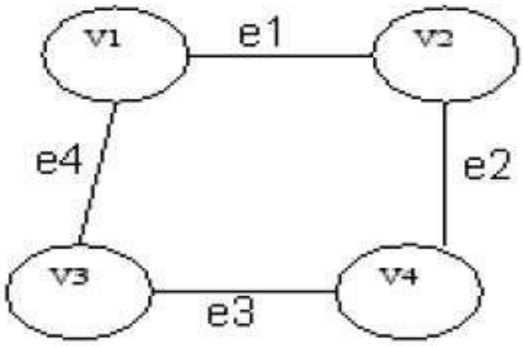
7. Weighted Graph

A graph is said to be weighted if there are some non-negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



A weighted graph

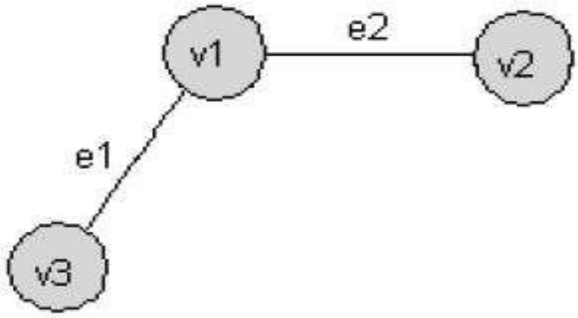
Outgoing Edge
A directed edge is said to be outgoing edge on its orign vertex.
Incoming Edge
A directed edge is said to be incoming edge on its destination vertex.
Degree
Total number of edges connected to a vertex is said to be degree of that vertex.
Indegree
Total number of incoming edges connected 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.
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.
Adjacent nodes
When there is an edge from one node to another then these nodes are called adjacent nodes.
Incidence
In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.
Walk
A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.
Closed walk
A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk.



If e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2 e2 v4 e3 v3 e4 v1 be its closed walk or circuit.

Path

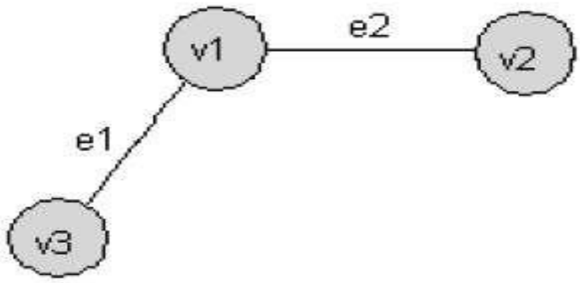
A open walk in which no vertex appears more than once is called a path.



If e1 and e2 be the two edges between the pair of vertices (v1,v3) and (v1,v2) respectively, then v3 e1 v1 e2 v2 be its path.

Length of a path

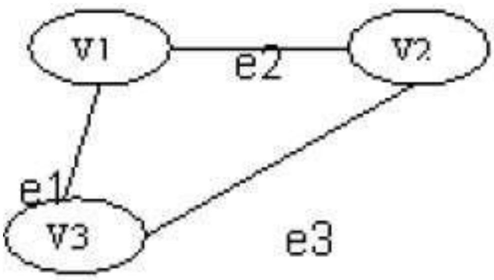
The number of edges in a path is called the length of that path. In the following, the length of the path is 3.



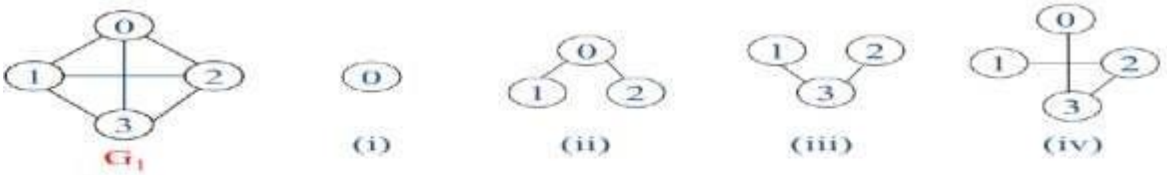
An open walk Graph

Circuit

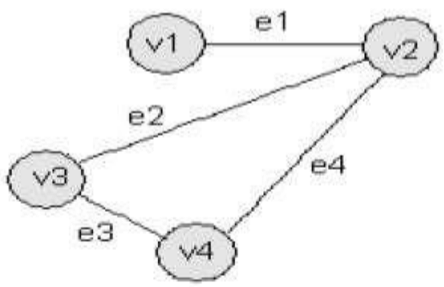
A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit. A circuit having three vertices and three edges.



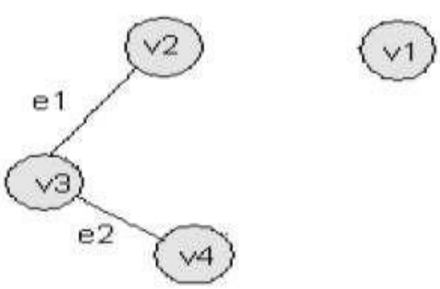
Sub Graph
A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of S has the same end vertices in S as in G. A subgraph of G is a graph G’ such that V(G’) □ V(G) and E(G’) □ E(G)



Connected Graph
A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise is disconnected.



A connected graph G

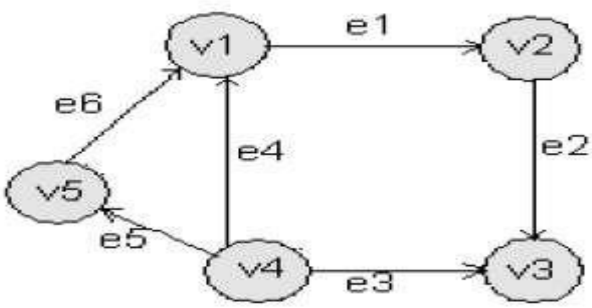


A disconnected graph G

This graph is disconnected because the vertex v1 is not connected with the other vertices of the graph.

Degree
In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree of a node is the number of edges incident on it.
In the above graph, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2 in a connected graph.

Indegree
The indegree of a node is the number of edges connecting to that node or in other words edges incident to it



Out degree
The out degree of a node (or vertex) is the number of edges going outside from that node

ADT of Graph:

Structure Graph is

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

functions:

Graph Create (): =return an empty graph
Graph Insert Vertex (*graph*, *v*)::= return a graph with *v* inserted. *v* has no edge.
Graph Insert Edge (*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*
Graph Delete Vertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed
Graph Delete Edge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed *Boolean*
Is Empty(*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

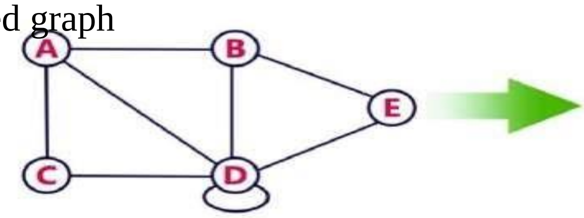
1. Adjacency Matrix
2. Adjacency List
3. Incidence Matrix

1. Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

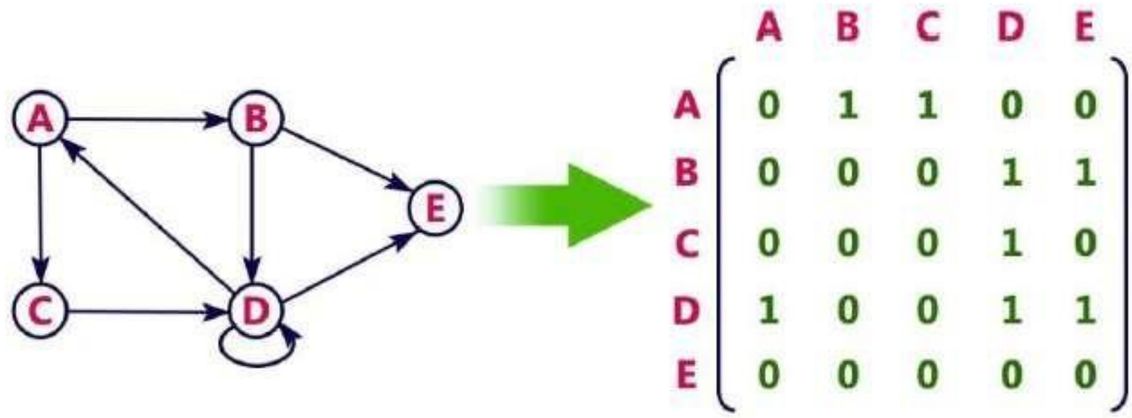
- In this matrix, rows and columns both represent vertices.
- This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.
- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j and otherwise adj[i][j] = 0 indicates that there is no edge from vertex I to vertex j.

example: for undirected graph



$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \left[\begin{array}{ccccc} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{array}\right] \end{array}$$

For a Directed graph

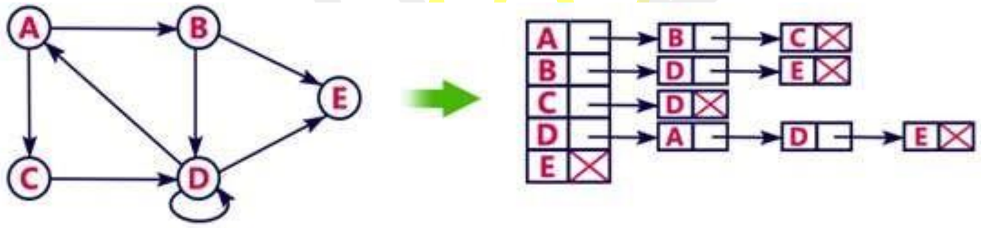


The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric.

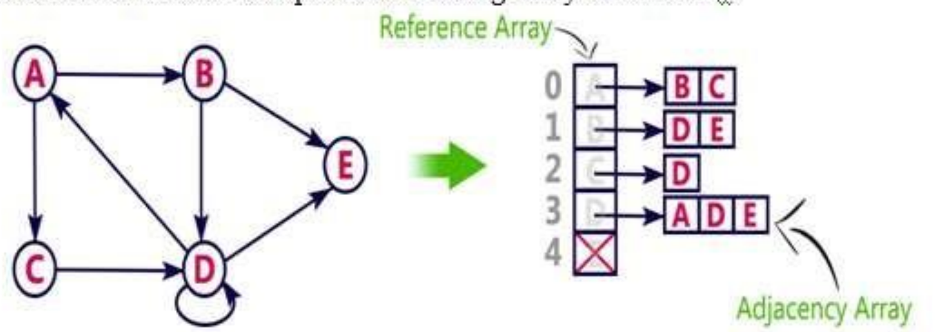
2. Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains.

- The nodes in chain I represent the vertices that are adjacent to vertex I.
- It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies.
- Example: consider the following directed graph representation implemented using linked list...



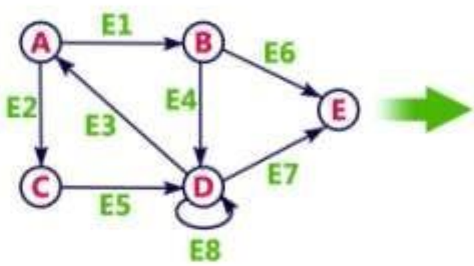
This representation can also be implemented using array as follows..



3. 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 4X6 class. In this matrix, rows represent vertices and columns represent 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.

