

SYLLABUS

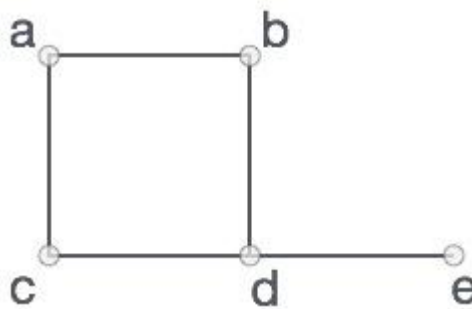
Module 4

Advanced Graph Structures : Representation of graphs, Depth First and Breadth First Traversals, Topological Sorting, Strongly connected Components and Biconnected Components Minimum Cost Spanning Tree algorithms- Prim's Algorithm, Kruskal' Algorithm,. Shortest Path Finding algorithms - Dijkstra's single source shortest paths algorithm

ADVANCED GRAPH STRUCTURES

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

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

$$E = \{ab, ac, bd, cd, de\}$$

GRAPH DATA STRUCTURE

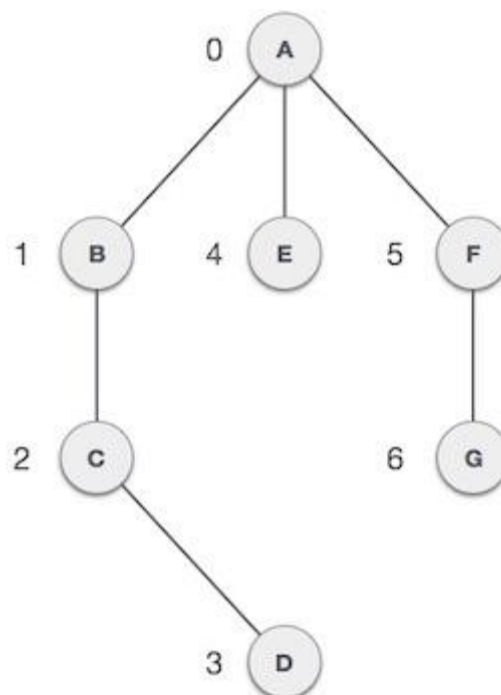
Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

Vertex – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

Edge – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

Adjacency – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

Path – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



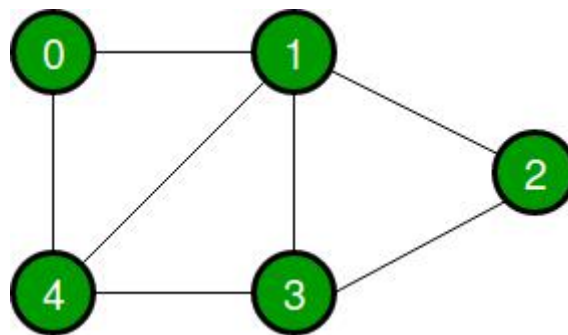
REPRESENTATION OF GRAPHS

The following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

ADJACENCY MATRIX:

Adjacency Matrix is a 2D array of size $V \times V$ 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 . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .



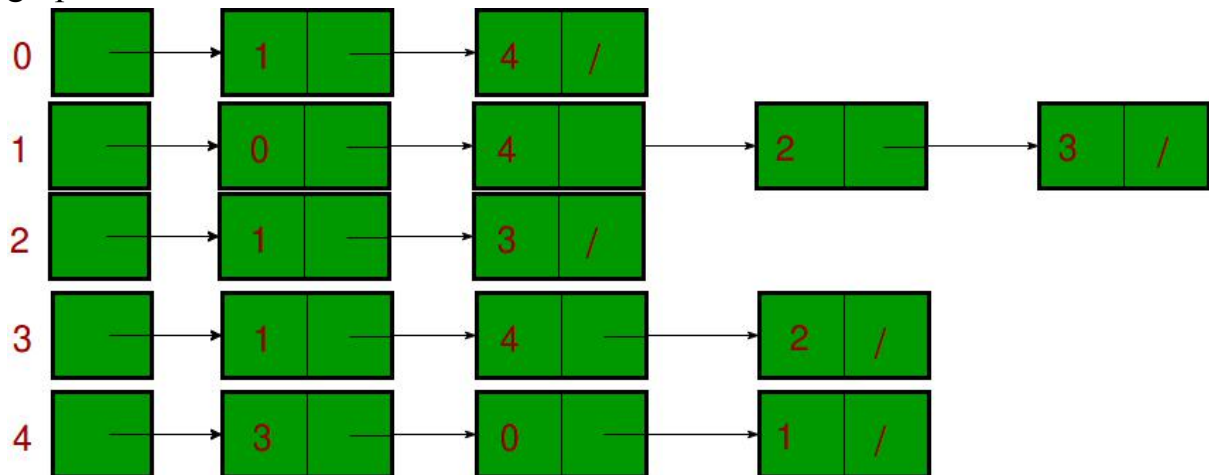
The adjacency matrix for the above example graph is:

