UNIT-IV

NON LINEAR DATA STRUCTURES- GRAPHS

Definition - Representation of Graph - Types of graph - Breadth first traversal - Depth first traversal - Topological sort - Bi-connectivity - Cut Vertex - Euler Circuits - Applications of graphs.

*——— * ——— * ——— * ——— * ——— *

Definitions:-

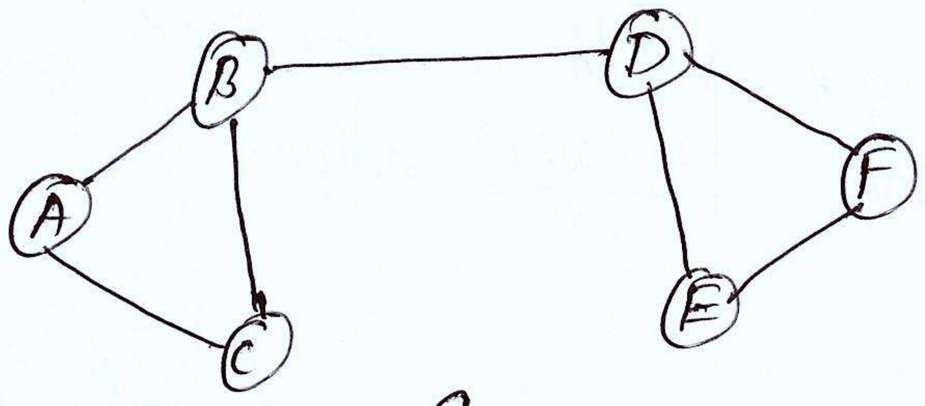
A graph $G = (V, E)$ consist of a set of vertices, V, and a set of edges, E. Each edge in a pair (v, w), where $v, w \in V$. The set V is a finite, nonempty set of vertices. The set E is a set of pairs of vertices representing edges.

$$G = (V, E)$$

$V(G)$ = Vertices of graph G

$E(G)$ = Edges of graph G

Vertices are referred to as nodes and edges are sometimes referred to as arcs.



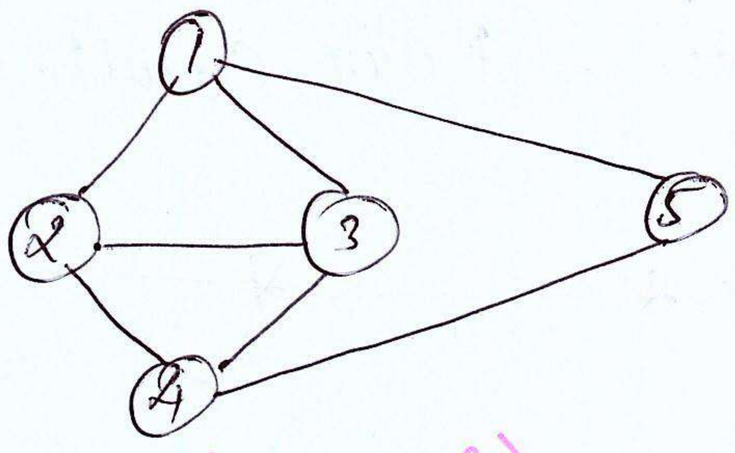
G_1

$$V(G_1) = \{A, B, C, D, E, F\}$$

$$E(G_1) = \{(A,B), (A,C), (B,C), (B,D), (D,E), (D,F), (E,F)\}$$

Undirected Graph;-

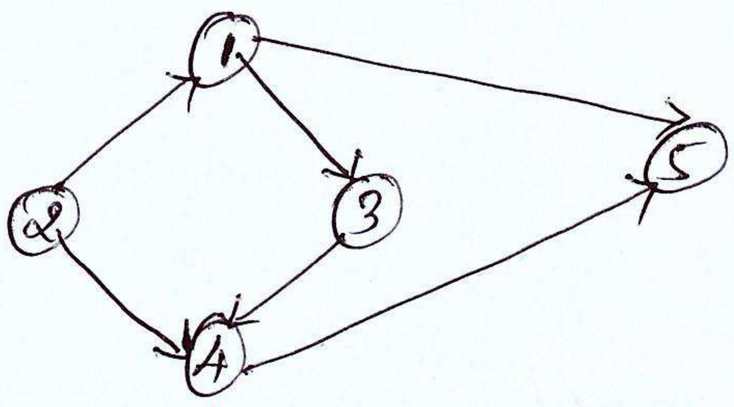
A graph containing unordered pair of vertices is called an undirected graph. In an undirected or unordered graph, (A,B), and (B,A) represent the same edge.



The set of vertices V = {1, 2, 3, 4, 5}.
The set of edges E = {(1,2), (1,3), (1,5), (2,3), (2,4), (3,4), (4,5)}

Directed Graph;-(Digraph)

A graph containing ordered pair of vertices is called directed graph. If an edge is represented using a pair of vertices (V_1, V_2) then the edge is said to be directed from V_1 to V_2.

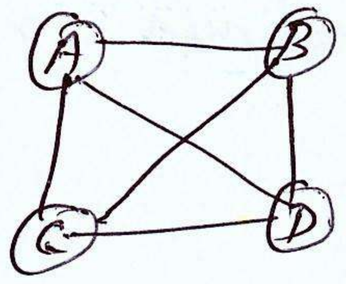


The set of vertices V = {1, 2, 3, 4, 5}
The set of edges E = {(1,3), (1,5), (2,1), (2,4), (3,4), (4,5)}.

Complete Graph;-

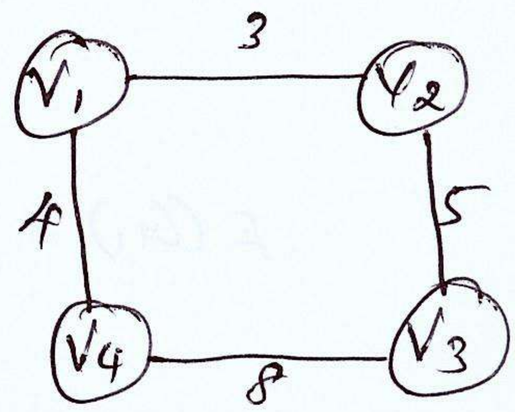
An undirected graph, in which every vertex has an edge to all other vertices is called as complete graph.

A complete graph with N vertices has $\frac{N(N-1)}{2}$ edges.



Weighted Graph;-

A weighted graph is a graph in which edges are assigned some value. Weight of an edge is also called its cost. It can be either directed or undirected.



Adjacent Nodes;-

Two vertices V_1 and V_2 are said to be adjacent if there is an edge between V_1 and V_2.

Path;-

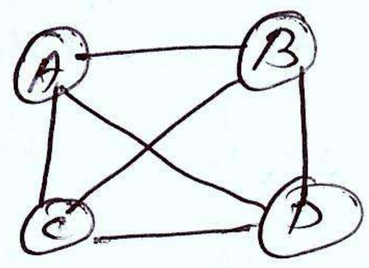
A path from vertex V_0 and V_n is a sequence of vertices $V_0, V_1, V_2, \ldots, V_{n-1}, V_n$. Here, V_0 is adjacent to V_1, V_1 is adjacent to V_2, and V_{n-1} is adjacent to V_n.

Length;-

The length of a path is the no. of edges on the path. A path with n vertices has a length of n-1.

Cycle;-

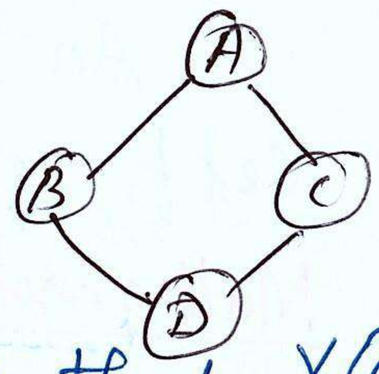
A cycle is a simple path that begins and ends at the same vertex.



ABDA is a cycle of length 3.
BDCB " " " " 3.
ABCDA " " " 4

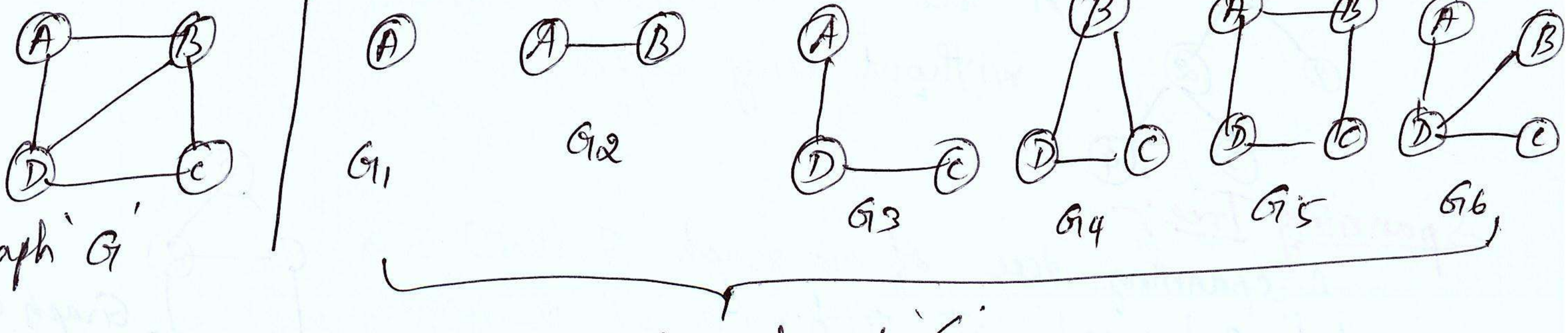
Connected Graph;-

A graph is said to be connected if there exists a path between every pair of vertices V_i and V_j.



Subgraph;-

A subgraph of G is a graph of G, such that $V(G_1)$ is a subset of $V(G)$ and $E(G_1)$ is a subset of $E(G)$



Degree of a vertex;-

The total no. of edges linked to a vertex is called its degree.

* The indegree of a vertex is the total no. of edges coming to that node.

* The outdegree of a vertex is the total no. of edges going out from that node.

* A vertex which has only outgoing edge and no incoming edges is called a source.

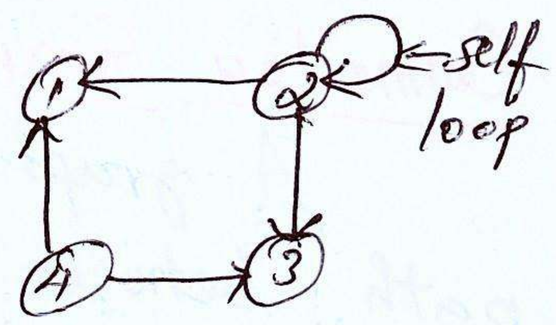
* A vertex which has only incoming edges and no outgoing edges is called a sink.

* When indegree of a vertex is one and outdegree is zero then such a vertex is called a pendant vertex.

* When the degree of a vertex is zero, it is an isolated vertex.

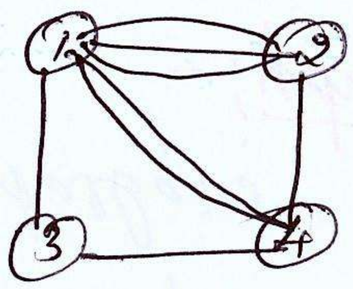
Self Edge/Loops;-

An edge of the form (V,V) is known as self edge or self loop.

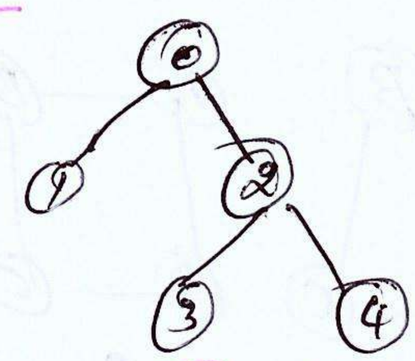


Multigraph;-

A graph with multiple occurrences of the same edge is known as a multigraph.