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

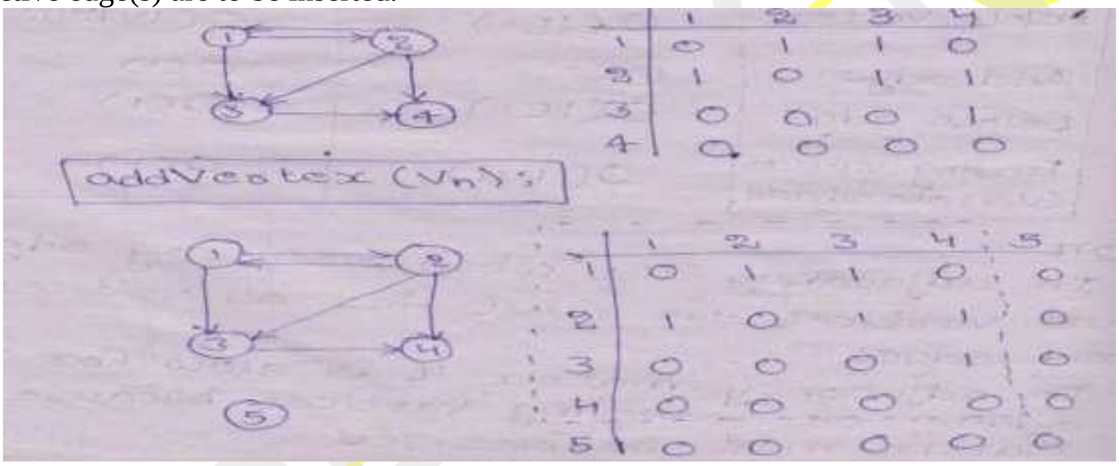


	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	0	-1	-1	0

Graph Operations

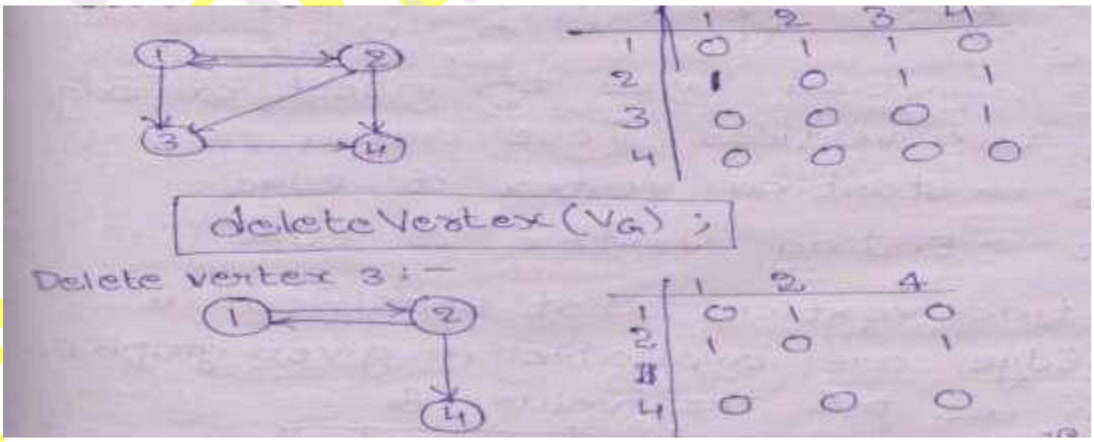
1.Insert Vertex:

The insert vertex operation inserts a new vertex into a graph and returns the modified graph. When the vertex is added, it is isolated as it is not connected to any of the vertices.
in the graph through an edge. If the added vertex is related with one (or more) vertices in the graph, then the respective edge(s) are to be inserted.



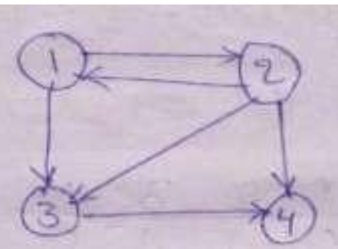
2.Delete Vertex:

The delete vertex operation deletes a vertex and all the incident edges on that vertex and return the modified graph.



3.Insert an Edge:

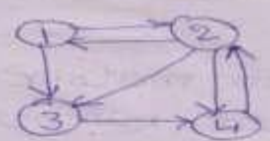
The insert edge operation adds an edge incident between two vertices. In an undirected graph, for adding an edge, the two vertices u and v are to be specified, and for a directed graph along with vertices, the start vertex and the end vertex should be known.



	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	0	0	0

If two vertices that specified in addEdge are available in given graph G then we put the value

$\boxed{G[V_s][V_e] = \text{cost of edge}}$



$G[4][2] = 1$

	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	0	0	0	1
4	0	1	0	0

4. Delete an Edge:

The delete edge operation removes one edge from the graph. Let the graph G be G(V, E). Now, deleting the edge (u, v) from G deletes the edge incident between vertices u and v ad keeps the incident vertices u, v.

Graph Traversal Techniques

Definition: - To solve many problems modeled with graphs, we need to visit all the vertices and edges in a systematic fashion called graph traversal. We shall study two types graph traversal techniques.

1. Depth first traversal (DFS)
2. Breadth first traversal. (BFS)

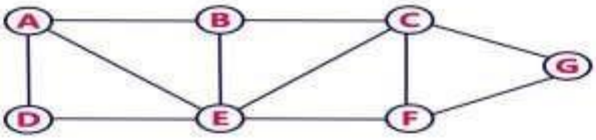
Breadth first traversal. (BFS):

In BFS all the unvisited vertices adjacent to i are visited after visiting the start vertex i and making it visited. Next, the unvisited vertices adjacent to these vertices are visited and so on until the entire graph has been traversed. This approach is called “breadth-first” because from the vertex i that we visit, we search as broadly as possible by next visiting all the vertices adjacent to i. This search algorithm uses a queue data structure to store the vertices of each level of the graph as and when they are visited. These vertices are then taken out from the queue in sequence, that is, first in first out (FIFO), and their adjacent vertices are visited until all the vertices have been visited. The algorithm terminates when the queue is empty.

Algorithm:

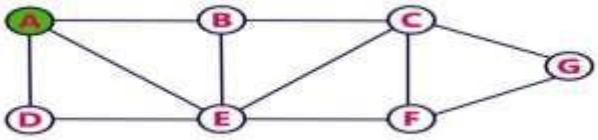
1. Define a queue of size as total number of vertices in the graph
2. Select any vertex as starting point for traversal and insert it into the queue
3. Visit that vertex which is at the front of the queue and delete it from the queue and place its all unvisited adjacent nodes in the queue.
4. repeat step 3 untill queue becomes empty
5. stop

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.

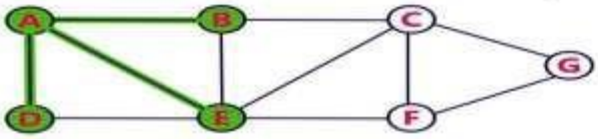


Queue

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..

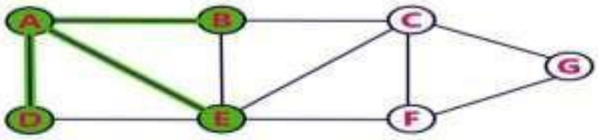


Queue

	D	E	B			

Step 3:

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

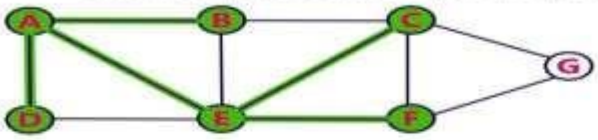


Queue

		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.

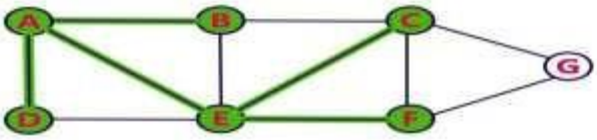


Queue

			B	C	F	

Step 5:

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

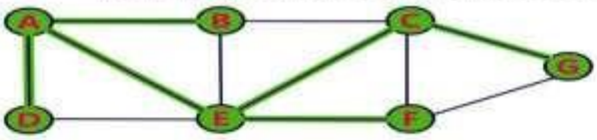


Queue

				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.

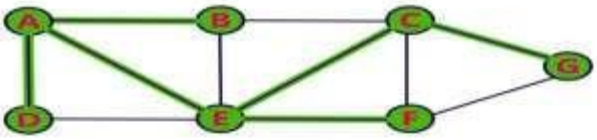


Queue

					F	G

Step 7:

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

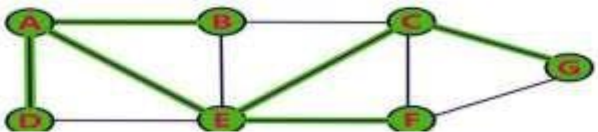


Queue

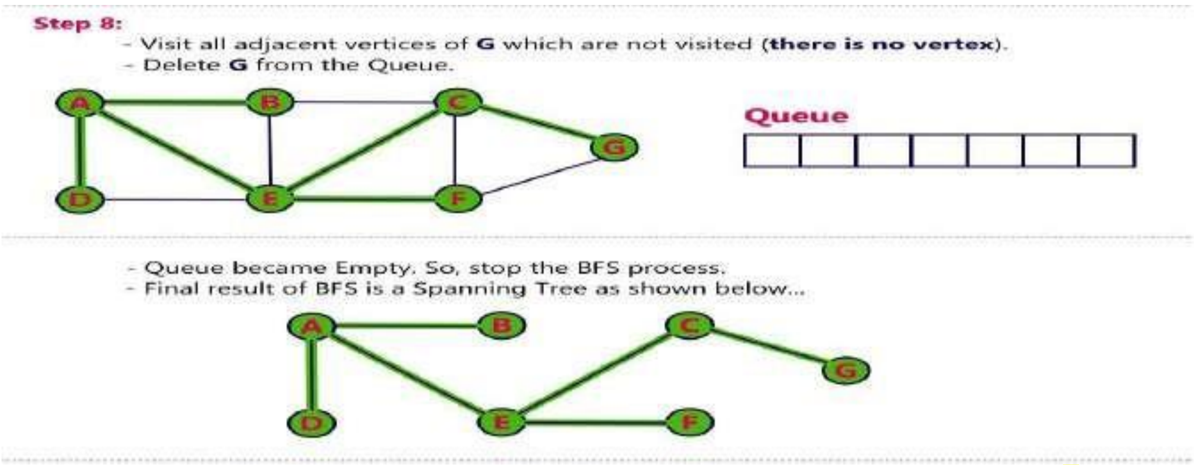
						G

Step 8:

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



Queue



Depth First Search (DFS): In DFS, as the name indicates, from the currently visited vertex in the graph, we keep searching deeper whenever possible. All the vertices are visited by processing a vertex and its descendents before processing its adjacent vertices. This procedure can be written either recursively or non-recursively. For recursive code, the internal stack would be used, and for non-recursive code, we would use a stack.