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

ADJACENCY

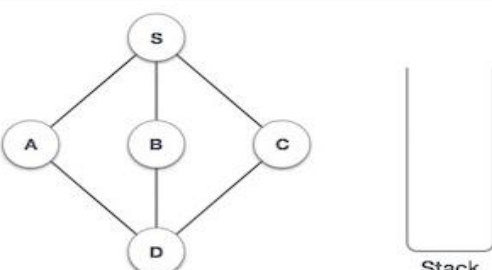
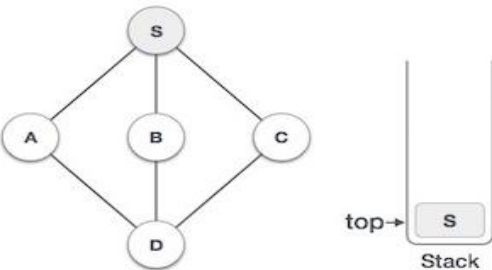
LIST:

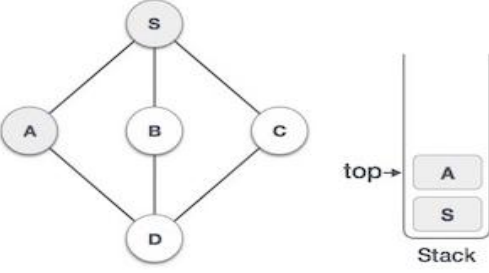
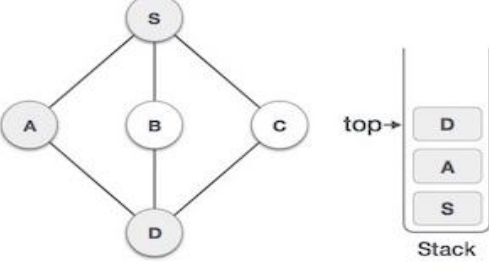
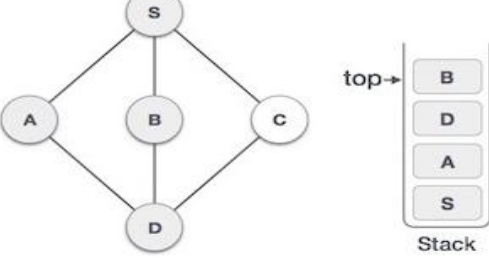
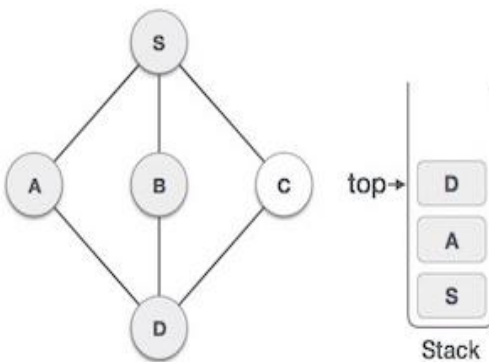
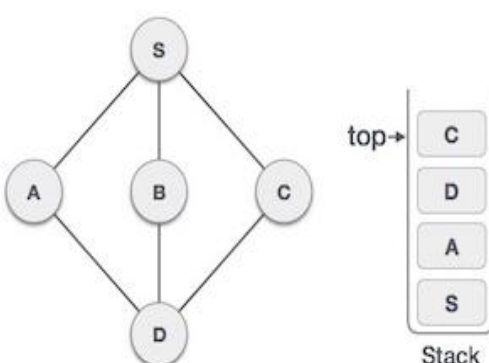
An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above grap



DEPTH FIRST TRAVERSAL

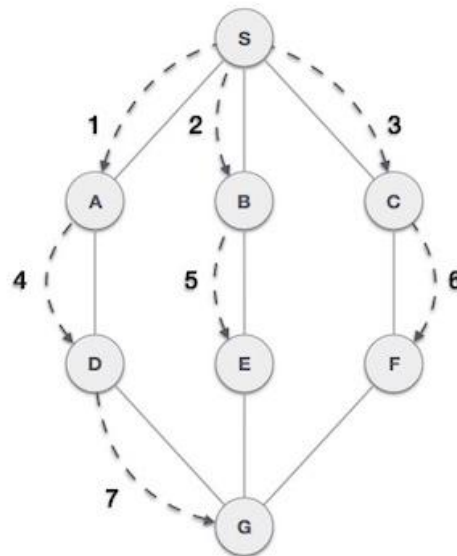
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3		<p>Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4		<p>Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.</p>
5		<p>We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.</p>
6		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>
7		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>