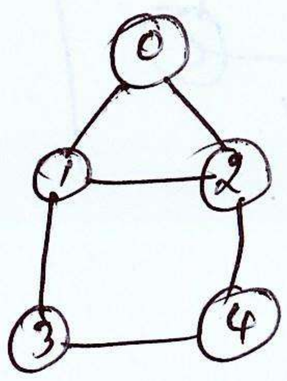
Tree;-



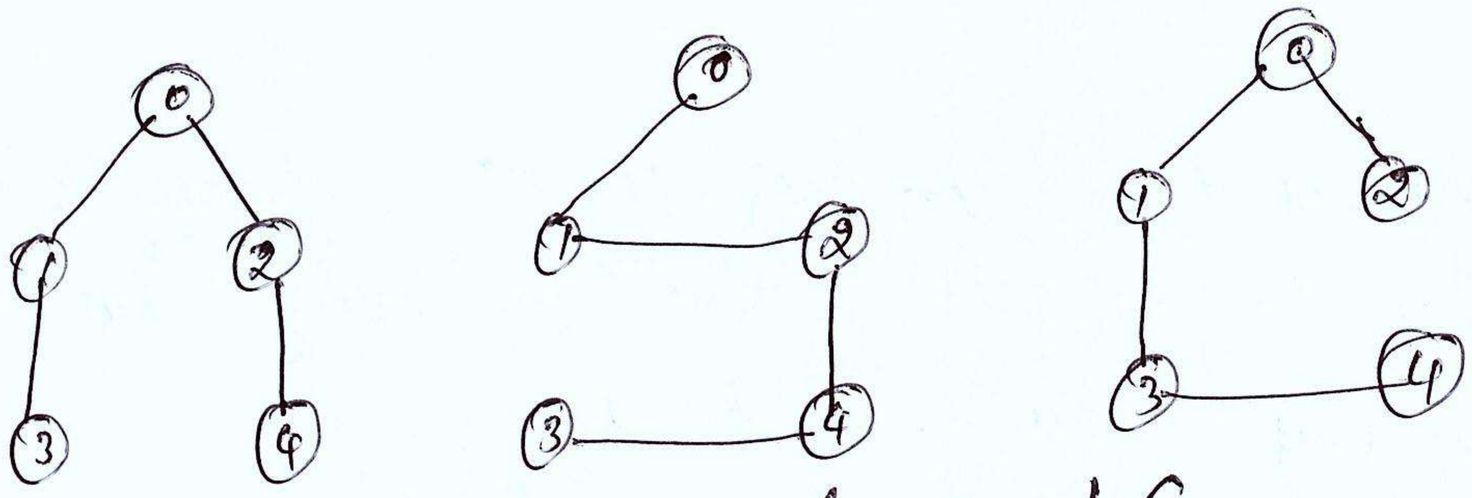
A tree is a connected graph without any cycle.

Spanning Tree;-

A spanning tree of a graph G=(V,E) is a connected subgraph of G having all the vertices of G and no cycles in it. A graph may have multiple spanning tree.



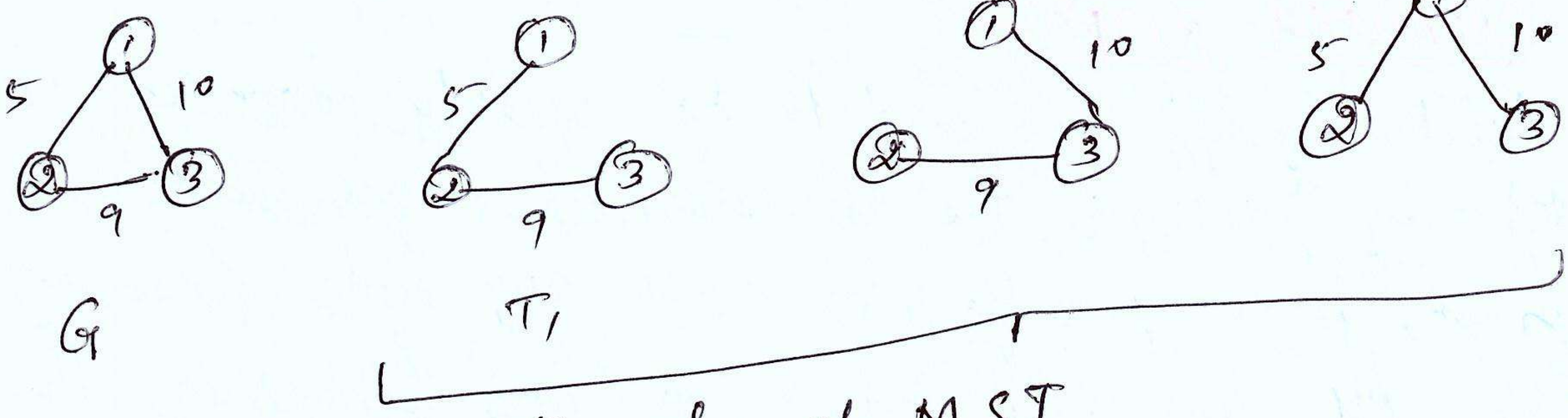
Graph G



Spanning trees of graph G.

Minimum Spanning Tree :- (MST)

The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a group $G=(V,E)$ is called a minimal cost spanning tree or simply the minimum spanning tree of G, if its cost is minimum.



G T_1

Examples of MST

G ⇒ A sample weighted graph

T_1 ⇒ A spanning tree of G with cost 5+9=14

T_2 ⇒ " " " " " " 10+9=19

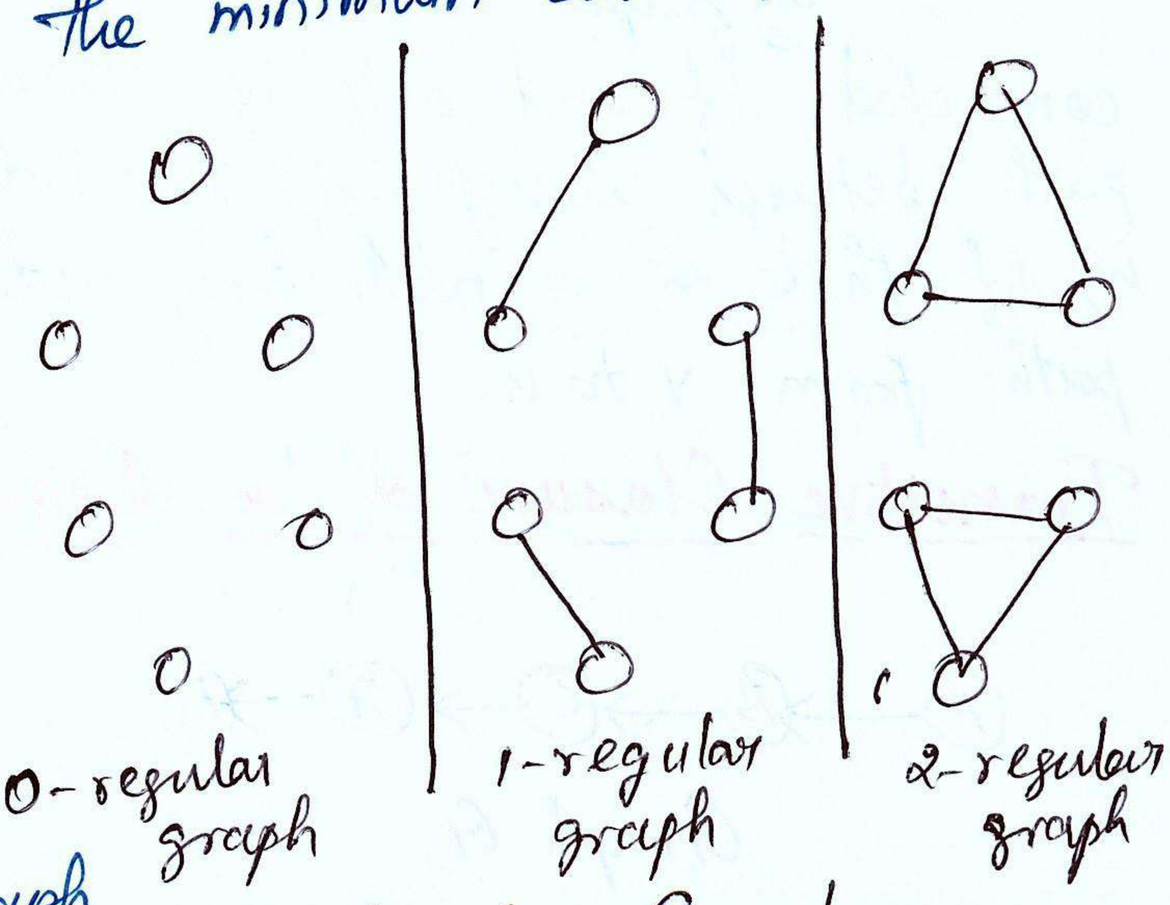
T_3 ⇒ " " " " " " 5+10=15

∴ T_1 with cost of 14 is the minimum cost spanning tree of the graph G.

Regular Graph :-

It is a graph where each vertex has the same no. of neighbours. i.e, every node has the same degree.

A regular graph with vertices of degree k is called k-regular graph of degree k.



0-regular graph 1-regular graph 2-regular graph

Fig. Regular Graphs

Clique:-

In an undirected graph $G=(V,E)$, clique is a subset of the vertex set $C \subseteq V$, such that for every two vertices in C, there is an edge that connects two vertices.

Size of a graph:-

The size of a graph is the total no. of edges in it.

Cut vertex:-

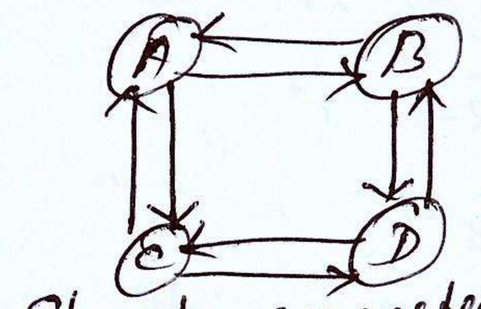
In an directed graph, a vertex which when deleted would disconnect the remaining graph.

Weakly connected graph:-

A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction. The nodes in a weakly connected graph must have either outdegree or indegree of at least 1.

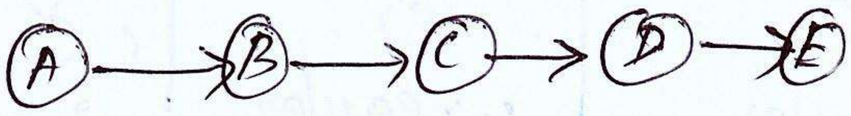
Strongly connected digraph:-

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

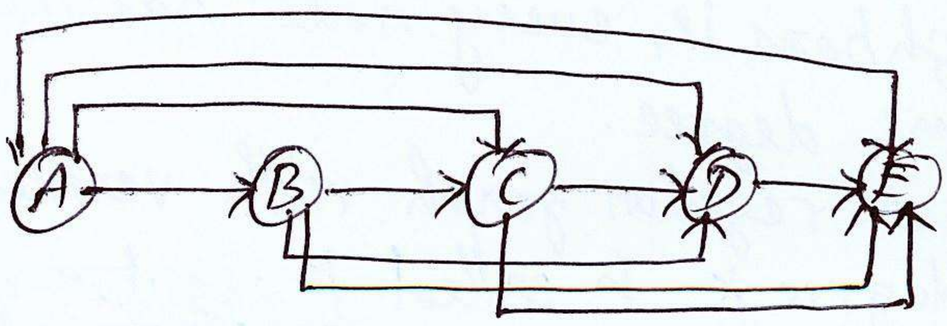


Strongly connected directed acyclic graph.

Transitive Closure of a Digraph:-



Graph G



Transitive Closure of G

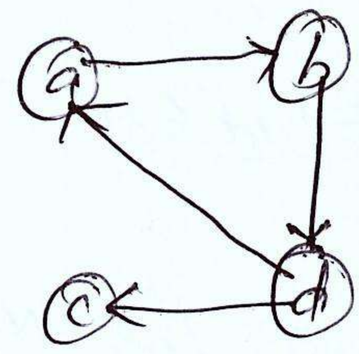
For a directed graph $G=(V,E)$, where V is the set of vertices and E is the set of edges, the transitive closure of G is a graph $G^* = (V, E^*)$. In G^*, for every vertex pair v,w in V there is an edge (v,w) in E^* if and only if there is a valid path from v to w. in G.

Where and why it is needed?

* It is used to find the reachability analysis of transition networks representing distributed and parallel system.
* It is used in the construction of parsing automata in Compiler construction.
* It is used to evaluate recursive database queries.

The transitive closure of a digraph with n vertices is the nxn boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the jth column is 1, if there exists a nontrivial path (i.e, directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.

We can generate the transitive closure of a digraph with the help of depth-first search (DFS).



Digraph G.

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Transitive Closure of G

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Adjacency matrix of G

Representation of Graph;-

Let $G=(V,E)$ be a graph with $n=|V|$, $m=|E|$ and $V=\{v_1, v_2, v_3, \ldots, v_n\}$. There are two types of data structures that are useful for representing graphs.