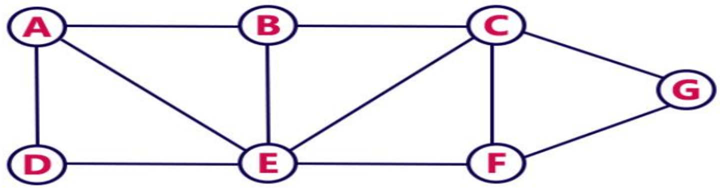
ALGORITHM:

1. start
2. define a stack of size as total number of vertices in the graph
3. select any vertex as starting point for traversal.visit that vertex and push it onto the stack and make status as visited
4. isit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it onto the stack.
5. repeat step 4 untill there is no new vertex to be visited from the vertex which is at the top of the stack
6. when there is no new vertex to visit then use back tracing and pop one vertex from the stack.
7. repeat steps 3,4 and 5 untill stack becomes empty.

8. when stack becomes empty ,then produce final spanning tree by removing unused edges from the graph.
9. stop.

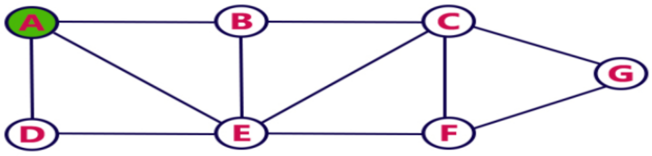
Example

Consider the following example graph to perform DFS traversal



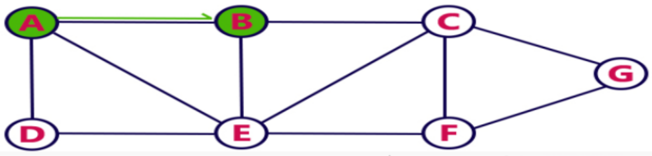
Step 1:

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



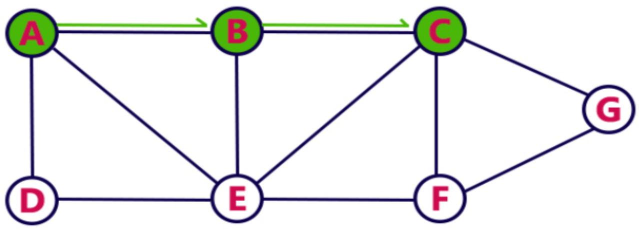
Step 2:

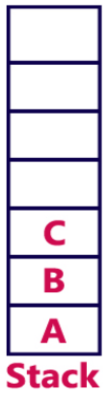
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

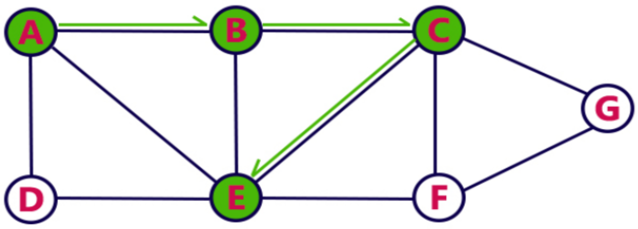
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

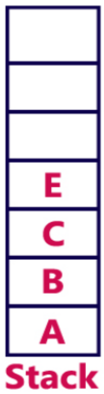




Step 4:

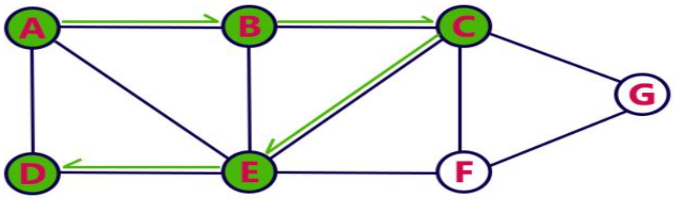
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

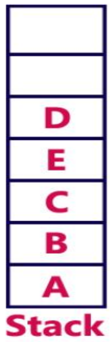




Step 5:

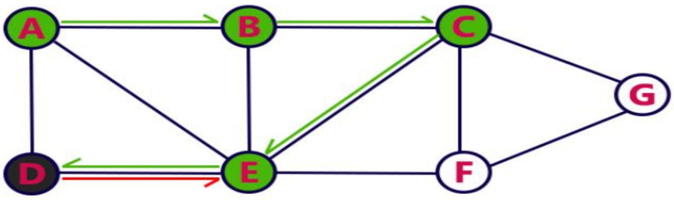
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

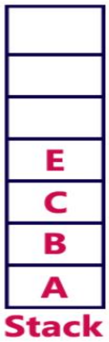




Step 6:

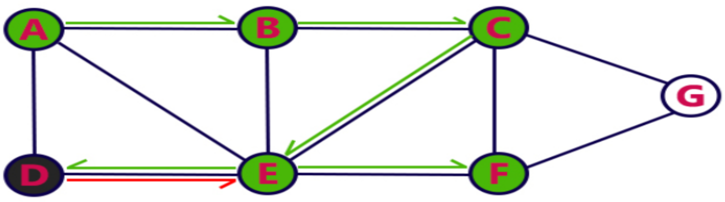
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

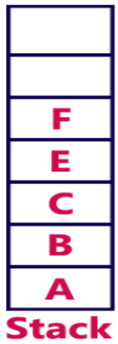




Step 7:

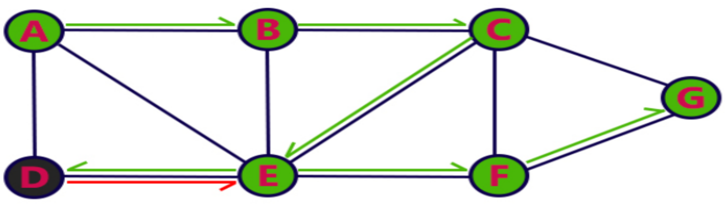
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

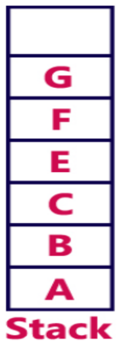




Step 8:

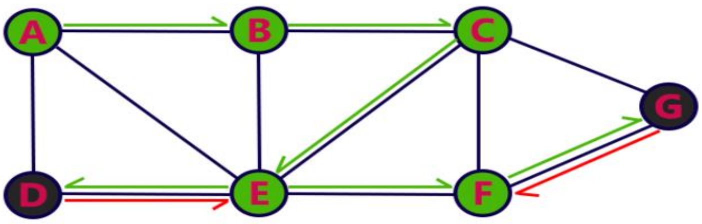
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

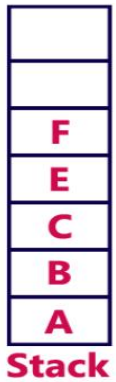




Step 9:

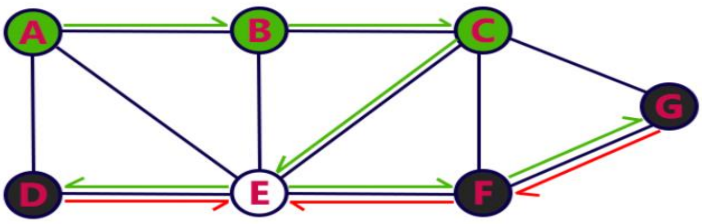
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

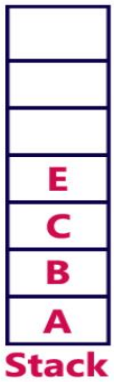




Step 10:

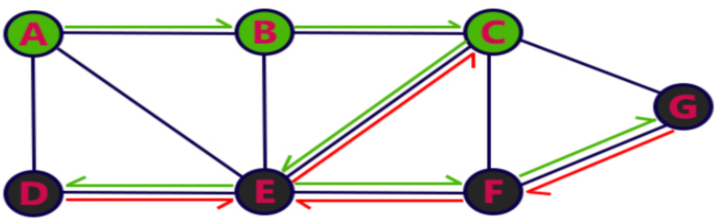
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

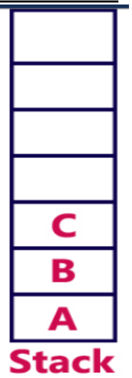




Step 11:

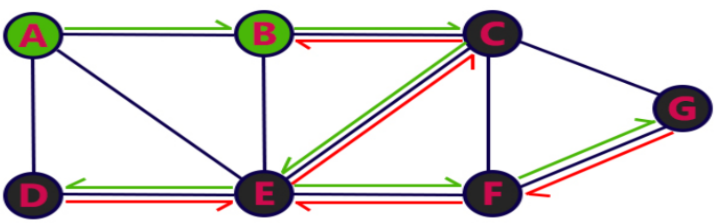
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

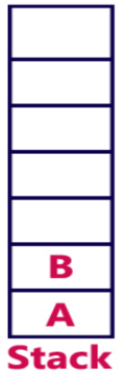




Step 12:

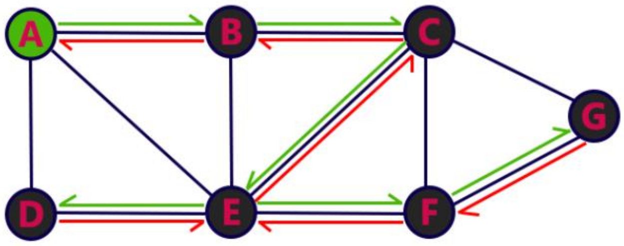
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

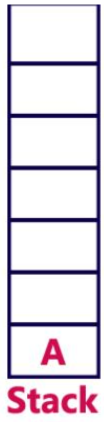




Step 13:

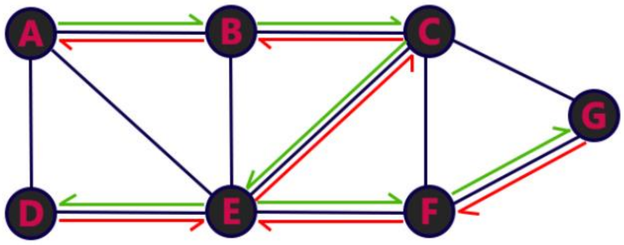
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

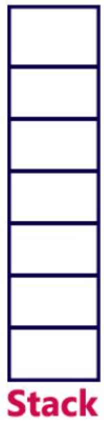




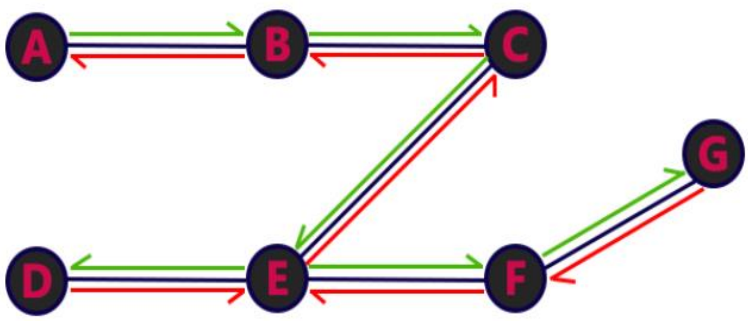
Step 14:

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

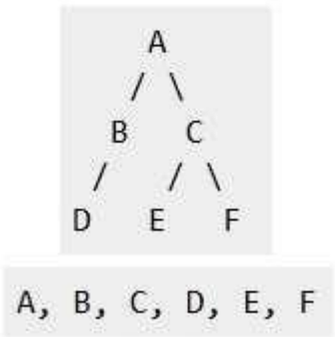
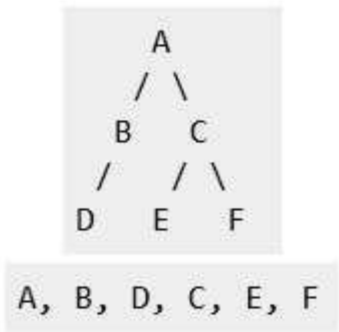




- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

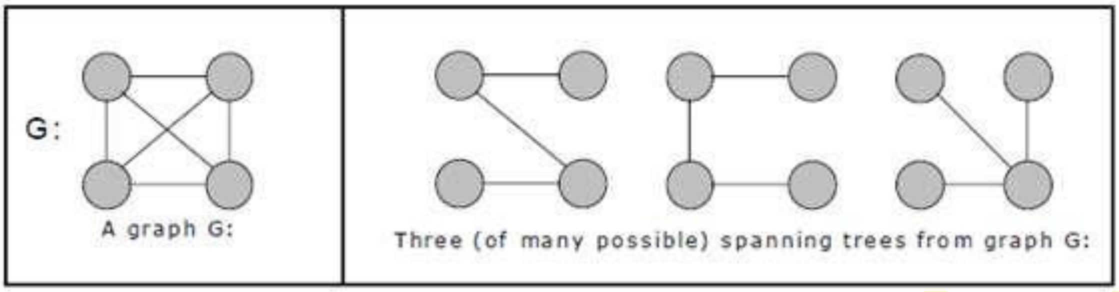


Difference between BFS and DFS :

BFS	DFS
BFS Stands for "**Breadth First Search**".	**DFS** stands for "**Depth First Search**".
BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.	DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise.
Breadth First Search can be done with the help of **queue** i.e. **FIFO** implementation.	Depth First Search can be done with the help of **Stack** i.e. **LIFO** implementations.
This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.	This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off.
BFS is **slower** than DFS.	DFS is more **faster** than BFS.
BFS requires **more** memory compare to DFS.	DFS require **less** memory compare to BFS.
Applications of BFS > To find Shortest path > Single Source & All pairs shortest paths > In Spanning tree > In Connectivity	**Applications of DFS** > Useful in Cycle detection > In Connectivity testing > Finding a path between V and W in the graph. > Useful in finding spanning trees & forest.
BFS is useful in finding shortest path.BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph.	DFS in not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (**backtrack**) to add branches also as long as possible.
Example: 	Example: 

Spanning Tree (ST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

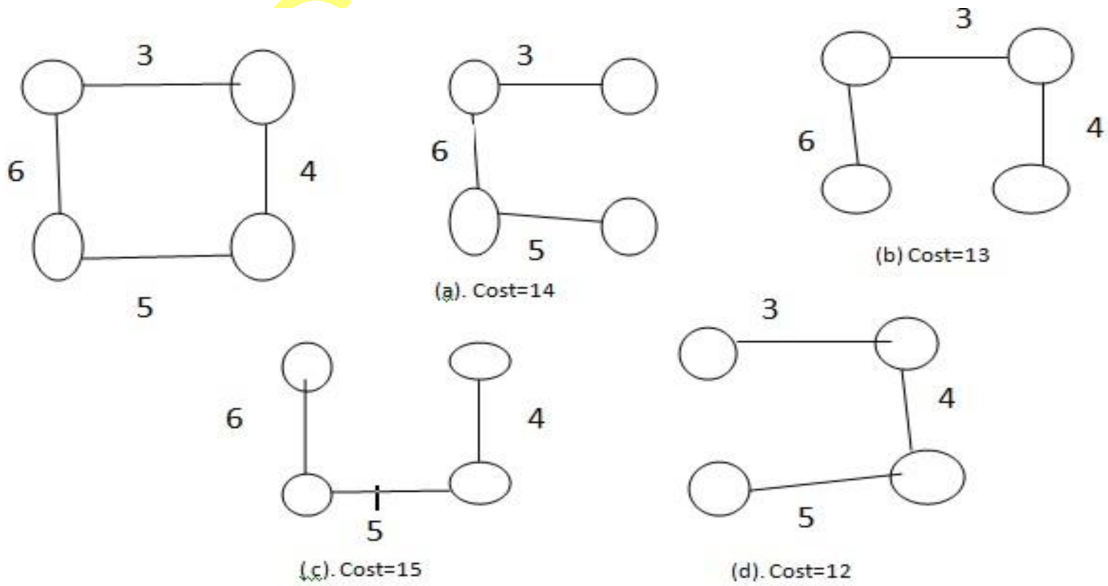


- A spanning tree T must satisfy four properties:
  1. The tree T should contain all the vertices of a given graph
  2. If given graph contain n vertices then the tree T contains (n-1) edges
  3. The spanning tree T should not contain any cycle.
  4. If any edge is removed from the tree T then tree becomes disconnected.

Minimum Spanning Tree (MST):

- For a weighted graph we are construct the minimum spanning tree.

A minimum spanning tree is a spanning tree in which sum of the weights associated with all edges is minimum. which means a spanning tree with minimum weight or cost among other spanning trees.



From above figure, (d) figure has the minimum cost so, (d) is the minimum cost spanning tree.

Let's consider a couple of real-world examples on minimum spanning tree:

- One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost
- Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

- **Minimum spanning tree, can be constructed using any of the following two algorithms:**

  1. Kruskal's algorithm and

  2. **Prim's algorithm**.

  Both are used for MST but both follow different methodology. Kruskal's uses edges ,prims uses vertex connections in determines the MST.

Kruskal's Algorithm:

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest(least cost) edge, while avoiding the creation of cycles, until (n - edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MST's may result, but they will all have the same total cost, which will always be the minimum cost.

ALGORITHM:

1.start

2.list all the edges of G in order of their weights.

3.choose an edge (u,v)with minimum weight from all the edges

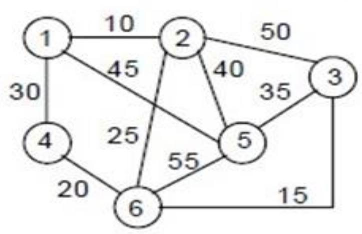
4.at each stage select an edge of minimum weight from all the remaining edges of G if it does not form a cycle.

4.repeat until (n-1) edges have been selected when n is the number of vertices in G.

5. stop

EXAMPLE:

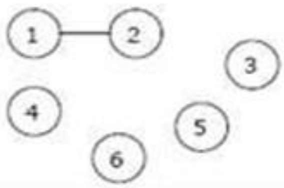
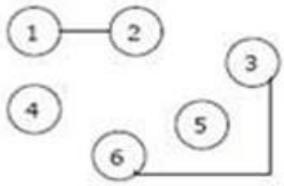
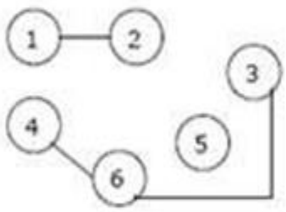
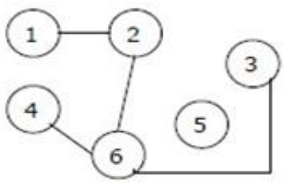
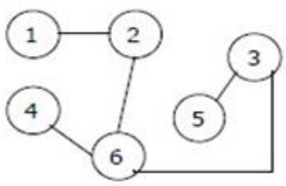
Construct the minimal spanning tree for the graph shown below:



Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

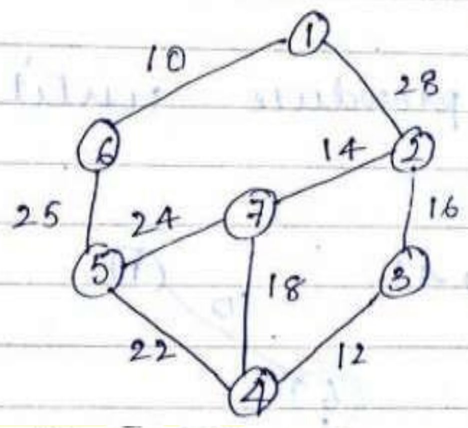
Prim’s algorithm:

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.

Algorithm:

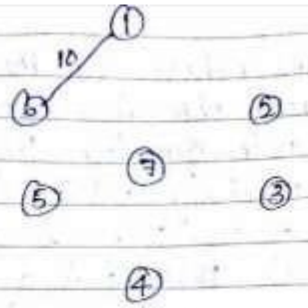
1. start
2. Start with any vertex v of a given graph
3. Find the edges associated with that vertex and add minimum cost edge to the spanning tree
4. At each stage choose an edge of minimum weight joining it to a vertex already include in a tree if it is not forming any cycle.
5. Repeate step 3 until all the vertices of G are included.
6. stop

Let us consider the foll graph :-

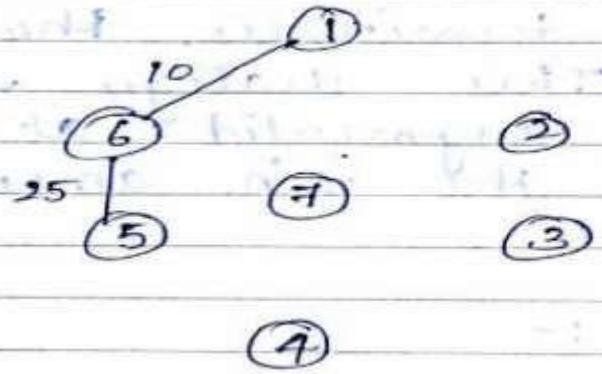


Step 1: Initially all vertices are unvisited i.e S is empty. Assume starting vertex as 1. i.e S = {1}.

The edges incident from vertex 1 are (1,2) (1,6) among which the min cost edge is (1,6). So Include (1,6)

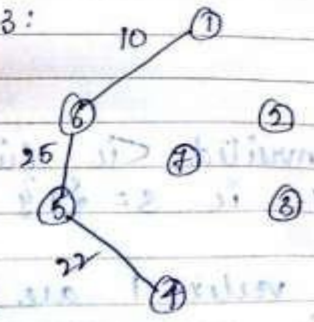


Step 2: Now S contains two vertices 1, 6 ie {1, 6}. The edges incident from these vertices are (1,2) & (1,5) among which (6,5) is min cost edge. Include (6,5) into min spanning tree.

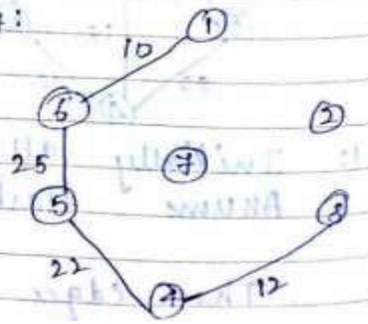


Now S = {1, 6, 5} Repeat above procedure until spanning tree contains (n-1) edges.