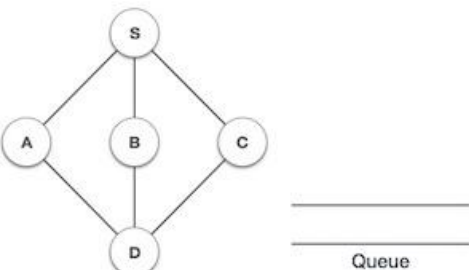
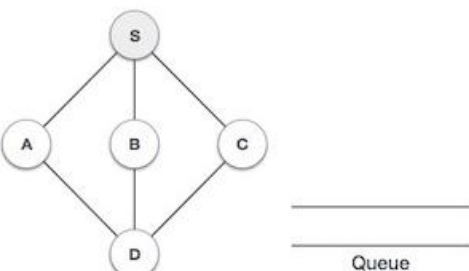
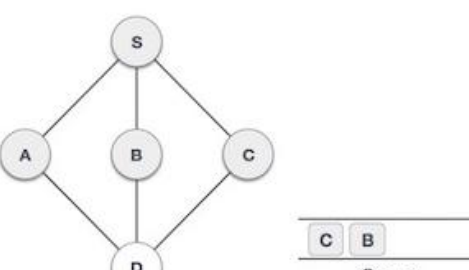
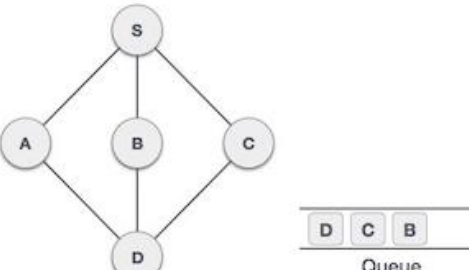
BREADTH FIRST TRAVERSAL

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



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

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

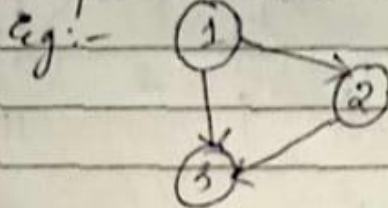
Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.
6		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A .
7		From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

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

Topological sorting

To find topological sorting of a graph we need to consider 3 cases.

i) Graph should be directed and acyclic (DAG).

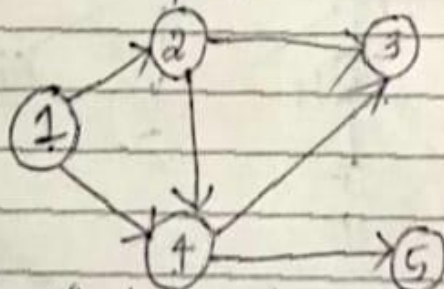


ii) It is a linear ordering of vertices such that for every directed edge u, v for ~~vertex~~ edge $u \rightarrow v$ u comes before v .



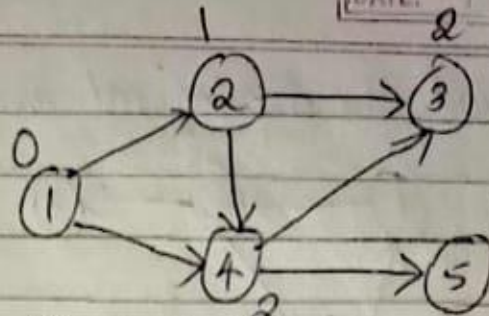
iii) Every directed acyclic graph will have at least one topological sorting.

Soln: Find topological ordering of the given graph.

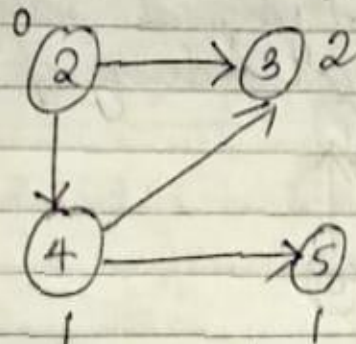


- i) It's a directed and acyclic graph.
- ii) Find the in-degree of all vertices
 $(1) \rightarrow 0$

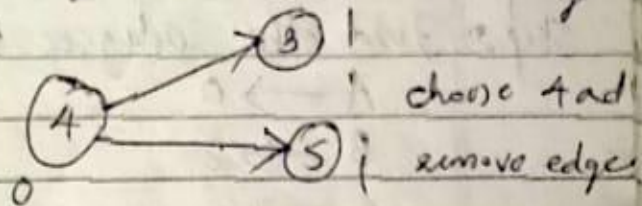
- ② → 1
- ③ → 2
- ④ → 2
- ⑤ → 1



- Choose vertex with 0 indegree and remove the edges starting from that vertex.
- ⇒ Update the indegree



- ② → 0
 - ③ → 2
 - ④ → 1
 - ⑤ → 1
- choose 2 and remove the edges

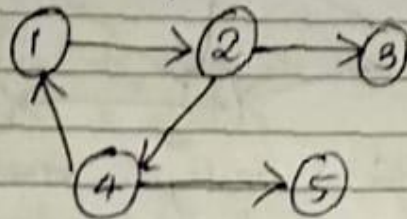


- Update the indegree

- ③ → 0
 - ⑤ → 0
- choose one vertex and sort over it. Then, we get the topological sorting.