1). Adjacency matrix representation
2). Adjacency list representation.

1) Adjacency Matrix Representation;-

An adjacency matrix is used to which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.

Let G can be represented by an $n \times n$ matrix $A=(a_{ij})$, called the adjacency matrix of G.

A is defined by

$$a_{ij} = \begin{cases} 1 & \text{if } v_i v_j \in E \quad \text{for } 1 \le i,j \le n \\ 0 & \text{otherwise.} \end{cases}$$

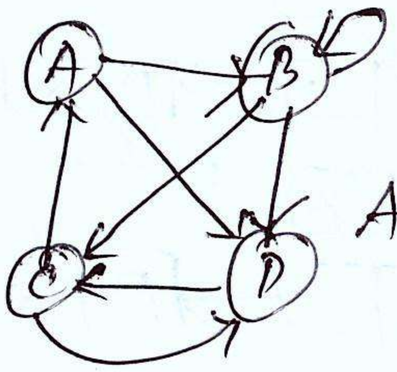
The adjacency matrix of an undirected graph is symmetric

If $G=(V,E,W)$ is a weighted graph, the weight can be stored in the adjacency matrix by modifying its definition as follows:

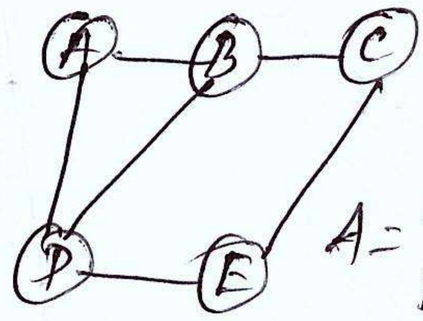
$$a_{ij} = \begin{cases} W(v_i, v_j) & \text{if } v_i v_j \in E \quad \text{for } 1 \le i,j \le n. \\ c & \text{otherwise} \end{cases}$$

where c is a constant whose depend on the weight of the given problem.

(i) If the weight of the given problem are thought of as costs then $c=\infty$.
(ii) If the weight of the given problem are capacities then $c=0$.



$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$



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

Note:

↳ For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.

↳ The adjacency matrix of an undirected graph is symmetric.

↳ The memory requirement of an adjacency matrix is $O(n^2)$, where n is the no. of nodes in a graph.

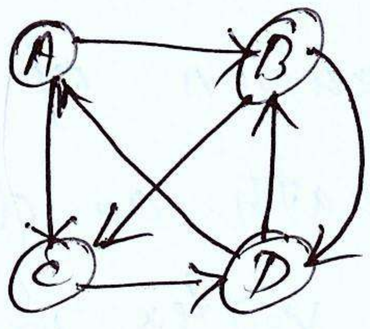
↳ No. of 1s in an adjacency matrix is equal to the edges connecting the nodes.

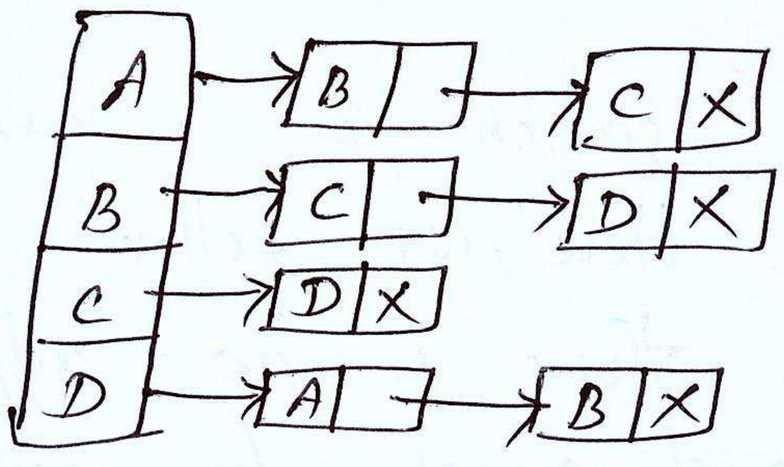
↳ The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

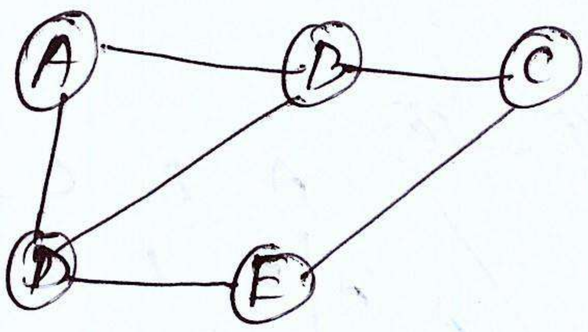
2) Adjacency List Representation:-

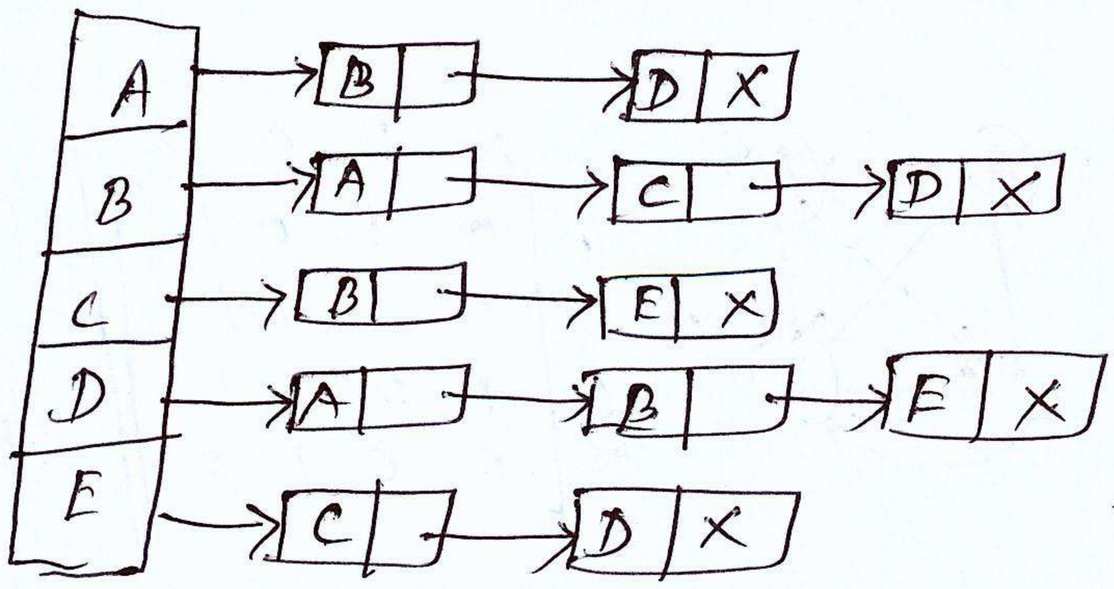
A graph containing m vertices and n edges can be represented using a linked list referred to as adjacency list.

The no. of vertices in a graph forms a singly linked list. Each vertex has a separate linked list, with nodes equal to the no. of edges connected from the corresponding vertex.



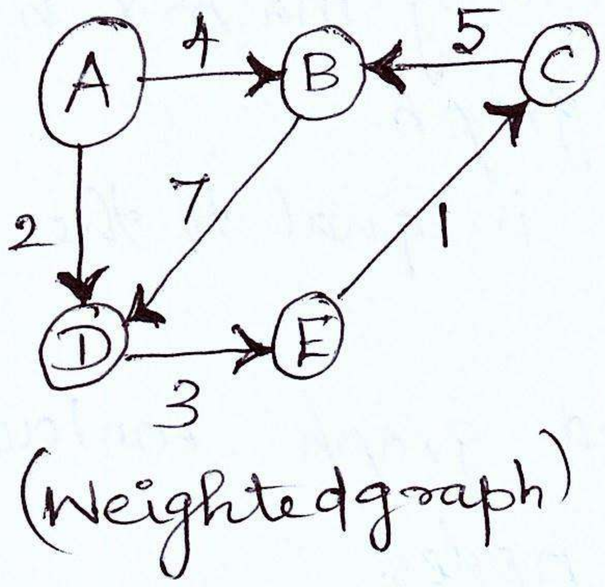




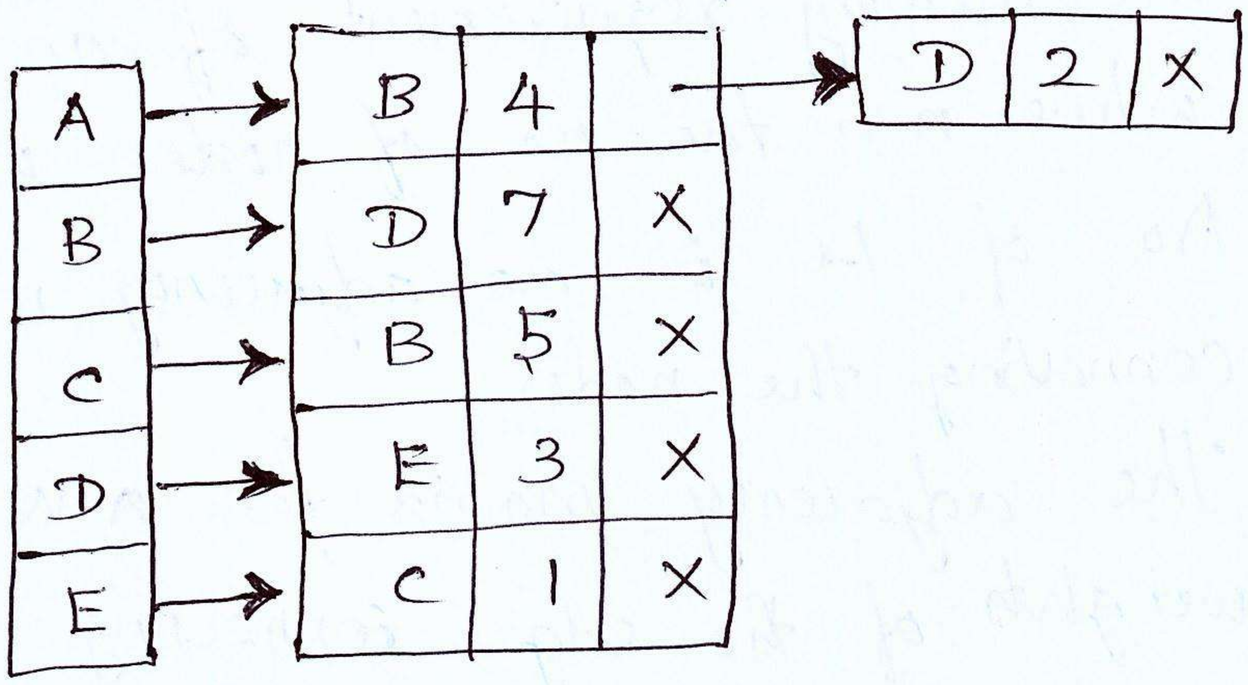


↳ For a directed graph, the sum of the lengths of all adjacency lists is equal to the no. of edges in G.

↳ For an undirected graph, the sum of lengths of all adjacency lists is equal to twice the no. of edges in G.



(Weighted graph)



Traversal of Graphs

Most of graph problems involve traversal of a graph. Traversal of a graph means visiting each node and visiting exactly once. Two commonly used techniques are:

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

1) Depth First Search:-

Depth-first search is a generalization of preorder traversal of tree. Here, we follow the path as deeply as we can go. When there is no adjacent vertex present, we traverse back and search for unvisited vertex. We will

maintain a visited array to mark all the visited vertices. In the case of DFS, the depth of a graph should be known.