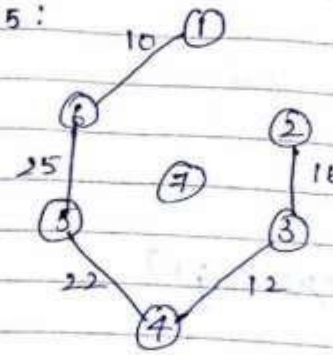
Step 3:



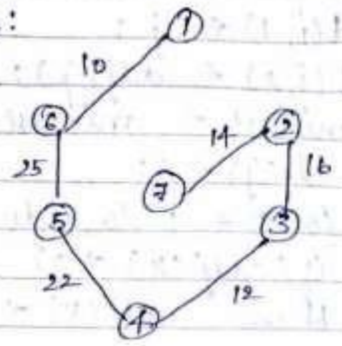
Step 4:



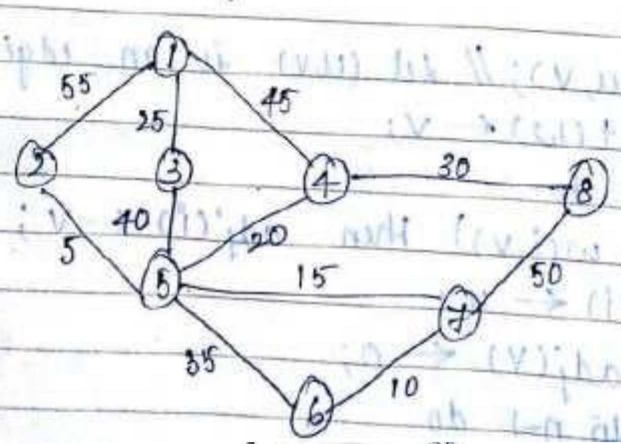
Step 5:



Step 6:



Prob: ~~Select~~ Show the step-by-step procedure the min cost spanning tree using Prim's & Kruskal's method on the foll graph:

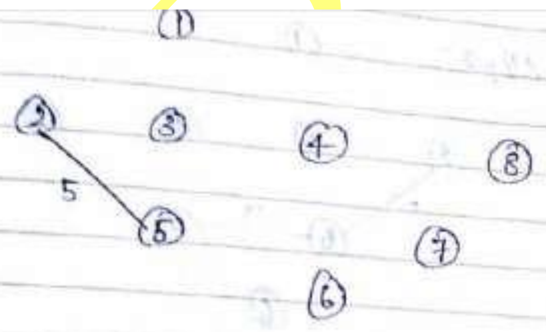


Step 1:

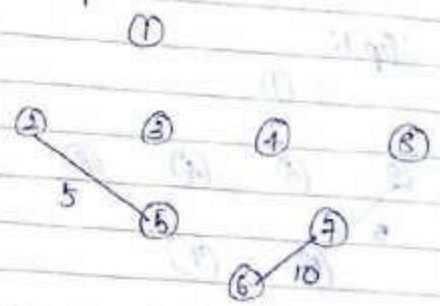
{3,5}

∞

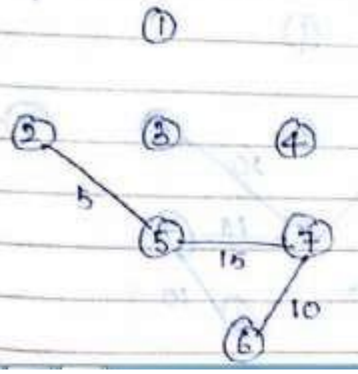
40



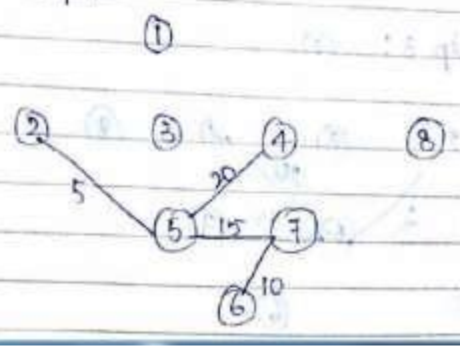
Step 2:



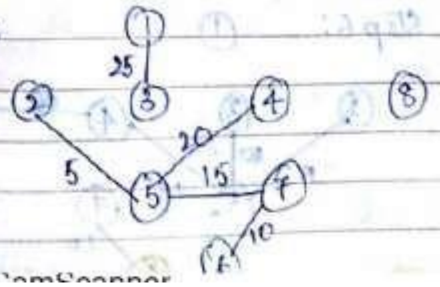
Step 3:



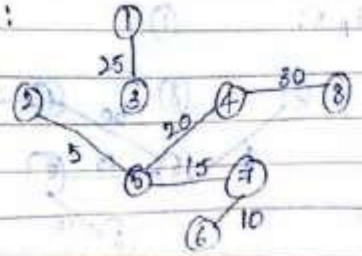
Step 4:



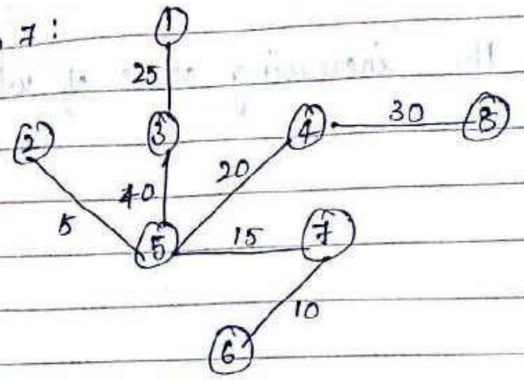
Step 5:



Step 6:



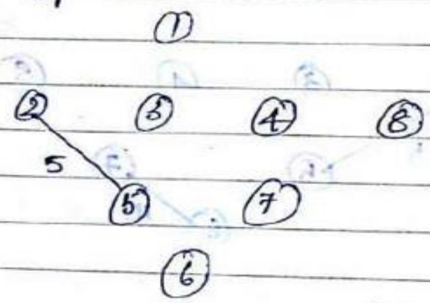
Step 7:



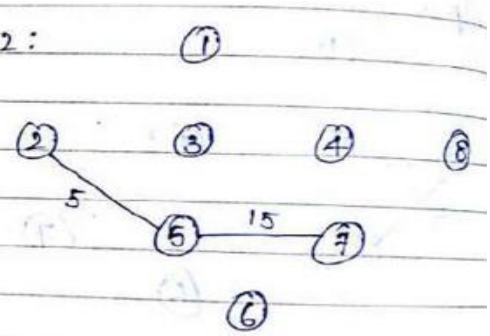
Total wt = 5 + 10 + 15 + 20 + 25 + 30 + 40 = 145

Prim's Method :-

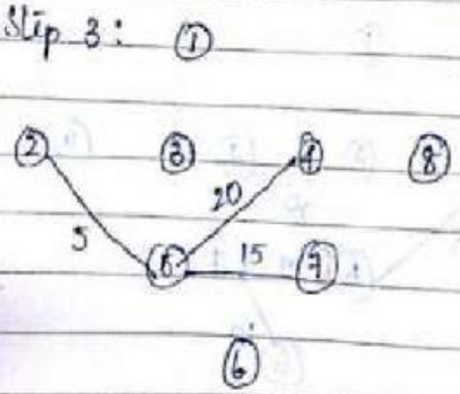
Step 1:



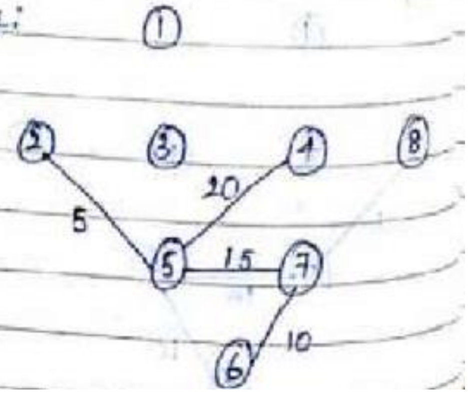
Step 2:



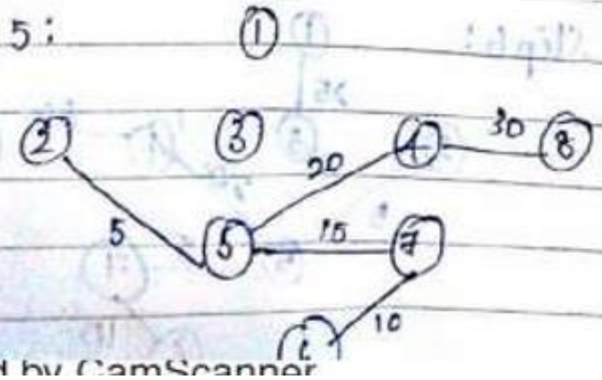
Step 3:



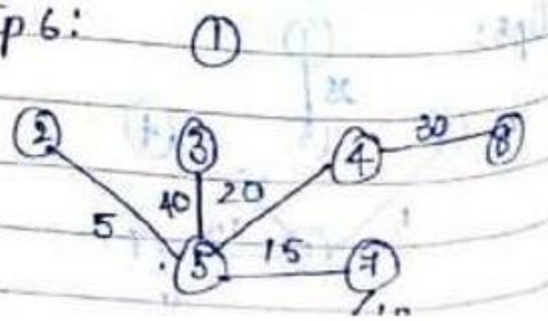
Step 4:



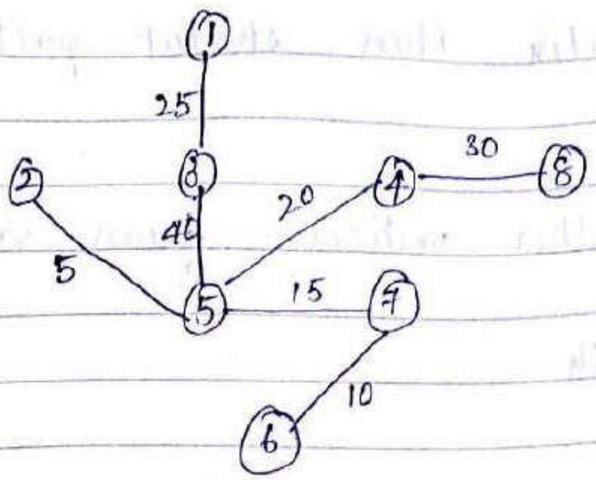
Step 5:



Step 6:



Step 7:

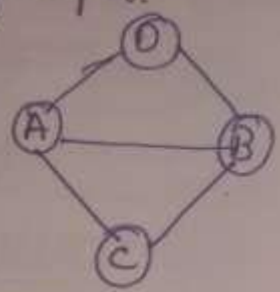


$\therefore$ Total wt $= 5 + 15 + 20 + 30 + 40 + 25 = 145$

Connected Components:

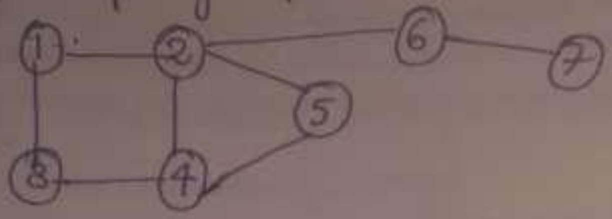
→ connected component is one of the application of DFS.

→ If there is a path between any two vertices then that graph is called connected graph.



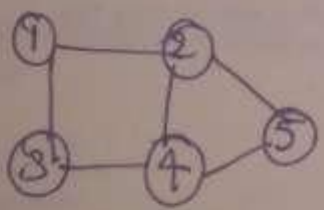
Connected component is a sub graph in which any two vertices are connected by paths and no vertices are connected with any other vertices in the super graph.

Example: let us conceder a graph as a supergraph.

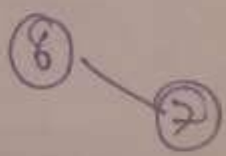


In this graph every node is connected with each other so, this graph have one connected component.