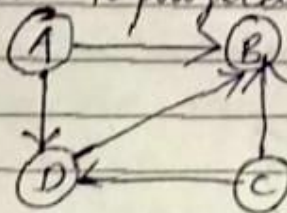
Topological sorting: 1, 2, 4, 3, 5 or 1, 2, 4, 5, 3.

Q: Find the topological ordering of gives graph.



Since this is a cyclic directed graph (1-2; 4-1) we can't find the topological sorting.

3. Find the topological ordering of



Step 1: It's a directed and acyclic graph.

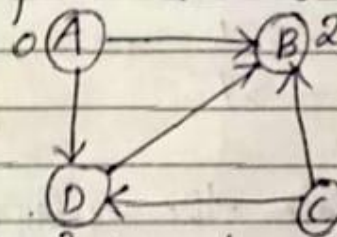
Step 2: Find the indegree of all vertices.

A \rightarrow 0

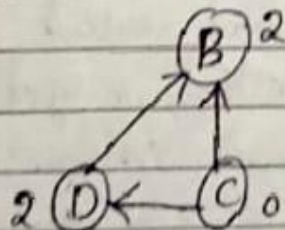
B \rightarrow 2

C \rightarrow 0

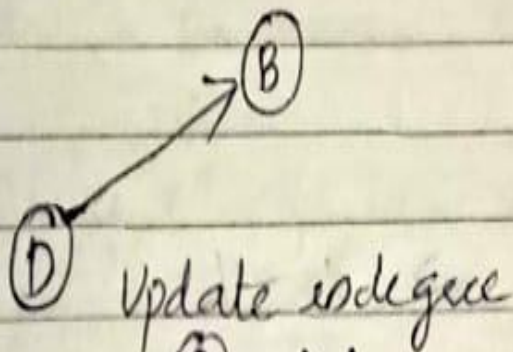
D \rightarrow 2



Case 1: select vertex² A and remove edges.



Case 2: select vertex c and remove edges.



Update indegree

$B \rightarrow 1$

$D \rightarrow 0$

select vertex D and remove edge.

B^0

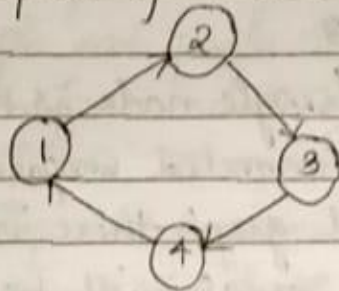
Topological sorting: - A, C, D, B.

Module - 4

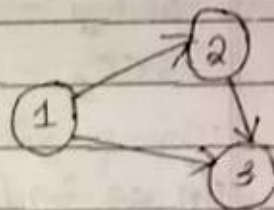
Strongly Connected components

Strongly connected graph:- A directed graph is strongly connected if there is a path between all pairs of vertices.

Eg:-



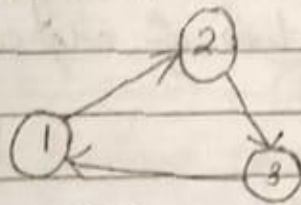
Strongly connected



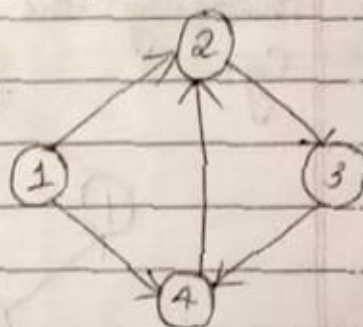
Not strongly connected

A strongly connected components of a directed graph is a maximal strongly connected subgraph.

Eg:-



It is a SCC
 $SCC = 1, 2, 3$

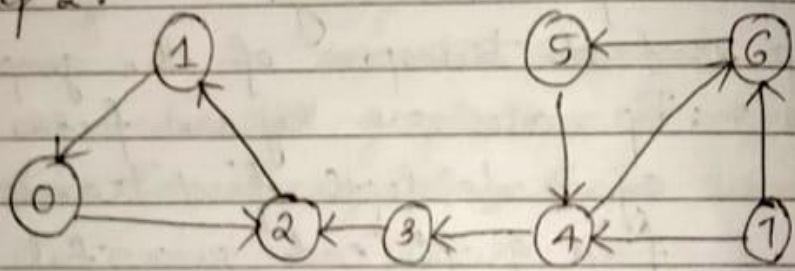


It's not a SCC

$SCC_1 = 2, 3, 4$ $SCC_2 = 1$

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7

step 2:



step 3:

- ~~0~~
- ~~1~~
- ~~2~~
- ~~3~~
- ~~4~~
- ~~5~~
- ~~6~~
- 7

0, 2, 1 - SCC₁
 3 - SCC₂
 4, 6, 5 - SCC₃
 7 - SCC₄

To find the strongly connected components of a graph, we can use different algorithms. Here we use Kosaraju's algo.