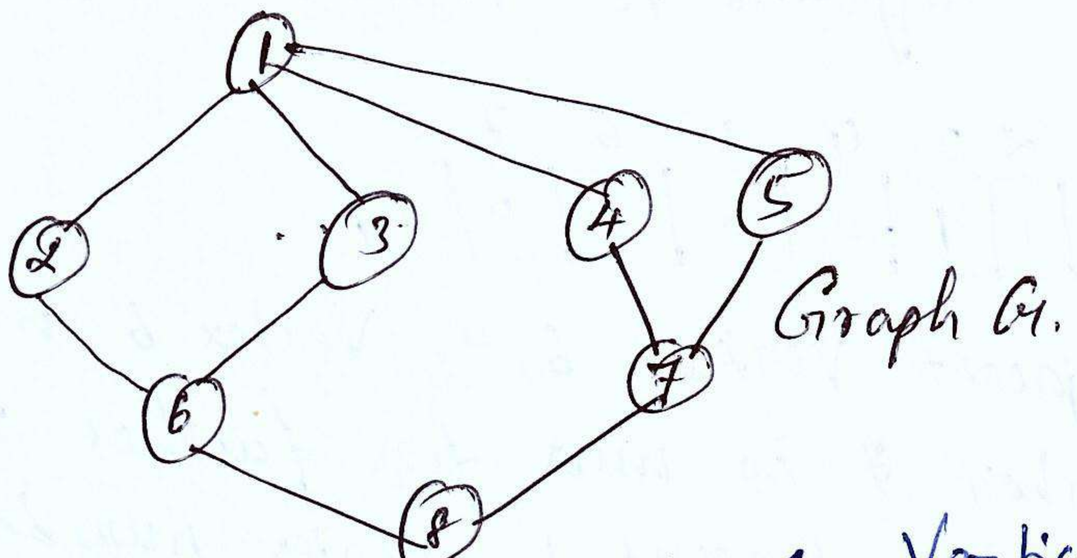
Procedure (Recursive);-

1). Select an unvisited vertex v, visit it. and treat it as the current node.
2). Search an unvisited adjacent of the current node, visit it and make it as new current node.
3). If the current node has no unvisited adjacents, back track to its parent and make it a new current node.
4). Repeat the step 2 and 3 until no more edges can be visited.
5). Repeat from step 1 for the remaining vertex.

Pseudocode (Recursive);-

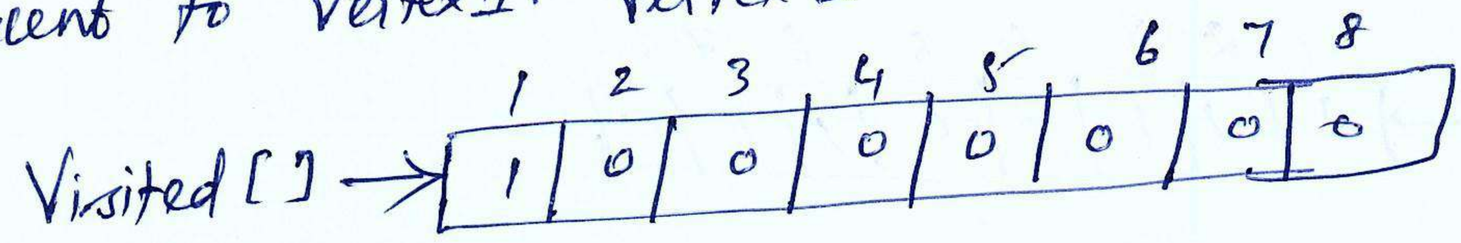
```
void Graph-DFS(Vertex v)
{
    v.visited = true;
    for each Vertex w adjacent to v
        if (!w.visited)
            DFS(w);
}
```

```
n = number of nodes
Initialize visited[] to zero (false)
for (i=0; i<n; i++)
    visited[i]=0
```

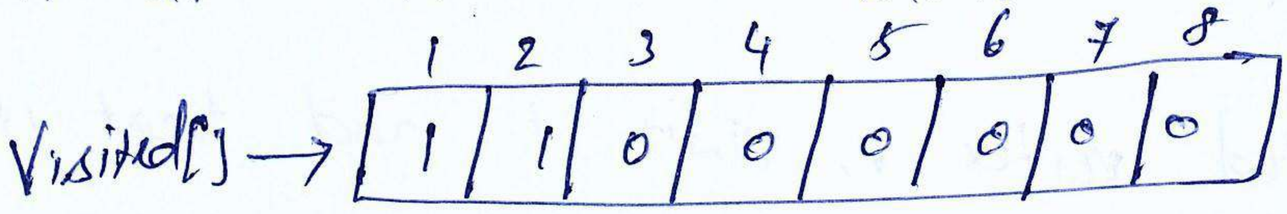


Graph G.

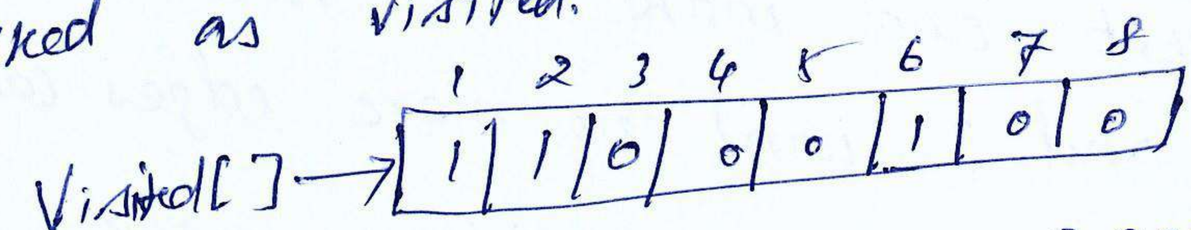
1) Traversal start from vertex 1. Vertices 2,3,4 and 5 are adjacent to vertex 1. Vertex 1 is marked as visited.



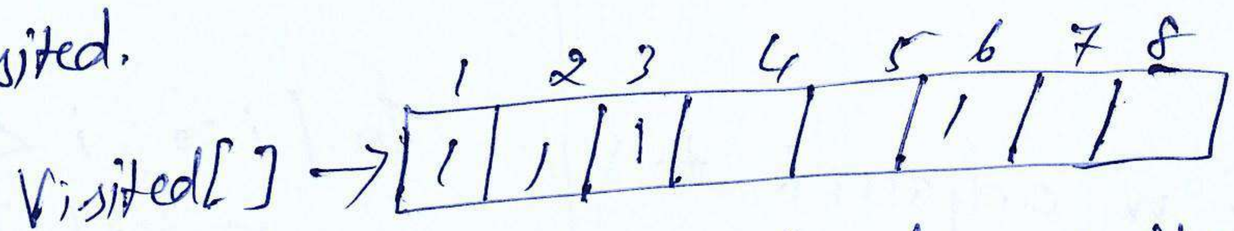
2) Out of the adjacent vertices 2, 3, 4 and 5, Vertex 2 is selected for further traversal. Vertices 1 and 6 are adjacent to vertex 2. Vertex 2 is marked as visited.



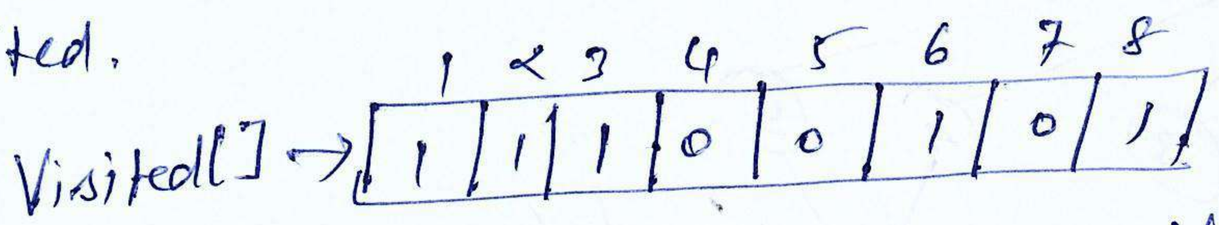
3) Out of the adjacent vertices 1 and 6, vertex 1 has already been visited. Vertex no 6 is selected for further traversal. Vertices 2, 3 and 8 are adjacent to vertex 6. Vertex 6 is marked as visited.



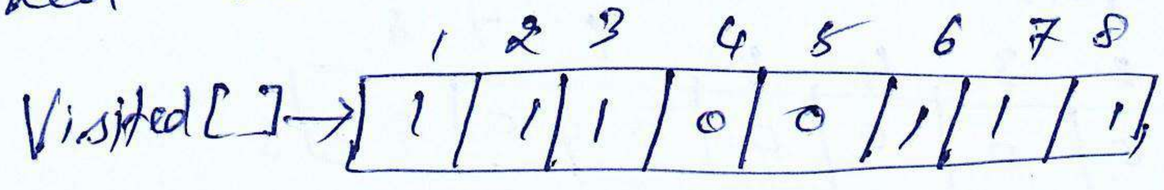
4) Out of the adjacent vertices 2, 3 and 8, vertex 2 is already visited. Vertex number 3 is used for further traversal. Vertices 1 and 6 are adjacent to vertex 3. Vertex 3 is marked as visited.



5). Vertices 1 and 6 are already visited. Therefore it goes back to vertex 6. Out of the adjacent vertices 2, 3, and 8; 2, and 3 are visited. It selects vertex 8 for further expansion. Vertices 6 and 7 are adjacent to vertex 8. Vertex 8 is marked as visited.



6) Out of the adjacent vertices 6, 7, Vertex 6 is already visited. Vertex number 7 is used for further traversal. Vertices 4, 5 and 8 are adjacent to vertex number 7. Vertex 7 is marked as visited.



7) Out of the adjacent vertices 4, 5 and 8, Vertex 4 is selected for further traversal. Vertices 1 and 7 are adjacent to vertex 4. Vertex 4 is marked as visited.

	1	2	3	4	5	6	7	8
Visited[ ] →	1	1	1	1	0	1	1	1

8) Adjacent vertices 1 and 7 are already visited. It goes back to vertex 7 and selects the next unvisited vertex 5 for further traversal. Vertex 5 is marked as visited.

	1	2	3	4	5	6	7	8
Visited[ ] →	1	1	1	1	1	1	1	1

DFS traversal sequence is 1, 2, 6, 3, 8, 7, 4, 5

Non-recursive DFS Traversal;-

Non-recursive DFS, uses a stack to remove recursion. All unvisited vertices adjacent to the one being visited are pushed into a stack. Traversal is continued by popping a vertex from the stack.

Pseudocode;- (Non-recursive)

```
DFS-Nonrecursive (vertex i)
{
    vertex w;
    stack s;
    initialize the stack s;
    push i in the stack s;
    while (stack is not empty)
    {
        i = pop(s);
        if (! visited[i])
        {
            visited[i] = 1;
            for each w adjacent to i
                if (! visited[w])
                    push w in the stack s;
        }
    }
}
```

Stack contents	Visited	Vertex visited	Action Status
NULL	0 1 2 3 4 5 6 0 0 0 0 0 0 0	—	Initial Status
top 0	0 1 2 3 4 5 6 0 0 0 0 0 0 0	—	Push the initial (first) vertex into the stacks
top 1 2	0 1 2 3 4 5 6 1 0 0 0 0 0 0	0	pop(0), visit(0), push adjacent vertices of 0 into the stack: 1, 2
top 1 3 6	0 1 2 3 4 5 6 1 0 1 0 0 0 0	0,2	pop(2), visit(2), push adjacent vertices of 2 into the stack: 3, 6
top 1 3 3 5	0 1 2 3 4 5 6 1 0 1 0 0 0 1	0,2,6	pop(6), visit(6), push adjacent vertices of 6 on to the stack: 3, 5
top 1 3 3	0 1 2 3 4 5 6 1 0 1 0 0 1 1	0,2,6,5	pop(5), visit(5), No adjacent vertices to vertex 5
top 1 3	0 1 2 3 4 5 6 1 0 1 1 0 1 1	0,2,6,5, 3	pop(3). No adjacent vertices to node 3
top 1	0 1 2 3 4 5 6 1 0 1 1 0 1 1	0,2,6,5, 3.	pop(3). Vertex 3 is already visited. Visit(3), push adjacent to 3: vertex 1
top 4	0 1 2 3 4 5 6 1 1 1 1 0 1 1	0,2,6,5, 3,1	visit(1). No adjacent vertex to 1.
NULL	0 1 2 3 4 5 6 1 1 1 1 1 1 1	0,2,6,5, 3,1,4	Traversal Complete

DFS traversal sequence is 0,2,6,5,3,1,4.

Features:-

Space Complexity:

The space complexity of a DFS is lower than that of BFS

Time Complexity:

The time complexity of a DFS is proportional to the no. of vertices plus the no. of edges in the graph. Therefore time complexity is $O(|V|+|E|)$.

Applications:-

Depth-first search is useful for:

↳ Finding a path b/n two specified nodes, u and v, of an unweighted graph as well as weighted graph.

↳ Finding whether a graph is connected or not.

↳ Computing the spanning tree of a connected graph.