→ now delete edge (2,6). then the graph is divid into two components



component ①



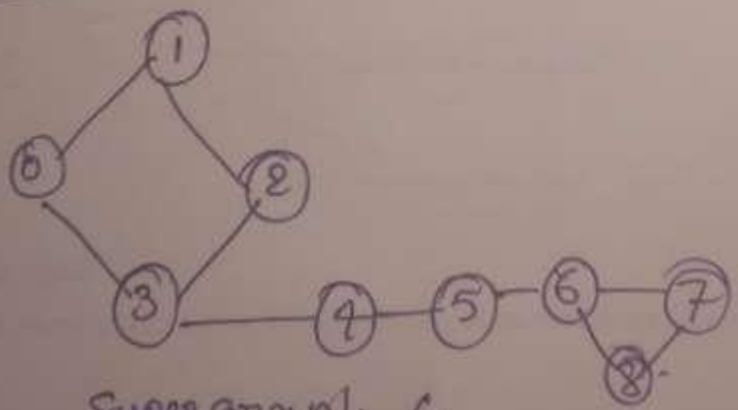
component ②

→ In component ① has 5 nodes are connected each other so this one is one connected component.

→ and the component ② also has two nodes and connected to each other. So, this is also one of the connected component.

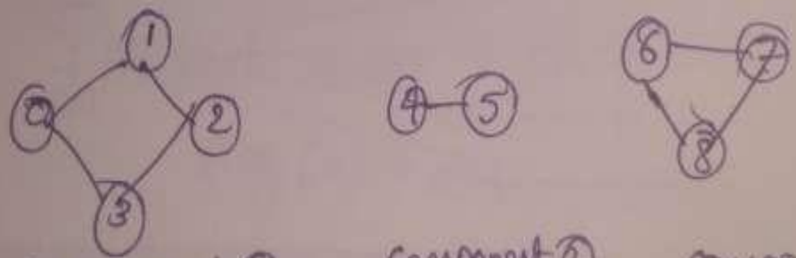
→ Both component ① & ② are connected components only because for each component according to first condition, every vertex are connected with paths. and according to second condition, no vertices are connected with any other vertex of super graph.

Example 2:



Supergraph G.

• delete edges (3, 4) & (5, 6) then the super graph divid into three componentents



component 1 component 2 component 3

all three components 1, 2 & 3 connected components.

Algorithm:

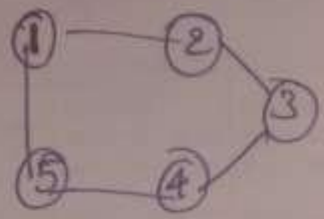
```
connected_component (G)
begin
    for each vertex v ∈ N
        flag [v] = -1;
        count = 0;
  for (int v=0; v<N; v++)
    begin
        if (flag[v] = = -1)
        begin
         DFS (v, flag)
          count ++;
        end
  end
    display count.
end

DFS (int v, int flag)
  begin
   flag[v] = 1;
   display visited vertex.
```

```
for each adjacent node u of v
        if ( flag[u] == -1)
         begin
            DFS(u, flag);
        end
end.
```

Biconnected graph:

a graph G is Biconnected if it contains NO Articulation point

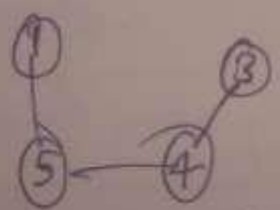
* In Biconnected graph, two distect paths connect each pair of vertices



this graph contains no AP

So, this is a Biconnected graph

* suppose remove vertex 2 still it is connected so, it is a Biconnected graph



* a graph that is not biconnected devides into Biconnected components

⇒ Bi Connected components:

A maximal Biconnected subgraph is a Biconnected components.

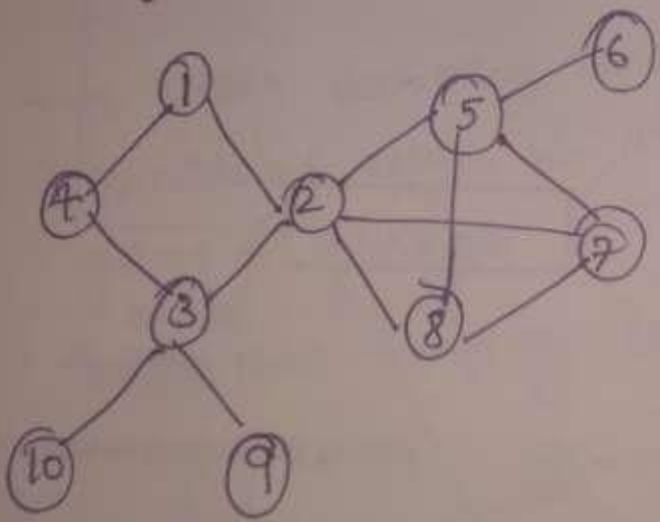
<u>Note</u>: two Biconnected components can have atmost one vertex is common and that vertex is an articulation point.

<u>Articulation point (AP)</u>

AP also known as cut vertex

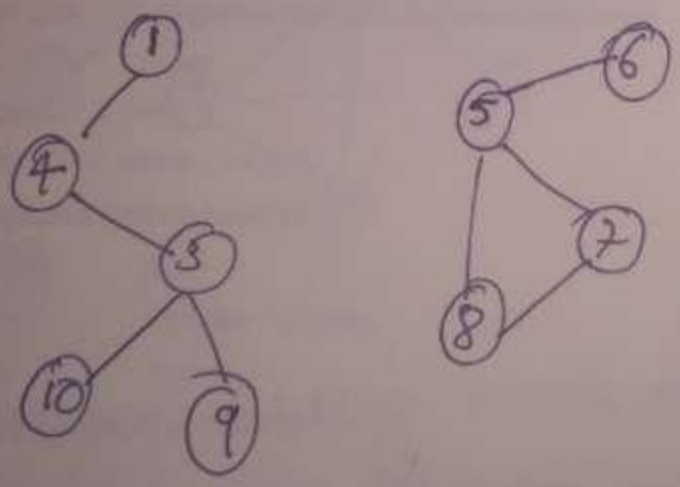
An articulation point in a connected graph G . is a vertex V. If & only if the deletion of vertex V together with all edges incident to V disconnect then the connected graph divid into two (or more non-empty components.

Eg: let us consider the connected graph.

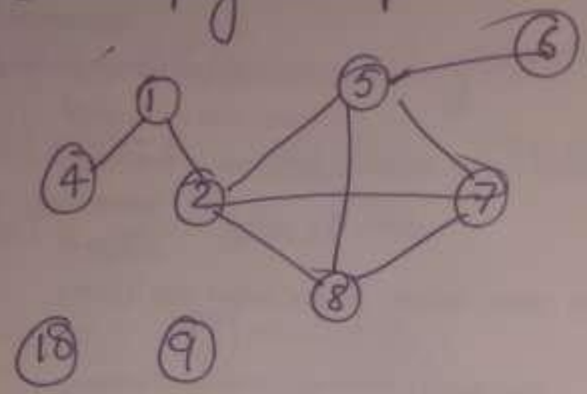


In this graph 2, 3 & 5 are articulation points.

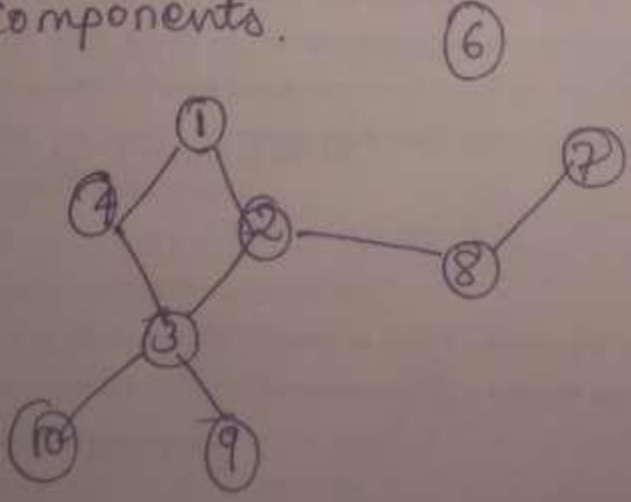
→ If we delete the vertex 2 & the incident edges of vertex 2 then the given graph is divid into two non-empty components



→ After deleting vertex 3 & the incident edges of vertex 3, the given graph is divided into two non-empty components.



→ After deleting vertex 5 & incident edges of vertex 5, the given graph is divided into two non-empty components.



⇒ Articulation point finding:

for finding a articulation points in a graph we have to find first spanning tree.

→ for spanning tree construction we are using DFS.

Steps:

discovery number

1. Find DFS and assign Dfn(Depth first number) to every vertex

2. Find Low( ) for every vertex

3. Check if

$$dfn(U) \leq Low(V)$$

U = parent

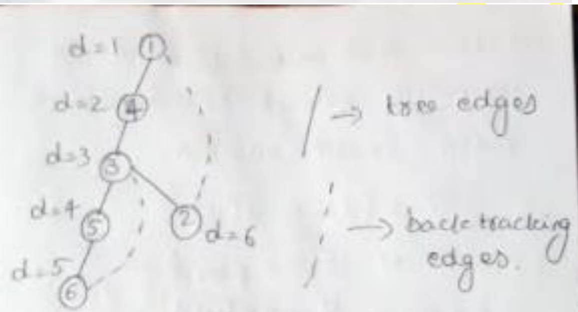
V = child

then 'U' is A.P.

Eg:

Sol: DFS (spanning tree for a given graph)

now write depth first number for each vertex means the vertices in DFS, the order in which they are visited.



→ Now we have to draw the table containing
1. DFS visited vertices
2. DFN numbers of corresponding vertices
3. lowest discovery number
↓
Lowest discovery number is a number reachable from any vertex by taking one back track edge (there is some path that is going back to parent with one back track edge)

DFS	1	2	3	4	5	6
DFN	1	6	3	2	4	5
Lowest number	1	1	1	1	3	3

→ now find articulatⁿ point by comparing parent DFN number & child lowest number.

$\Rightarrow L[V] \geq d[U]$

where U → parent

V → child

→ if above condition satisfied then 'U' is an A.P and this condition is true for all the vertices (excluding) except root vertex. means it's not work for root. so, leave the root & check remaing all vertices.

① U = 4, V = 3

$\Rightarrow L[V] \geq d[U] \Rightarrow L[3] \geq d[4]$

$\Rightarrow 1 \geq 2$ ✗

② U = 3, V = 5

$\Rightarrow L[5] \geq d[3] \Rightarrow 3 \geq 3$ ✓

so, U=3 is an articulatⁿ point.

③ U = 5, V = 6

$\Rightarrow 3 \geq 4$ ✗

④ U = 3, V = 2

$\Rightarrow 1 \geq 3$ ✗

so, the graph has one Articulatⁿ point that is 3 vertex which divid the graph into components known as biconnected component.

Note: If root has more than one child then root is the A.P.

Shortest Paths:

Transitive Closure:

⇒ Transitive closure of a graph:

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph.

Here reachable means that there is a path from vertex 'i' to vertex j

→ The reachability matrix is called the closure of a graph

→ if there is a path from vertex 'i' to vertex 'j' then the entry in reachability matrix is one. other wise zero.

reachable[i][j] = reachable [i] [j]

(or)

reachable[i][k] and reachable [k] [j].