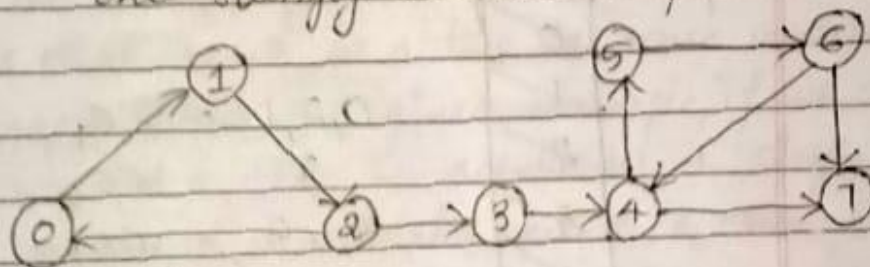
1. Kosaraju's algo

Step 1: Perform Depth First Traversal of a graph and push the nodes again to stack before returning.

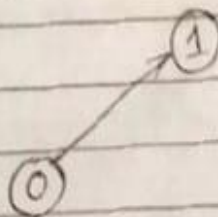
Step 2: Find transpose of the graph.

Step 3: Pop nodes one by one from stack and again do depth first traversal on modified graph (each successful DFS gives one strongly connected component.)

Eg:-

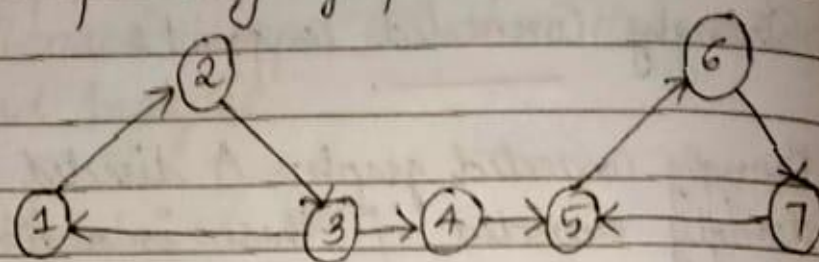


Step 1: Perform DFS.



0, 1, 2, 3, 4, 5, 6, 7

Qn:- Find the strongly connected components of the following graph.



$SCC_1 - 5, 6, 7$

$SCC_2 - 1, 2, 3$

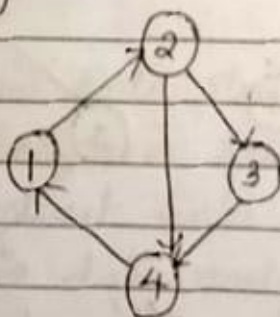
$SCC_3 - 4$ // single node is always a strongly connected component

In an undirected graph, there is an edge b/w vertices then the graph will be strongly connected.

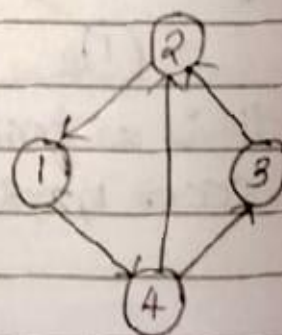
Transpose of a graph

Transpose of a graph is simply obtained by changing the direction of edges.

Eg:-



G_1

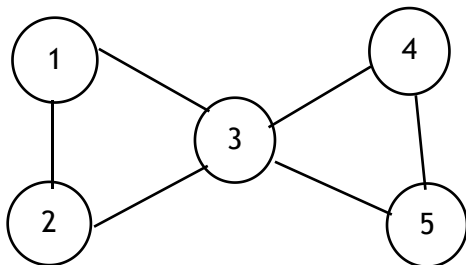


G_{1-T}

BICONNECTED COMPONENTS

A **biconnected component** of a graph is a connected subgraph that cannot be broken into disconnected pieces by deleting any single node. That is, a Graph G is biconnected if and only if it contains no articulation point.

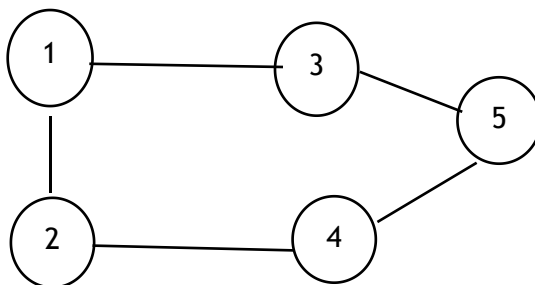
An **articulation point** is a node of a graph whose removal would cause an increase in the number of connected **components**. In other words, A vertex in a graph G (connected graph) is an articulation point if and only if we delete the vertex v and all its edges then it disconnects the graph into 2 or more non empty components.



We can delete any vertex and its associated edges that result in two or more connected subgraphs, the given example:

- If we delete the vertex 1 then it results in a single connected graph.
- If we delete vertex 3 then it results in 2 subgraphs then, vertex 3 is an articulation point, so this graph is not biconnected.