2) Breadth First Search:-

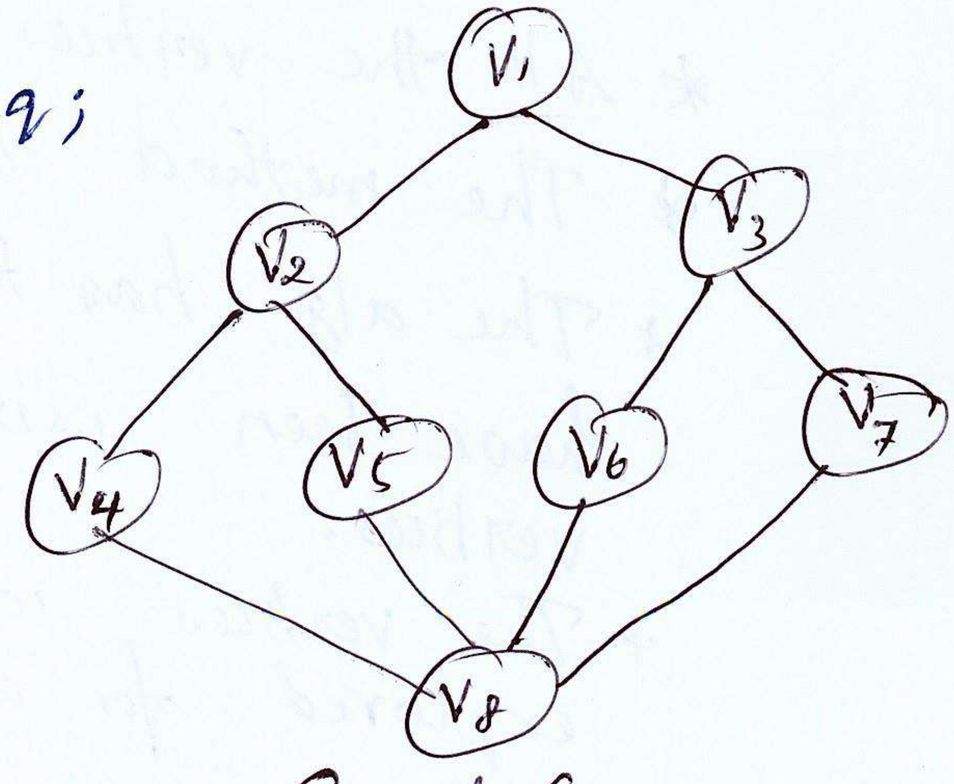
Breadth-first search (BFS) is a graph traversal alg that begins at the root node and explores all the neighboring nodes.

* This method start from a given vertex, say, 'Vo'.
* Vo is marked as visited.
* All the vertices adjacent to Vo are visited next.
* The method continues until all vertices are visited.
* The alg has to maintain a list of vertices which have been visited but not explored for adjacent vertices.
* The vertices which have been visited but not explored for adjacent vertices can be stored in queue.

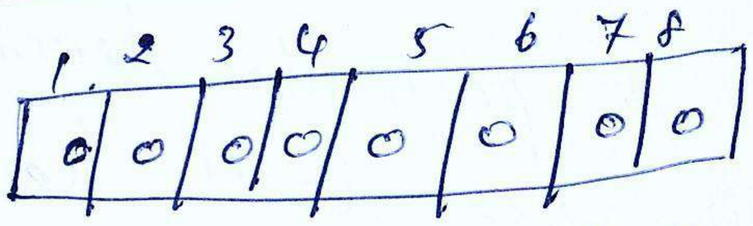
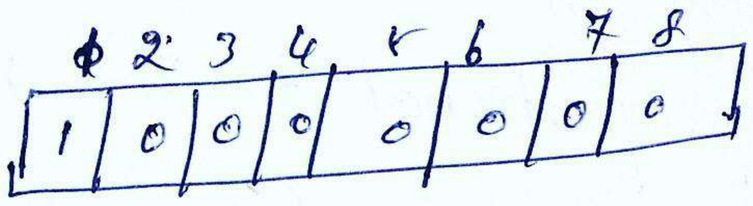
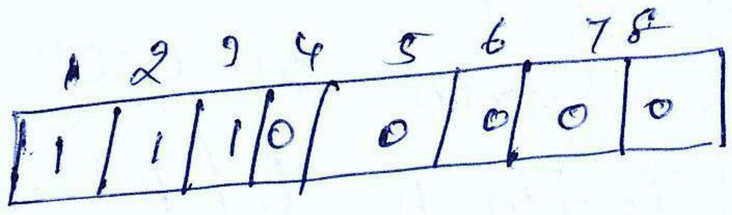
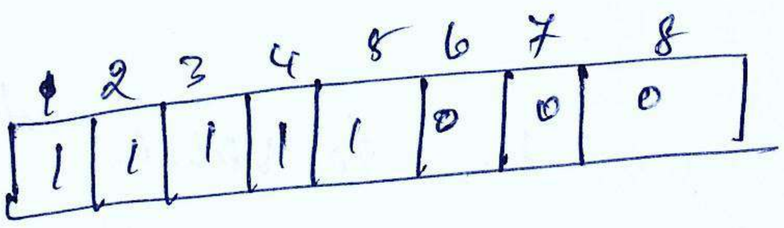
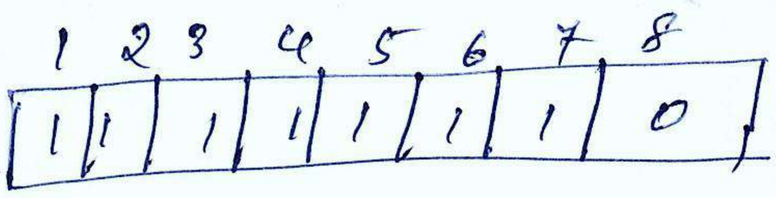
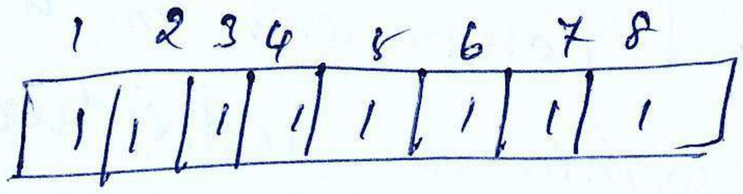
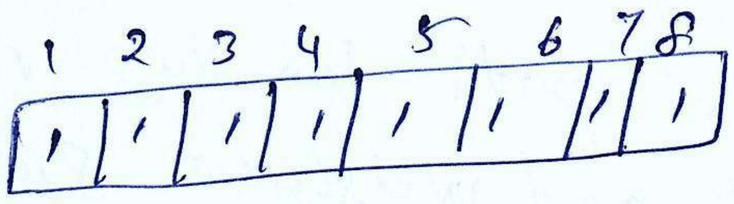
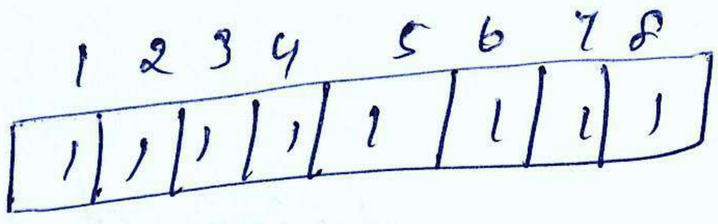
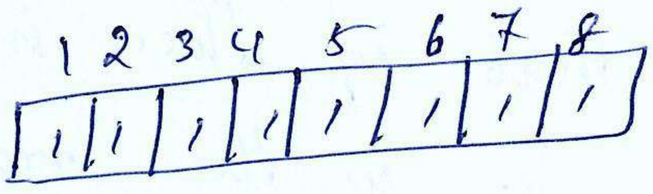
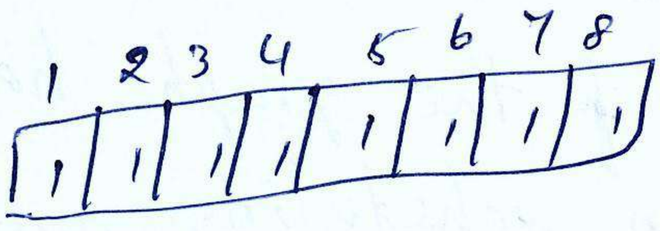
* Initially the queue contains the starting vertex
* In every iteration, a vertex is removed from the queue and its adjacent vertices which are not visited as yet are added to the queue.
* The alg terminates when the queue becomes empty.

Pseudocode - BFS

```
/* Array visited[] is initialized to 0 */
/* BFS traversal on Graph G is carried out begins at
                                    vertex V */
void BFS(int v)
{
   q: a queue type variable;
   initialize q;
   visited [v] = 1;     /* make v as visited */
   add the vertex v to queue q;
   while(q is not empty)
   {
      v ← delete an element from the queue;
      for all vertices w adjacent from v
      {
         if(! visited[w])
         {
            visited [w] = 1;
            add the vertex w to queue q;
         }
      }
   }
}
```



Graph G

Queue Content	Visited[ ]	Vertex Visited	Action Status
NULL		—	Initial status
V_1		V_1	add (q, V_1), visit(V_1).
V_2, V_3		V_1 V_2 V_3	delete (q, V_1), add and visit adjacent vertices
V_3 V_4 V_5		V_1 V_2 V_3 V_4 V_5	delete (q, V_2), add and visit adjacent vertices
V_4 V_5 V_6 V_7		V_1 V_2 V_3 V_4 V_5 V_6 V_7	delete (q, V_3), add and visit adjacent vertices
V_5 V_6 V_7 V_8		V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8	delete (q, V_4), add and visit adjacent vertices
V_6 V_7 V_8		V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8	delete (q, V_5)
V_7 V_8		V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8	delete (q, V_6), add and visit adjacent vertices.
V_8		V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8	delete (q, V_7)
NULL		V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8	Traversal complete

BFS traversal sequence is V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8.

Features:-

Space Complexity:-

In BFS, all nodes at a particular level must be saved until their child nodes in the next level have been generated The space complexity is proportional to the no. of nodes at the deepest level of the graph. Given a graph with branching factor b (no. of children at each node) and depth d, then space complexity is $O(b^d)$.

Time Complexity:-

In worst case, BFS has to traverse through all paths to all possible nodes. Time complexity of BFS is $O(|E|+|V|)$

Applications:-

↳ Finding all connected components in a graph G.
↳ Finding all nodes within an individual connected component.
↳ Finding the shortest path b/n two nodes, u and v, of an unweighted and weighted graph.

Topological Sort:-

A topological sort is an ordering of vertices in a directed acyclic graph such that if there is a path from V_i to V_j, then V_j appears after V_i in the ordering.

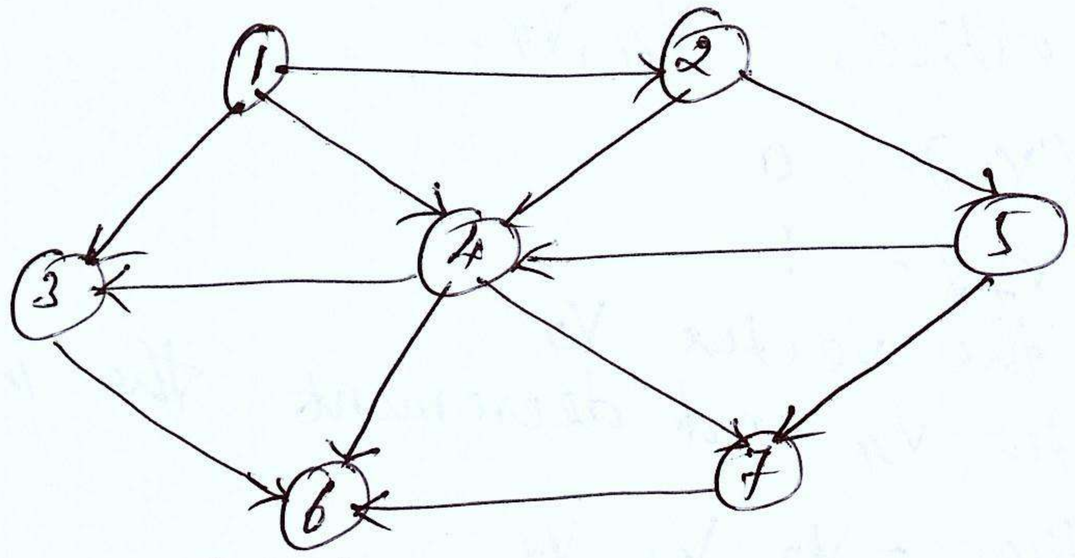
* It is not possible if the graph has a cycle.
* It is widely used in scheduling applications, jobs or tasks.
  ↳ The jobs that have to be completed are represented as nodes and there is an edge from node u to v.

if job u must be completed before job v can be started.

↳ A topological sort of such a graph given an order in which the given jobs must be performed.