→ A vertex - i is said to be reachable from vertex - j if and only if

* there is a direct path from i to j

(or)

*. there is a path from vertex - i to vertex - k and from vertex - k to vertex j

⇒ The transitive closure of a graph is represented in the form of adjacency matrix.

Warshall's Algorithm:

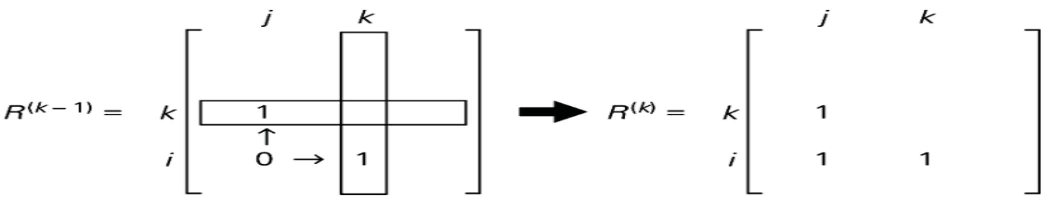
- Warshall's algorithm is used to determine the transitive closure of a directed graph or all paths in a directed graph by using the adjacency matrix. For this, it generates a sequence of n matrices. Where, n is used to describe the number of vertices.

$$R^{(0)}, ..., R^{(k-1)}, R^{(k)}, ... , R^{(n)}$$

- A sequence of vertices is used to define a path in a simple graph. In the k^{th} matrix $(R^{(k)})$, $(r_{ij}^{(k)})$, the element's definition at the i^{th} row and j^{th} column will be one if it contains a path from v_i to v_j. For all intermediate vertices, w_q is among the first k vertices that mean $1 \leq q \leq k$.
- The $R^{(0)}$ matrix is used to describe the path without any intermediate vertices. So we can say that it is an adjacency matrix. The $R^{(n)}$ matrix will contain ones if it contains a path between vertices with intermediate vertices from any of the n vertices of a graph. So, we can say that it is a transitive closure.
  - **Warshall's Algorithm (matrix generation)**
  - Recurrence relating elements R^K to elements of $R^{(K-1)}$ is:

$$R^K [i, j] = R^{(K-1)} [i, j] \text{ or } (R^{(K-1)} [i, k] \text{ and } R^{(K-1)} [k, j])$$

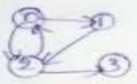
  - It implies the following rules for generating R(k) from R(k-1): **Rule 1** If an element in row i and column j is 1 in R(k-1), it remains 1 in R(k) **Rule 2** If an element in row i and column j is 0 in R(k-1), it has to be changed to 1 in R(k) it has to be changed to 1 in R if and only if (k) if and only if the element in its row i and column k and the element in its column j and row k are both 1's in R(k-1)




```
ALGORITHM   Warshall(A[1..n, 1..n])
    //Implements Warshall's algorithm for computing the transitive closure
    //Input: The adjacency matrix A of a digraph with n vertices
    //Output: The transitive closure of the digraph
    R(0) ← A
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                R(k)[i, j] ← R(k−1)[i, j] or (R(k−1)[i, k] and R(k−1)[k, j])
    return R(n)
```

EXAMPLE:

Transitive closure of a graph by using warshall's algorithm:



$$R^0 = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

$k = 1$ intermediate vertex is 0 so, R 1st row & 1st column are not change in R^1 matrix.

$$R^1 = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

$R^1(1,1) = R^0(1,1) \vee (R^0(1,0) \wedge R^0(0,1))$
$= 0 \vee (0 \wedge 1) = 0$

$R^1(1,2) = 1$

$R^1(1,3) = R^0(1,3) \vee (R^0(1,0) \wedge R^0(0,3))$
$= 0 \vee (0 \wedge 0) = 0$

$R^1(2,1) = R^0(2,1) \vee (R^0(2,0) \wedge R^0(0,1))$
$= 0 \vee (1 \wedge 1) = 1$

$R^1(2,2) = R^0(2,2) \vee (R^0(2,0) \wedge R^0(0,2))$
$= 0 \vee (1 \wedge 1) = 1$

$R^1(2,3) = 1$

$R^1(3,1) = R^0(3,1) \vee (R^0(3,0) \wedge R^0(0,1))$
$= 0 \vee (0 \wedge 1) = 0$

$R^1(3,2) = 0 \vee (0 \wedge 1) = 0$

$R^1(3,3) = 0 \vee (0 \wedge 0) = 0$

$k = 2$ intermediate vertex 1 so, write row & columns as it is in R^2 matrix.

$$R^2 = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 1 & 1 \\ 3 & 0 & 0 & 0 & \end{array}$$

$R^2(0,0) = R^1(0,0) \vee (R^1(0,1) \wedge R^1(1,0))$

$= 0 \vee (1 \wedge 0) = 0$

$R^2(0,3) = R^1(0,3) \vee (R^1(0,1) \wedge R^1(1,3))$

$= 0 \vee (1 \wedge 0) = 0$

$R^2(3,0) = R^1(3,0) \vee (R^1(3,1) \wedge R^1(1,0))$

$= 0 \vee (0 \wedge 0) = 0$

$R^2(3,2) = R^2(3,2) \vee (R^2(3,1) \wedge R^1(1,2))$

$= 0 \vee (0 \wedge 1) = 0$

$R^2(3,3) = R^2(3,3) \vee (R^2(3,1) \wedge R^2(1,3))$

$= 0 \vee (0 \wedge 0) = 0$

$\therefore k = 3$ Intermediate node 2.

so, write Row 3 & column 3 as it is in R^3 matrix

$$R^3 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$R^3(0,0) = R^2(0,0) \vee (R^2(0,2) \wedge R^2(2,0))$

$= 0 \vee (1 \wedge 1) = 1$

$R^3(0,3) = R^2(0,3) \vee (R^2(0,2) \wedge R^2(2,3))$

$= 0 \vee (1 \wedge 1) = 1$

$R^3(1,0) = R^2(1,0) \vee (R^2(1,2) \wedge R^2(2,0))$

$= 0 \vee (1 \wedge 1) = 1$

$R^3(1,1) = R^2(1,1) \vee (R^2(1,2) \wedge R^2(2,1))$

$= 0 \vee (1 \wedge 1) = 1$

$R^3(1,3) = R^2(1,3) \vee (R^2(1,2) \wedge R^2(2,3))$

$= 1$

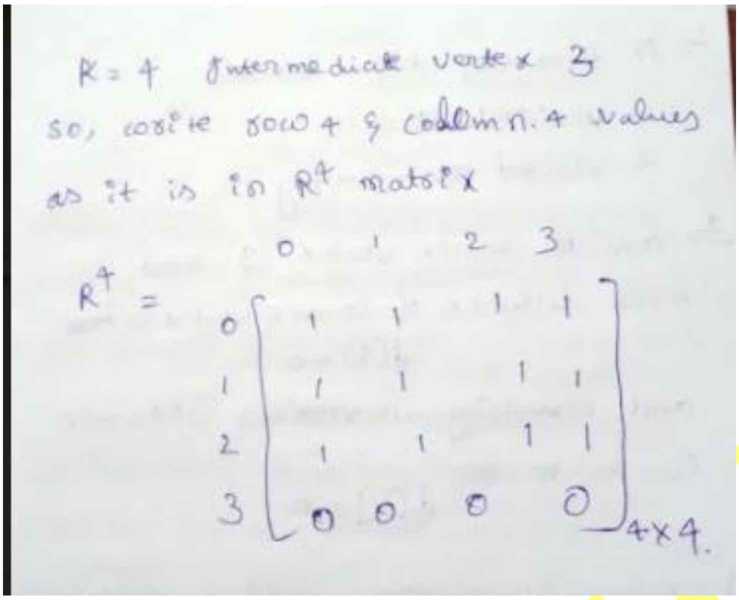
$R^3(3,0) = R^2(3,0) \vee (R^2(3,2) \wedge R^2(2,0))$

$= 0$

$R^3(3,1) = R^2(3,1) \vee (R^2(3,2) \wedge R^2(2,1))$

$= 0 \vee (0 \wedge 1) = 0$

$R^3(3,3) = 0$



K = 4 Intermediate vertex 3
So, write row 4 & column 4 values
as it is in R^4 matrix

$$R^4 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}_{4\times 4} \end{array}$$

All-Pairs Shortest Path:

The problem is to find shortest distances between every pair of vertices in a given edge (negative or positive) weighted directed Graph.

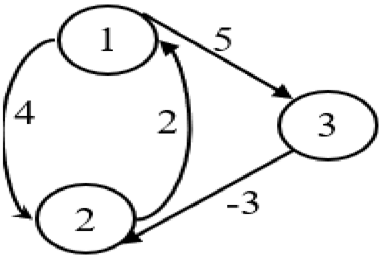
- The all-pair shortest path problem solved by using the algorithm is known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.
  - **Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$.**
  - **When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all, hence $d_{ij}^{(0)} = w_{ij}$.**

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \ , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \ . \end{cases} \tag{1}$$

Floyd-Warshall(W)

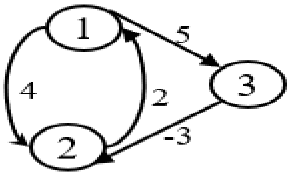
$n \leftarrow rows[W]$
$D^{(0)} \leftarrow W$
for $k \leftarrow 1$ **to** n
    do for $i \leftarrow 1$ **to** n
        do for $j \leftarrow 1$ **to** n
            do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Example1:



$W = D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0



$D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

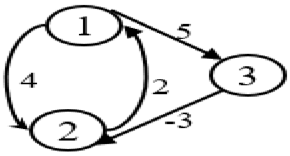
k = 1
Vertex 1 can be intermediate node

$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

$D^1[2,3] = \min(D^0[2,3], D^0[2,1]+D^0[1,3])$
$= \min(\infty, 7)$
$= 7$

$D^1[3,2] = \min(D^0[3,2], D^0[3,1]+D^0[1,2])$
$= \min(-3, \infty)$
$= -3$



$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

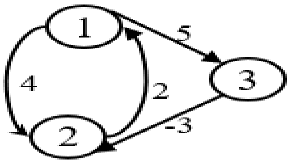
k = 2
Vertices 1, 2 can be intermediate

$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

$D^2[1,3] = \min(D^1[1,3], D^1[1,2]+D^1[2,3])$
$= \min(5, 4+7)$
$= 5$

$D^2[3,1] = \min(D^1[3,1], D^1[3,2]+D^1[2,1])$
$= \min(\infty, -3+2)$
$= -1$



$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

k = 3
Vertices 1, 2, 3 can be intermediate

$D^3 =$

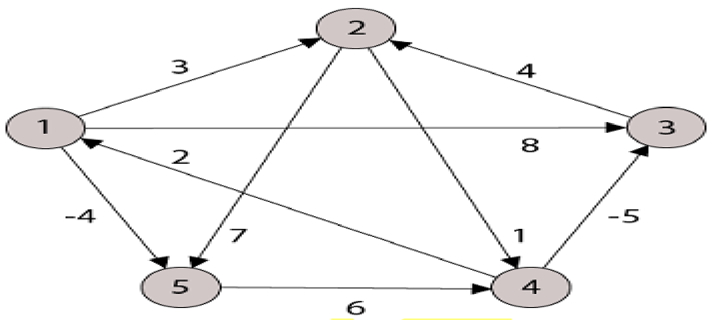
	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

$$D^3[1,2] = \min(D^2[1,2],\ D^2[1,3]+D^2[3,2]) = \min(4,\ 5+(-3)) = 2$$

$$D^3[2,1] = \min(D^2[2,1],\ D^2[2,3]+D^2[3,1]) = \min(2,\ 7+(-1)) = 2$$

Example 2:

Apply Floyd-Warshall algorithm for constructing the shortest path. Show that matrices $D^{(k)}$ computed by the Floyd-Warshall algorithm for the graph.