Eg: for biconnected graph



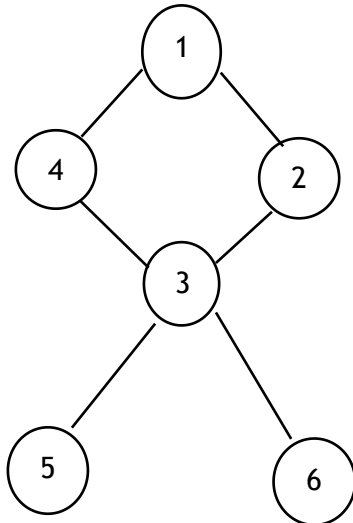
Q: Find the articulation point in a graph

Step 1: Construct depth first traversal and provide number for each node according to the

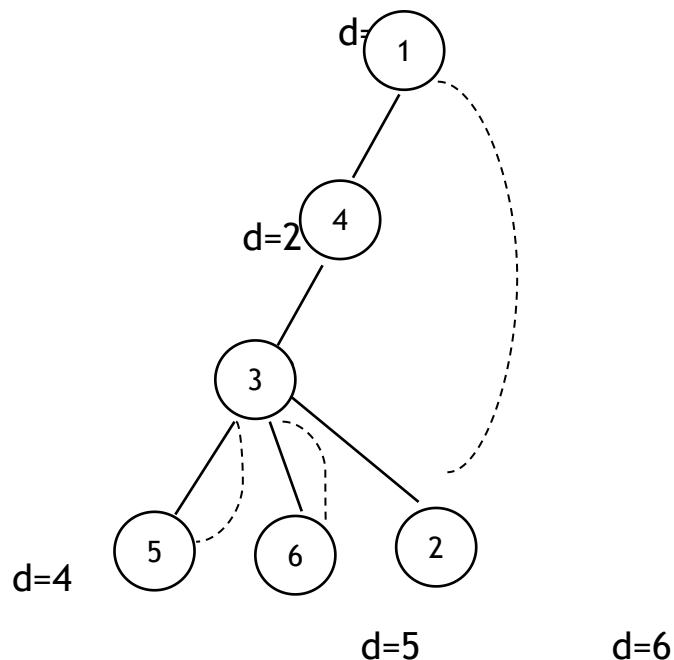
Traversal. Find the lowercase number of parent for each node.

Step 2: If a root node has atleast 2 children then it will be articulation point. Also, leaf node has no Articulation point.

Eg:



Dfs for the above graph



vertices	1	2	3	4	5	6
Discovery time(d)	1	6	3	2	4	5
Lowest discovery number(L)	1	1	1	1	3	3

Here leaf node has no articulation point so , leaf node is biconnected.

To find the articulation point consider 2 edges u,v.here,u is the parent and v is the child

then u,v become an articulation point if and only if

$$L[v] \geq d[u]$$

If this satisfies then u is an articulation point.

Consider the next vertex v=4 and u=3

$$L[4] \geq d[3]$$

$$1 \geq 3$$

Here the condition is not satisfied.

Consider the next 2 vertex v=5 and u=3

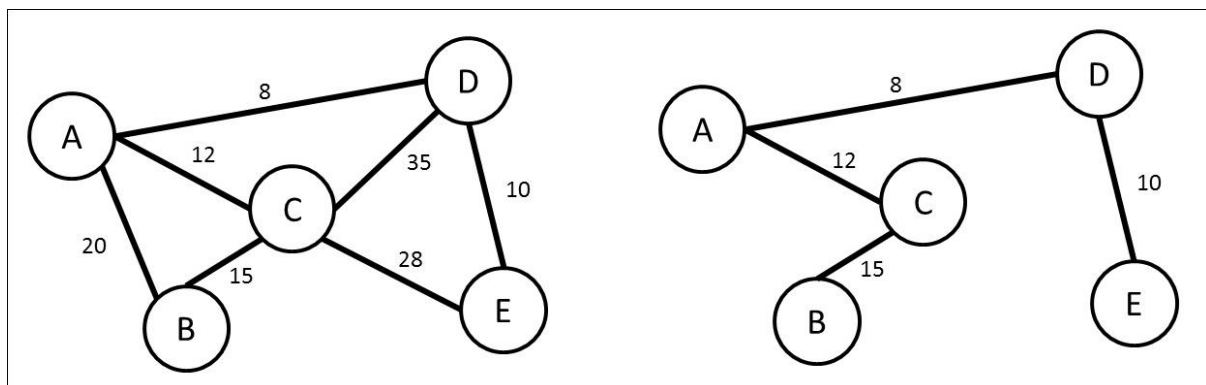
$$L[5] \geq d[3]$$

$$3 \geq 3$$

Here the condition is satisfied .therefore, u is an articulation point.so,3 is an articulation point.