Procedure:-

1). Find the indegree of every vertex in a graph.
2). Place the vertices whose indegree is 0 on the empty queue.
3). Delete the vertex V and decrement the indegree of all its adjacent vertices.
4). Increment the counter.
5). Insert the vertex on the queue, if its indegree is zero.
6). Repeat from steps 3 until the queue becomes empty.
7). If counter ! = No. of vertex in the graph, then display the message as "Cyclic Graph".



Graph G.

Steps:

1. Find the indegree of all the vertices.

Indegree [V1] = 0
Indegre [V2] = 1
Indegree [V3] = 2
Indegree [V4] = 3
Indegree [V5] = 1
Indegree [V6] = 3
Indegree [V7] = 2
~~Indegree [V8] =~~

2). Enqueue the vertex, whose indegree is 0. The vertex V_1's indegree is zero. So place it on the queue.

3). Delete the vertex V_1 from the queue.

4). Decrement the indegree of V_1's adjacent vertices.

Adjacent vertices of V_1 are V_2, V_3 and V_4.

Indegree [V_2] = 0

Indegree [V_3] = 1

Indegree [V_4] = 2.

Now enqueue the vertex V_2 as its indegree is 0.

5). Dequeue the vertex V_2 and decrement the indegree of its adjacent vertices - V_4, V_5.

Indegree [V_4] = 1

Indegree [V_5] = 0

Now enqueue the vertex V_5.

6). Dequeue the vertex V_5 and decrement the indegree of its adjacent vertices - V_4, V_7.

Indegree [V_4] = 0

Indegree [V_7] = 1

Enqueue the vertex V_4.

7). Dequeue the vertex V_4 and decrement the indegree of its adjacent vertices - V_3, V_6, V_7.

Indegree [V_3] = 0

Indegree [V_6] = 2

Indegree [V_7] = 0

Now Enqueue the vertices V_3, V_7.

8). Degnene the vertex V_2 and decrement its adjacent vertices's indegree. - V_6

Indegree $[V_6] = 1$.

9) Degnene the vertex V_7 and decrement indegree of it adjacent vertices. - V_6.

Indegree $[V_6] = 0$.

Enqnene the vertex V_6.

10). Degnene the vertex V_6.

Topological order for the graph is $V_1, V_2, V_5, V_4, V_3, V_7$ and V_6.

Vertex	Indegree Before Degnene #						
	1	2	3	4	5	6	7
V_1	0	0	0	0	0	0	0
V_2	1	0	0	0	0	0	0
V_3	2	1	1	1	0	0	0
V_4	3	2	1	0	0	0	0
V_5	1	1	0	0	0	0	0
V_6	3	3	3	3	2	1	0
V_7	2	2	2	1	0	0	0
Enqnene	V_1	V_2	V_5	V_4	V_3, V_7	-	V_6
Degnene	V_1	V_2	V_5	V_4	V_3	V_7	V_6

Routine:

```
void Topsort (Graph G)
{
    Queue Q;
    int counter=0;
    vertex V, W;
    Q = CreateQueue (NumVertex);
    MakeEmpty (Q);
    for each vertex V
        if (Indegree [V] ==0)
            Enqueue (V, Q);
    while (! IsEmpty (Q))
    {
        V = Dequeue (Q);
        Topnum [V] = ++ counter;
        for each W adjacent to V
            if (--Indegree [W] ==0)
                Enqueue (W, Q);
    }
    if (counter != NumVertex)
        Error ("Graph has a cycle");
    DisposeQueue (Q);
}
```

Algorithm Analysis:-

* The time to perform this alg is $O(|E|+|V|)$ if adjacency lists are used.
* The queue operations are done atmost once per vertex, and the initialization steps also take time proportional to the size of the graph.
  - size of the graph ⇒ total no. of edges

Applications of Graph :-

1). In circuit networks, where points of connection are drawn as vertices and component wires become the edges of the graph.

2). In transportation networks, where stations are drawn as vertices and routes become the edges of the graph

3). In maps that draw cities / states / regions as vertices and adjacency relations as edges.

4). In program flow analysis, where procedures / modules are treated as vertices and calls to these procedures are drawn as edges of the graph.

5). Once, we have graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.

6) In flowcharts or control-flow graphs, the statements & conditions in a program are represented as nodes and the flow of control is represented by the edges.

7) In state transition diagrams, the nodes are used to represent states, and the edges represent legal moves from one state to the other.

8). Graphs are also used to activity network diagrams. These diagrams are extensively used as a project management tool to represent the independent relationship b/n groups, steps and tasks that have a significant impact on the project.

9) In operating system, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource.

10) Graph theory is also used to study molecules in chemistry and physics.

11) Traffic flow can be modeled by a graph. Each street intersection represents a vertex, and each street is an edge.

12) In airport system, each airport is a vertex, and two vertices are connected by an edge if there is a nonstop flight from the airports that are represented by the vertices.

Bi-Connectivity :-

An undirected graph is said to be connected graph if for any pair of nodes of the graph, there exists a path b/n them. If it is able to visit all other vertices starting from any vertex of a graph, then we can say that the graph is connected. Fig. shows the connected graph.

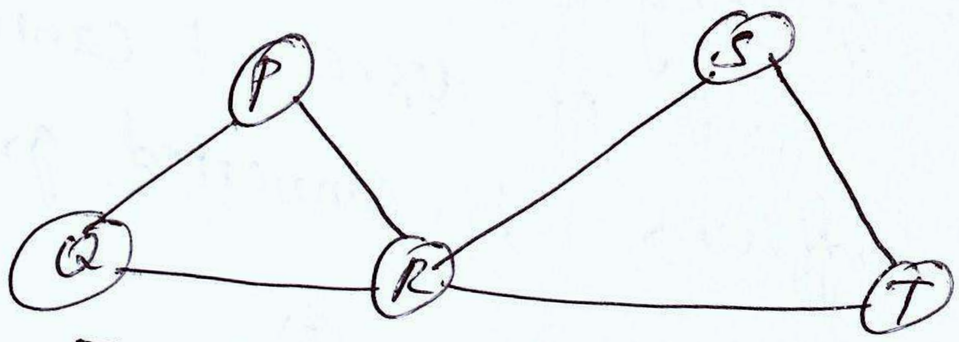


Fig. A Connected graph.

A connected undirected graph is <u>biconnected</u> if there are no vertices whose removal disconnects the rest of the graph.

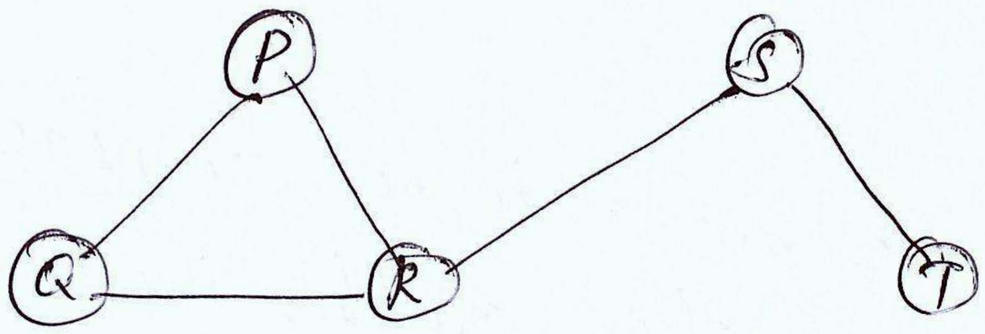
* If a vertex whose removal from the graph makes a disconnected graph then the vertex is referred to as <u>cut-vertex</u> or an <u>articulation point</u>.

* In the above Fig. "R" is an articulation point.

* If we remove the articulation point "R", then the connected graph becomes two disconnected graphs.

* If the removal of an edge disconnects the graph, then the edge is referred to as a <u>bridge</u>.

* A bridge will have at least one articulation point at its end, but an articulation point doesn't necessarily need to be linked to a bridge.



(R,S) and (S,T) are bridges

Fig. A connected graph

A graph with no articulation points and no bridges is referred to as <u>bi-connected graph.</u> In other words, a connected undirected graph is said to bi-connected, if the graph is still connected after removing any one vertex.

* Any graph containing a node of degree 1 cannot be bi-connected. Fig. shows the different bi-connected graphs.

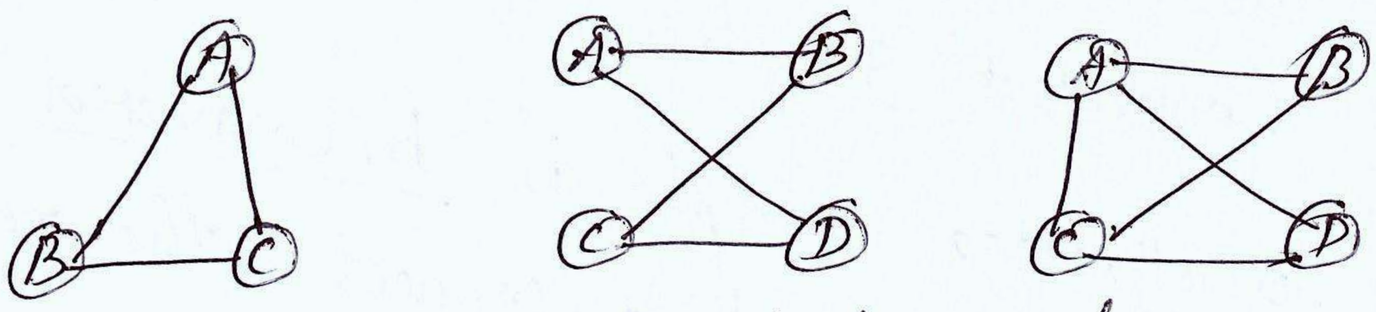
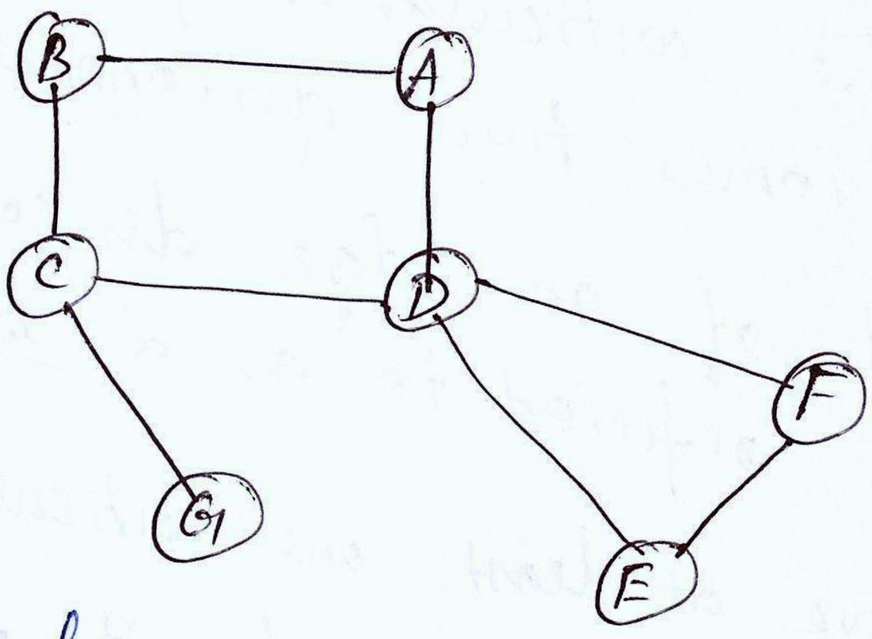


Fig. Bi-connected graphs

* The following graph in Fig① is not biconnected: C and D are articulation points.

* The removal of C would disconnect G, and the removal of D would disconnect E and F, from the rest of the graph.



Fig①. A graph with articulation points C and D.

* A depth first search alg can be used to find the articulation points of a graph, and hence find the biconnected components in a graph.

Depth-first search provides a linear-time alg to find all articulation points in a connected graph.

* First, starting at any vertex, we perform a DFS and number of nodes as they are visited.

* For each vertex v, we call this preorder number, Num(v).

* Then, for every vertex v in the depth-first search spanning tree, we compute the lowest-numbered vertex, which we call Low(v), that is reachable from v by taking zero or more tree edges and then possibly one back edge (in that order).

* The depth-first search tree in Fig ② shows the preorder number first, and then the lowest-numbered vertex reachable

Num(v)

Low(v)

under the rule.