Solution : solve in your own .

Single Source/All Destination:

- It is a shortest path problem where find shortest path from a given source vertex to all other remaining vertices (remaining vertices are destination vertices).
- Dijkstra’s Algorithm and Bellman Ford Algorithm are the famous algorithms used for solving single-source shortest path problem.

Dijkstra’s Algorithm:

- Dijkstra's algorithm makes use of weights of the edges for finding the path that minimizes the total distance (weight) among the source node and all other nodes. This algorithm is also known as the single-source shortest path algorithm.
- It is important to note that Dijkstra’s algorithm is only applicable when all weights are positive because, during the execution, the weights of the edges are added to find the shortest path

➢ And therefore if any of the weights are introduced to be negative on the edges of the graph, the algorithm would never work properly. However, some algorithms like the ***Bellman-Ford Algorithm*** can be used in such cases.

Algorithm:

1> In first step define two sets

1. unvisited set
2. visited set — empty

2> consider source vertex 'S' and make distance of source vertex is zero

$$d[S] = 0$$

and remaining all vertices distance is set to ∞

$$d[V] = 0$$

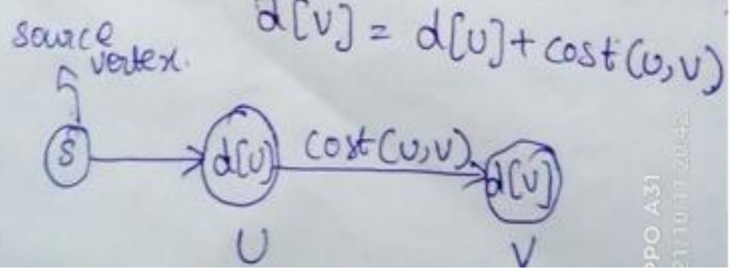
3> Relax all vertices adjacent to the current vertex.

Relaxation property:

consider the edge (a,b)

for the edge from vertex U to the vertex V, if $(d[U] + cost(U,V) < d[V]$

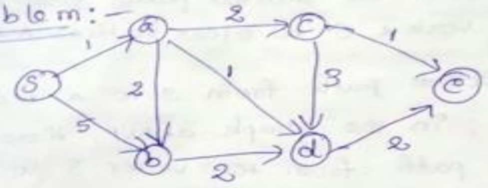
$$d[V] = d[U] + cost(U,V)$$



4. choose the closest vertex as the next current vertex.

→ Repeat steps 3 & 4 until we reach the destination.

⇒ problem:-



step 1

unvisited set = {S, a, b, c, d, e}

visited set = { }

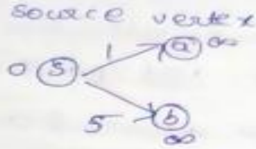
step 2 → select source vertex as 'S' make distance of source vertex zero

$d[S] = 0$

remaining other vertices distance set as ∞

$d[a] = d[b] = d[c] = d[e] = \infty$

step 3: now update visited vertex set with the source vertex why because the source vertex cost is zero and now relax all adjacent edges of source vertex 'S'



Now

$d[v] > d[u] + cost(u, v)$

$d[v] = d[u] + cost[u, v]$

$\Rightarrow d[a] > d[s] + cost(s, a)$

$\infty > 0 + 1 \Rightarrow \infty > 1$ ✓

$\therefore d[a] = d[s] + cost(s, a)$

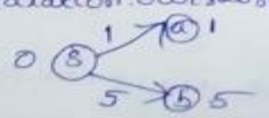
$= 1$

$\Rightarrow d[b] > d[s] + cost(s, b)$

$\infty > 0 + 5 \Rightarrow \infty > 5$

$\therefore d[b] = 5$

after edge Relaxation our shortest path graph is



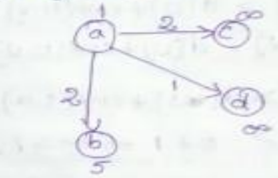
Now, the sets are updated as

unvisited set : {a, b, c, d, e}

visited set : {s}

step-4: now vertex 'a' is chosen this because shortest path estimation for vertex 'a' is least. and update in visited set.

now relaxed all adjacent (outgoing - edges) edges of 'a'.



Now,

$\Rightarrow d[c] > d[a] + cost(a, c)$

$\infty > 1 + 2 \Rightarrow \infty > 3$ ✓

$d[c] = 3$

$\Rightarrow d[d] > d[a] + cost(a, d)$

$\infty > 1 + 1 \Rightarrow \infty > 2$ ✓

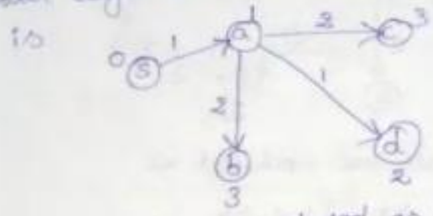
$\Rightarrow d[d] = 2$

$d[b] > d[a] + cost(a, b)$

$5 > 1+2 \Rightarrow 5 > 3$

$d[b] = 3$

after edge relaxation, our shortest path is



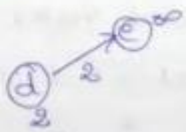
Now, the sets are updated as

unvisited set : $\{b, c, d, e\}$

visited set : $\{s, a\}$

Step 5: vertex d chosen becoz least cost and now add 'd' to the visited set

Now, relax all adjacent edges of 'd'



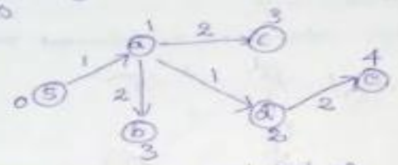
Now,

$\Rightarrow d[e] > d[d] + cost(d,e)$

$\Rightarrow \infty > 2+2 \Rightarrow \infty > 4$

$d[e] = 4.$

after edge relaxation our shortest path is

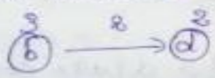


Now, the set are updated as

unvisited set : $\{b, c, e\}$

visited set : $\{s, a, d\}$

Step 6: select 'b' is chosen. becoz 'b' has shortest path (cost) least cost & here vertex 'c' may also select since both vertices having least cost here I selected vertex 'b' so, relax all adjacent vertices of b



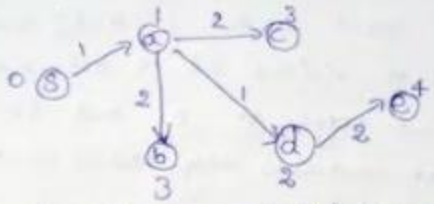
$d[d] \geq d[b] + cost(b,d)$

$2 > 3 + 2$

$2 > 5$ ×

No change in cost of vertex 'd'

after edge relaxatⁿ, our shortest path tree remains the same as in step-5

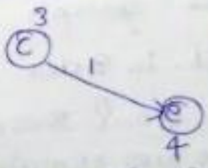


Now, the sets are updated as

unvisited set: {b, c, e}

visited set: {s, a, d}

Step-7: select 'c' vertex has least cost. add in visited set and relax all adjacent ~~vertices of~~ edges 'c'



$$d[e] > d[c] + cost(c,e)$$

$$4 > 3+1 \Rightarrow 4 > 4$$ no change

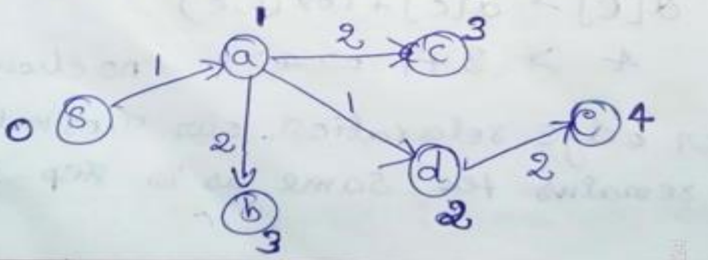
after edge relaxation, our shortest path tree remains the same as in step-5

Now sets are updated as
unvisited set: {e}
visited set: {s, a, d, b, c}

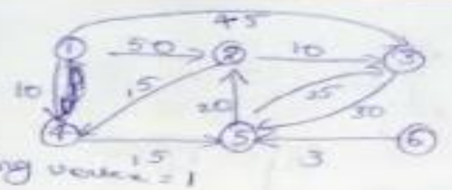
step-8: select vertex 'e' becoz leftout add to visited set & relax all adjacent edges of 'e' but it does not contains any adjacent vertices.
So, our shortest path tree remains same as in step-5, 6, 7

Now, the sets are updated as
unvisited set: { }
visited set = {s, a, d, b, c, e}

from the graph all vertices are proceeded. in order s, a, d, b, c, e.
∴ our final shortest path from s to all other remaining vertices.



Example-2

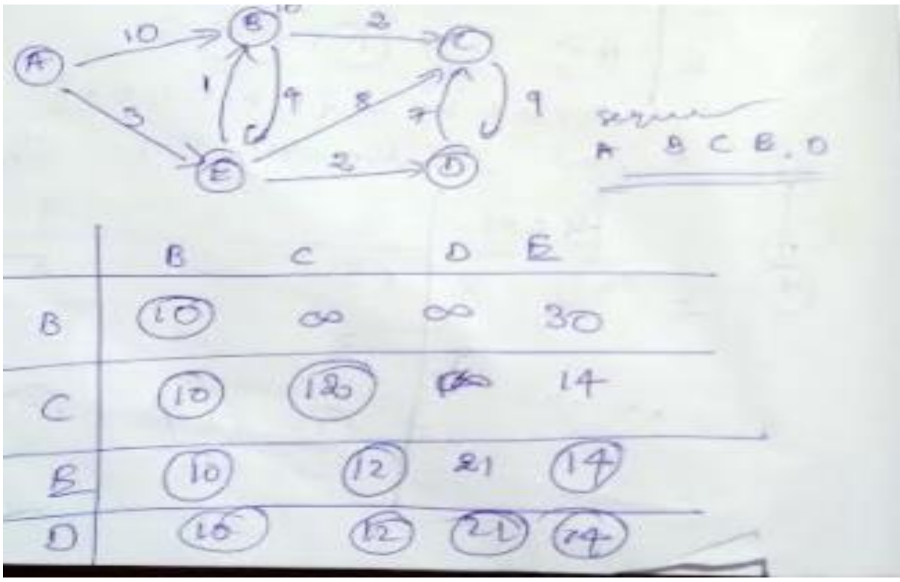


starting vertex = 1

Selected Vertex	2	3	4	5	6
4	50	45	10	∞	∞
5	50	45	10	25	∞
2	45	45	10	25	∞
3	45	45	10	25	∞
6	45	45	10	25	∞

visited set = {1, 4, 5, 2, 3, 6}

Example-3



Example-4