Minimum cost spanning tree

A **Minimum Spanning Tree (MST)** works on graphs with directed and weighted (non-negative costs) edges. Consider a graph G with n vertices. The spanning tree is a subgraph of graph G with all its n vertices connected to each other using $n-1$ edges. Thus, there is no possibility of a cycle with the subgraph. If the spanning tree does have a cycle, then it is advisable to remove any one edge, probably the one with the highest cost. The spanning tree with the least sum of edge weights is termed as a MST. It is widely used in applications such as laying of power cables across the city, connecting all houses using the least length of power cables. Here, the weight of each edge is the length of the cable, and the vertices are houses in the city. The most common algorithms to find the minimum cost spanning tree are Prim's algorithm and Kruskal's algorithm. *Figure 8.11* shows the minimum cost spanning tree for an undirected-weighted graph.



1 .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 explore 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.

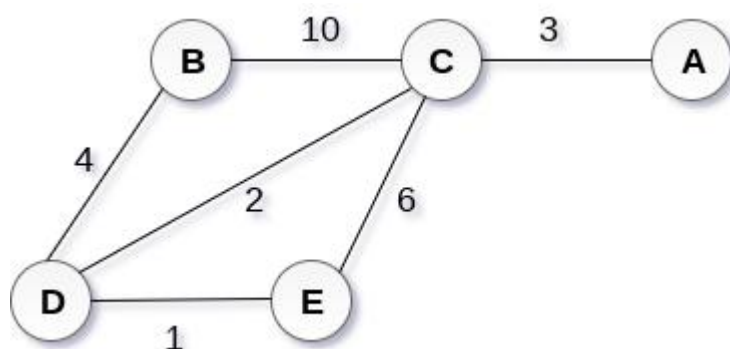
The algorithm is given as follows.

Algorithm

- **Step 1:** Select a starting vertex
- **Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
- **Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
- **Step 4:** Add the selected edge and the vertex to the minimum spanning tree T
[END OF LOOP]
- **Step 5:** EXIT

Example :

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



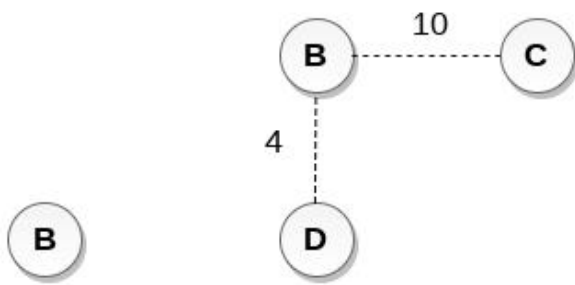
Solution

- **Step 1 :** Choose a starting vertex B.
- **Step 2:** Add the vertices that are adjacent to A. the edges that connecting the vertices are shown by dotted lines.
- **Step 3:** Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.
- **Step 3:** Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e. E and A.
- **Step 4:** Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

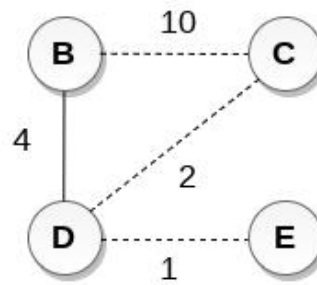
The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.

The cost of MST will be calculated as;

$$\text{cost(MST)} = 4 + 2 + 1 + 3 = 10 \text{ units.}$$

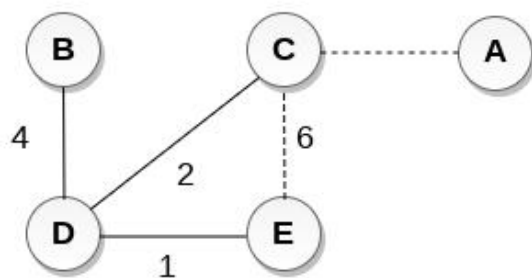


Step 1

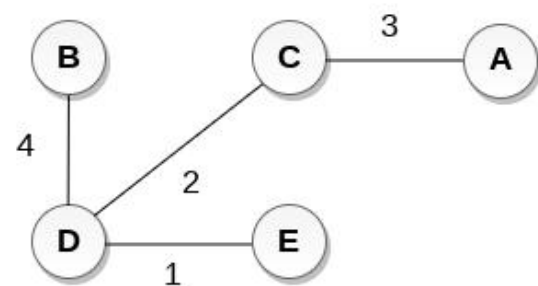


Step 2

Step 3



Step 4



Step 5

2. Kruskal's Algorithm

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.

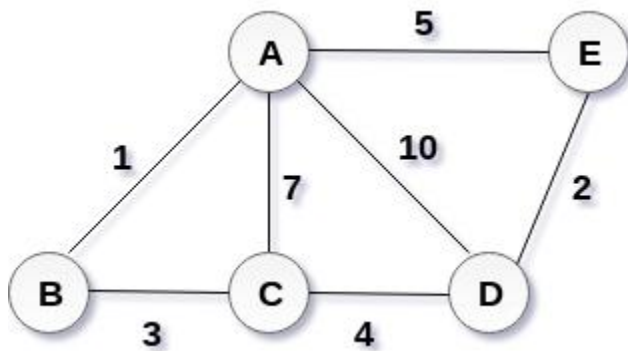
Step 1: Remove all loops and parallel edges.