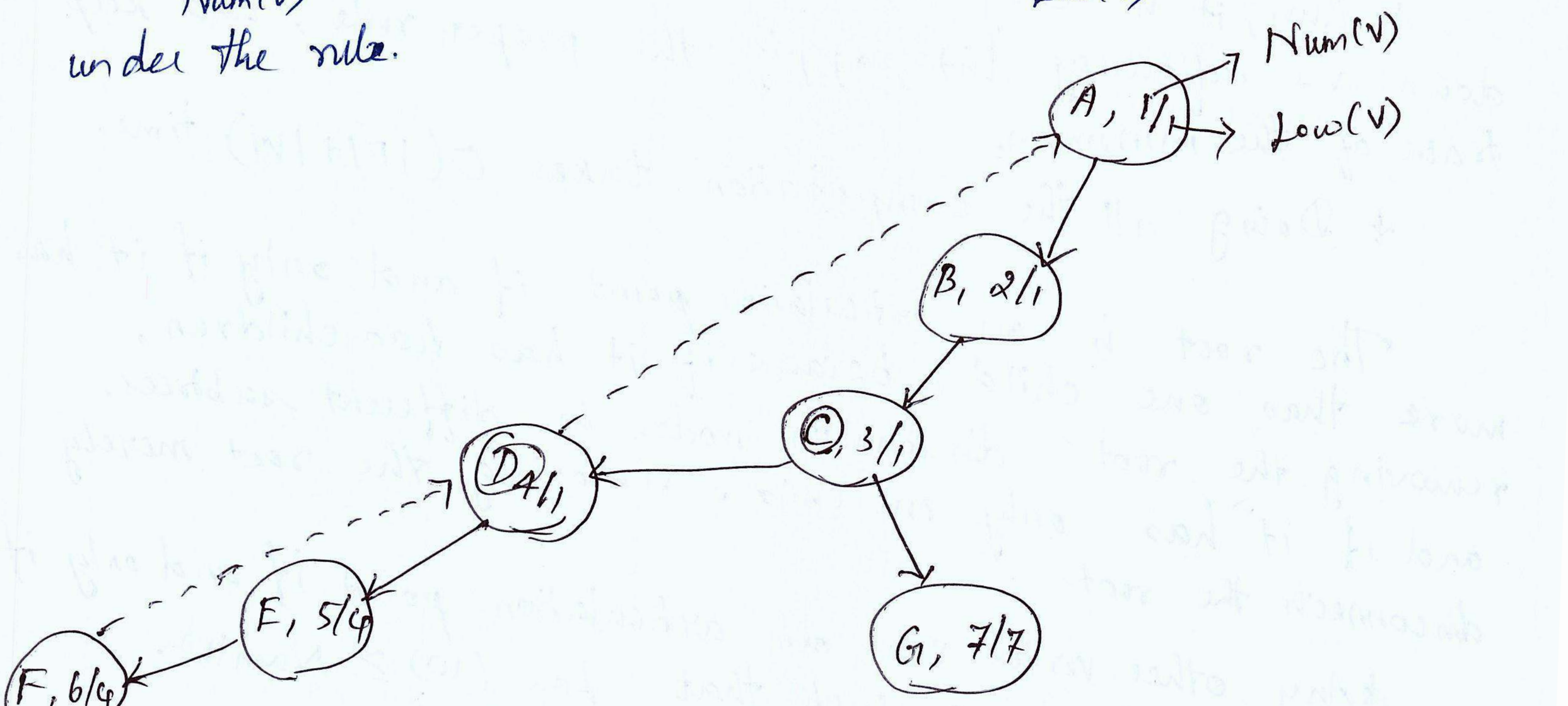


Fig ②. Depth-first tree for previous graph, with Num and Low

The lowest-numbered vertex reachable by A, B, and C is vertex 1 (A), because they can all take tree edges to D and then one back edge back to A.

* We can efficiently compute Low by performing a postorder traversal of the depth-first spanning tree.

By the definition of Low, Low(v) is the minimum of

1. Num(v)
2. the lowest Num(w) among all back edges (v, w)
3. the lowest Low(v) among all tree edges (v, w).

* The first condition is the option of taking no edges, the second way is to choose no tree edges and a back edge, and the third way is to choose some tree edges and possibly a back edge.

* For any edge (v, w), we can tell whether it is a tree edge or a back edge. merely by checking Num(v) and Num(w)

* Thus, it is easy to compute Low(v): we merely scan down v's adjacency list, apply the proper rule, and keep track of the minimum.

* Doing all the computation takes O(|E|+|V|) time.

The root is an articulation point if and only if it has more than one child, because if it has two children, removing the root disconnects nodes in different subtrees, and if it has only one child, removing the root merely disconnects the root

* Any other vertex v is an articulation point if and only if v has some child w such that Low(w) ≥ Num(v).

* If you see the example in Fig (2), Low(E) ≥ Num(D), since both are 4.

* If you take another example, Low(G) ≥ Num(C). Therefore. C and D is an articulation point / cut vertices.

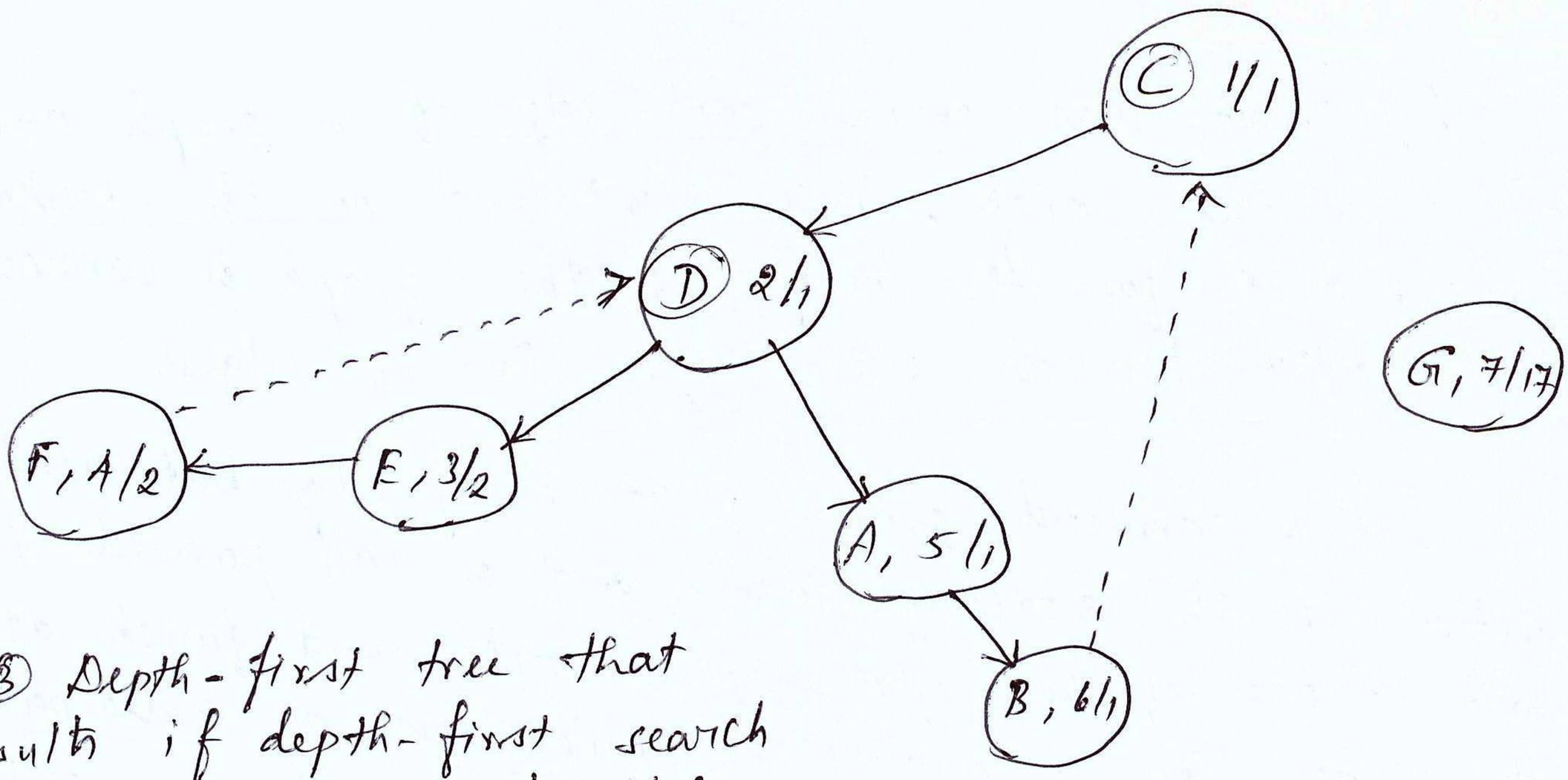


Fig. ③ Depth-first tree that results if depth-first search starts at C.

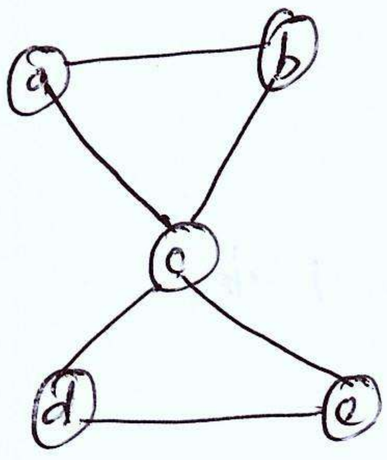
Pseudcode - To find Articulation Point

```
void FindArt (Vertex V)
{
  Vertex W;
  Visited[V] = True;
  Low[V] = Num[V] = Counter++ ;   /* Rule 1 */
  for each W adjacent to V
  {
    if(! Visited[W])   /* foward edge */
    {
      Parent[W] = V;
      FindArt(W);
      if (Low[W] >= Num[V])
        printf("%v is an articulation point\n", V);
      Low[V] = Min (Low[V], Low[W]);   /* Rule 3 */
    }
    else if (Parent[V] != W)   /* Back edge */
      Low[V] = Min (Low[V], Num[W]);   /* Rule 2 */
  }
}
```

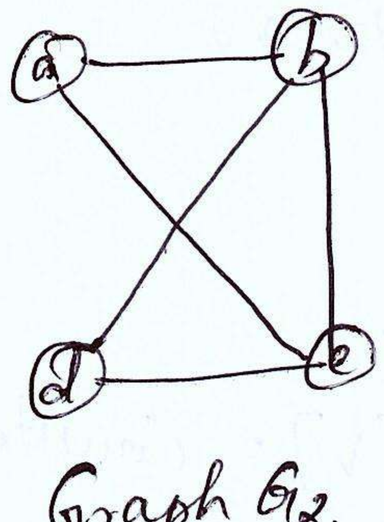
Euler Circuit :-

An Euler circuit that covers each edge of a graph once but not more than once, which must end on its starting vertex and is possible only if the graph is connected & each vertex has an even degree (no. of edges).

In a connected graph, visit every edge but need not return to its starting vertex, and is still possible if we start at one of the odd-degree vertices and finish at the other vertex. This is known as Euler Tour or Euler path. If more than two vertices have odd degree, then an Euler tour is not possible.



