Unit:-6 Exploring Graph

* Topics :-

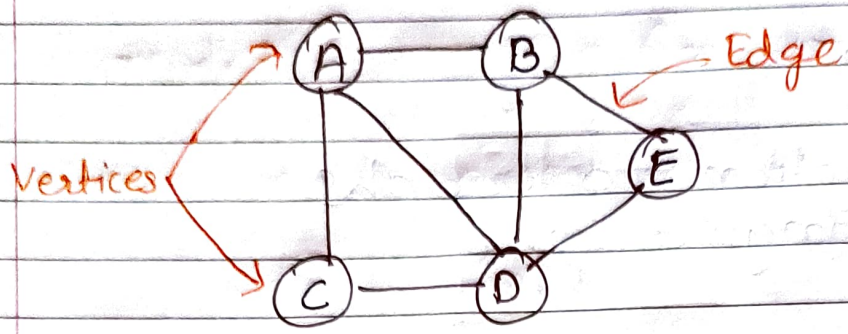
- Introduction to Graphs
- Traversing Graphs
  1. DFS
  2. BFS
- Topological sort
- Connected Components
  1. Strongly Connected Components
  2. Graph Components
     - (i) Articulation Point
     - (ii) Bi-Connected Component

* Introduction to Graphs :-

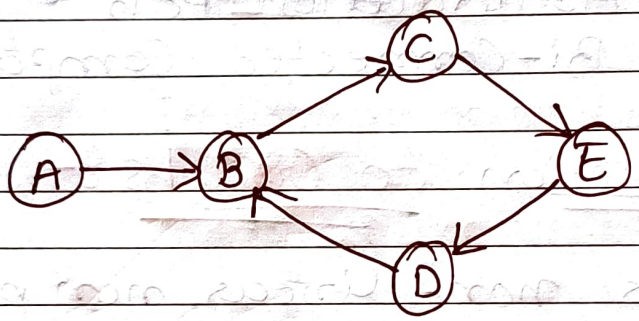
- Graph is non-linear, non primitive data structure, used in many applications in computer science.
- Game playing, file system, shortest path and many other problems are effectively formulated and solved using graph.

(I) **Graph** :- A Graph is a collection of nodes and edges. $G = (V, E)$

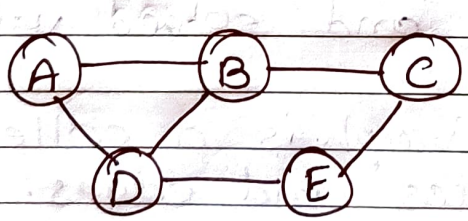
V = nodes (vertices, points)
E = edges (links, arcs) bet^n pair of nodes.



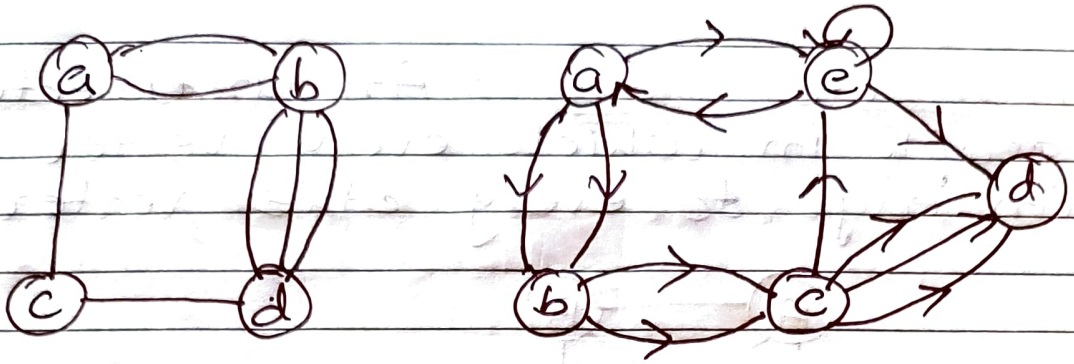
(2) Directed Graph :- A graph in which every edge is directed from one node to another is called a directed graph or digraph.