Step 2: Arrange all edges in the increasing order of their weight.

Step 3: Add the edges which have least weight.

Example :

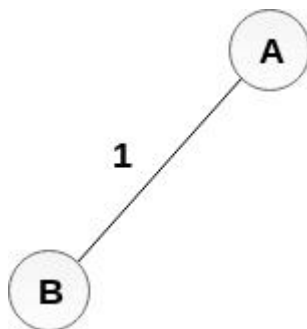
Apply the Kruskal's algorithm on the graph given as follows.



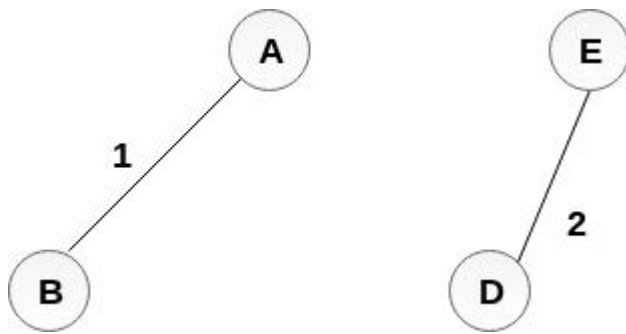
Solution:

Start constructing the tree;

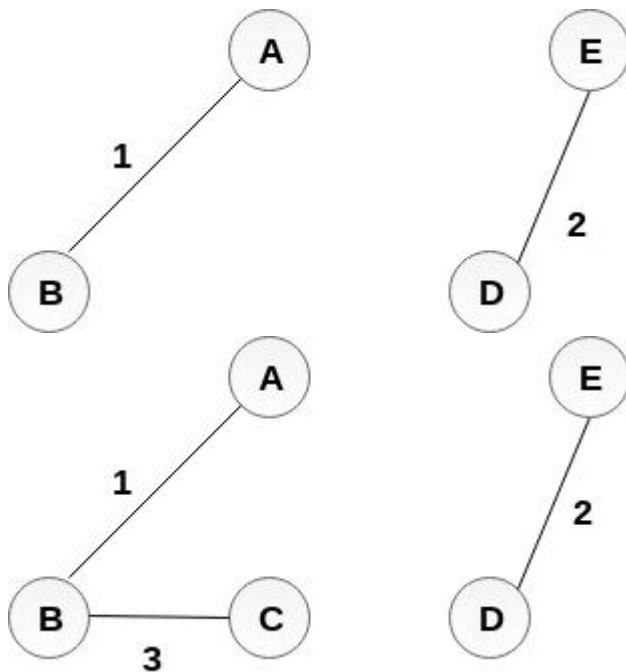
Add AB to the MST;



Add DE to the MST;



Add BC 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 = $1 + 2 + 3 + 4 = 10$.

SHORTEST PATH FINDING ALGORITHM

It is the shortest distance path from one source to another vertices

- All pair shortest path
- Single source

To find the shortest path in a graph. There are 2 types of algorithm to find shortest path.

1. All pair shortest path

To find the shortest path from each vertex to every other vertex using the algorithm Floyd warshall's Algorithm.

2. Single source shortest path

To find the shortest path from a single source vertex 'u' to a destination vertex 'v' using the algorithm Dijkstra's Algorithm.

Dijkstra's Algorithm

Step 1: Assign every node in tentative distance infinite.

Step 2: set initial node as current and other nodes as unvisited.

Step 3: For current node consider all other unvisited nodes and calculate tentative distance. compare current distance with calculated distance and assign the smaller value.

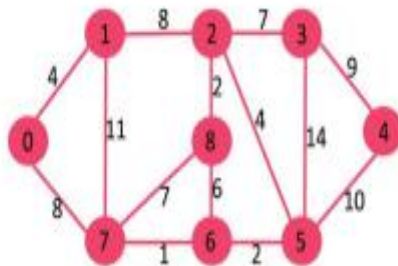
$$d(u) + c(u, v) < d(v)$$

$$d(v) = d(u) + c(u, v)$$

Step 4: When all the neighbors are considered of the current node make it visited.

Step 5: When the destination node is visited then we can stop the process.

Let us understand with the following example:



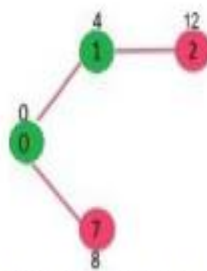
Zero is the source vertex.

Assign a tentative distance.

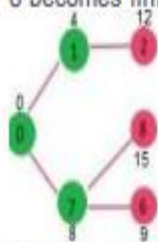
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



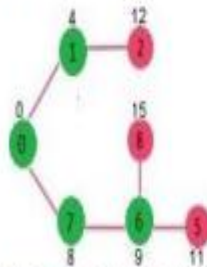
Pick the vertex with minimum distance value and not already included in SPT (not in *sptSet*). The vertex 1 is picked and added to *sptSet*. So *sptSet* now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSet). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSet). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until sptSet does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