Graph G_1,
Euler Circuit : a, c, d, e, c, b, a



Graph G_2.
Euler path : b, a, e, d, b, e

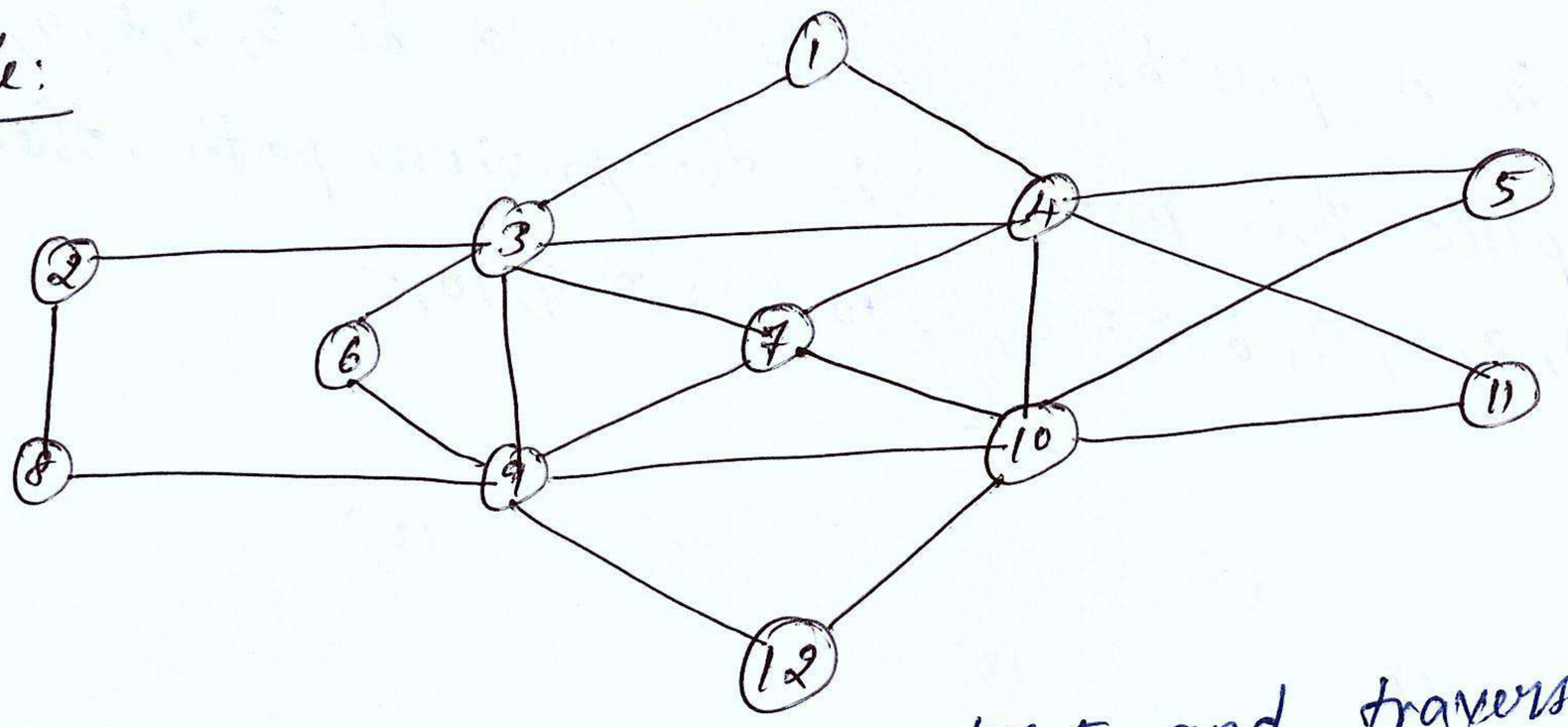
Note :-

* A connected undirected graph has an Euler circuit if each vertex is of even degree.
* A connected undirected graph has an Euler path/tour (not a cycle) if it has exactly two vertices of odd degree.
* A digraph has an Euler circuit if it is strongly connected and Indegree [v] = Outdegree [v] for all vertices.

Procedure - Euler Circuit (G is a connected graph with even edges)

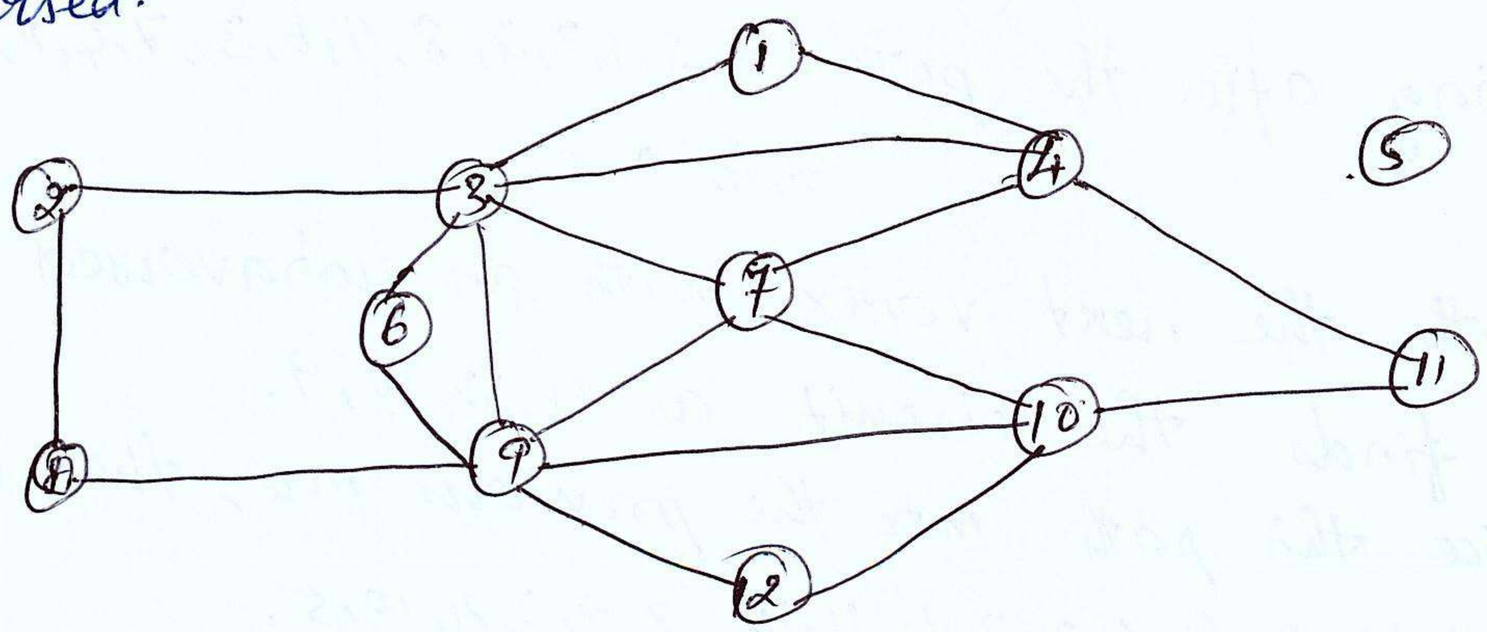
1). We start a proper vertex and construct a cycle.
2). We decrease the degree of the vertex each time we visit it.
3). If this cycle contains all edges of the graph, then stop.
4). Otherwise, select a vertex of degree greater than 0 (that belongs to the graph and to the cycle) and construct another cycle.
5). Next, splice these two cycles into one cycle.
6) If a join cycle contains all edges of the graph, then stop.
7) If not, select a common vertex of degree greater than 0 and construct a cycle and do the step 5.

Example:



Graph for Euler circuit problem

1) Suppose we start at vertex 5, and traverse the circuits, 4, 10, 5. Then we are struck, and most of the graph is still untraversed.

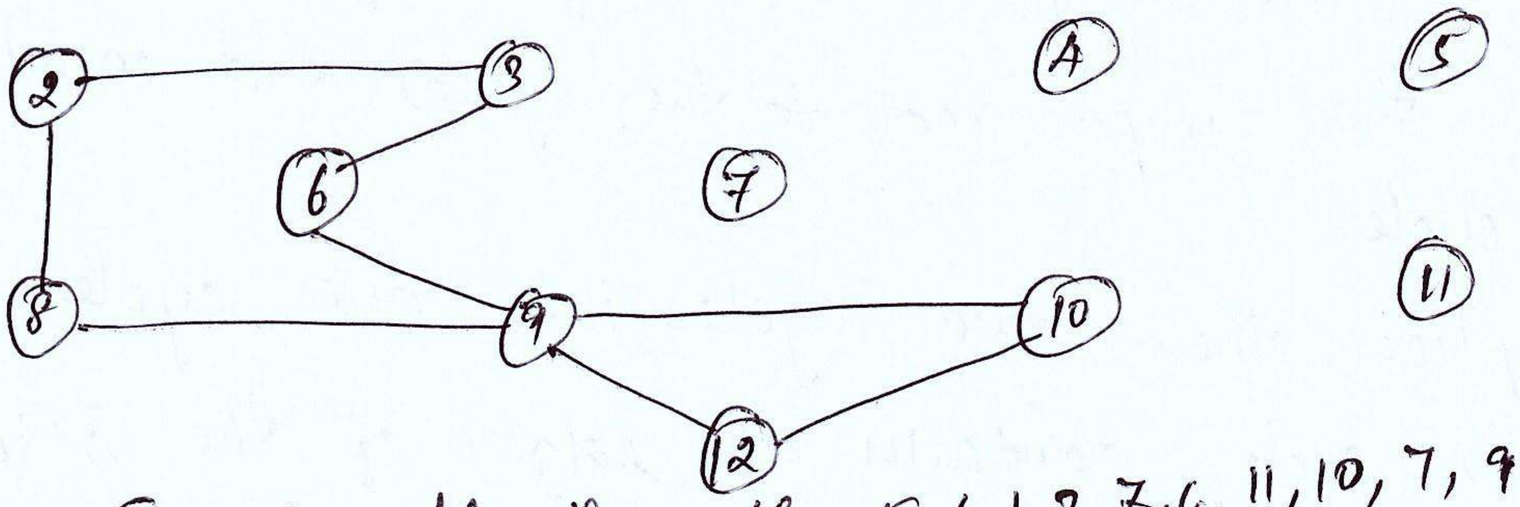


Graph remaining after 5, 4, 10, 5

2) We then continue from vertex 4. A DFS might come up with the path 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4.

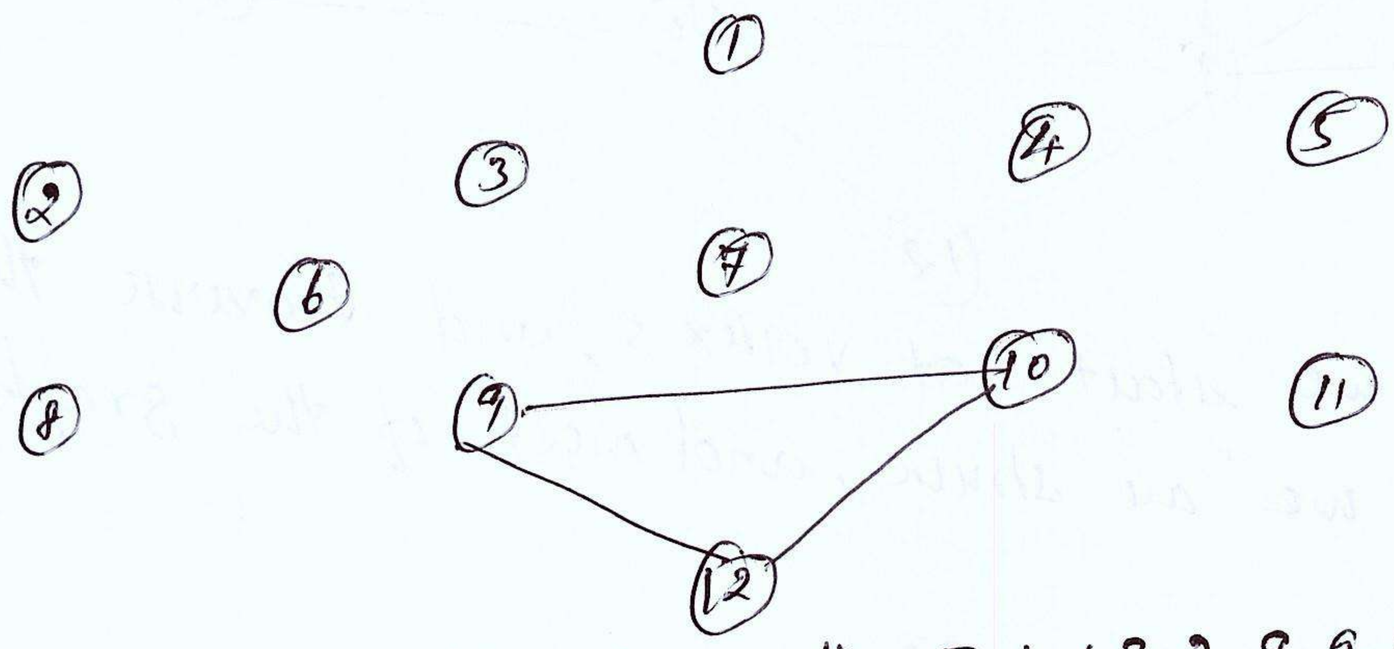
If we splice this path into the previous path of 5, 4, 10, 5, then we get a new path 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

(1)



Graph after the path 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

3) The next vertex on the path that has untraversed edges is vertex 2. A possible circuit would be 3, 2, 8, 9, 6, 3. If we splice this path into the previous path, then we get 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5



Graph remaining after the path 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

4) On this path, the next vertex with an untraversed edge is 9, and the alg finds the circuit as 9, 12, 10, 9. If we splice this path into the previous one, then we get 5, 4, 1, 3, 2, 8, 9, 12, 10, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

As all the edges are traversed, the alg terminates with an Eulers circuit.

The running time of this alg is $O(|E|+|V|)$.