(3) Undirected Graph :- A graph in which every edge is undirected and no direction is associated with them is called an undirected graph.

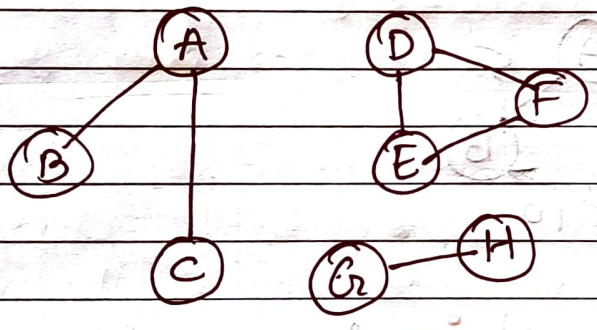


(4) Multigraph :- A multigraph is a graph which is permitted to have multiple edges, that is, edges that have the same end nodes. Thus two vertices may be connected by more than one edge.

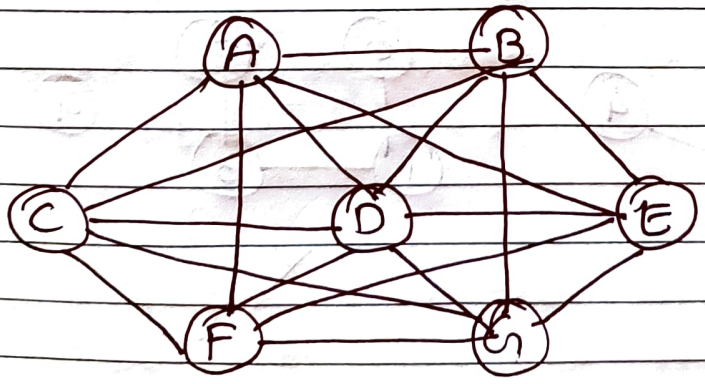


(5) Sparse Graph :- It is a graph in which the no. of edges is close to the minimal no. of edges.

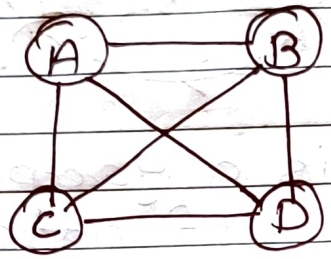
- Sparse Graph can be a disconnected graph.



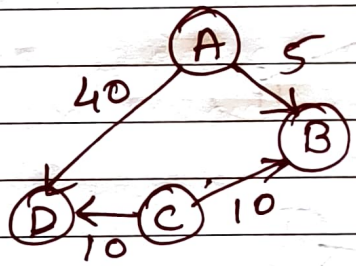
(6) Dense Graph :- It is a graph in which the no. of edges is close to the maximal no. of edges.



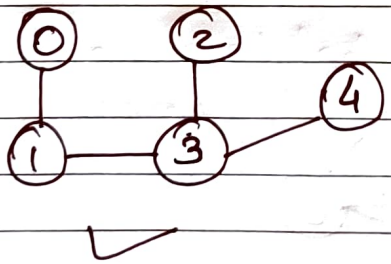
(7) **Complete Graph:-** An undirected graph in which every vertex has an edge to every other vertex.



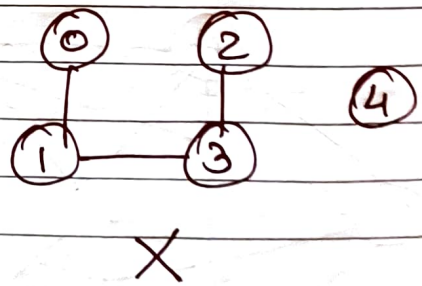
(8) **Weighted Graph:-** It is a graph for which each edge has an associated weight.



(9) **Connected Graph:-** An undirected graph is called connected graph if every vertex is reachable from all other vertices.



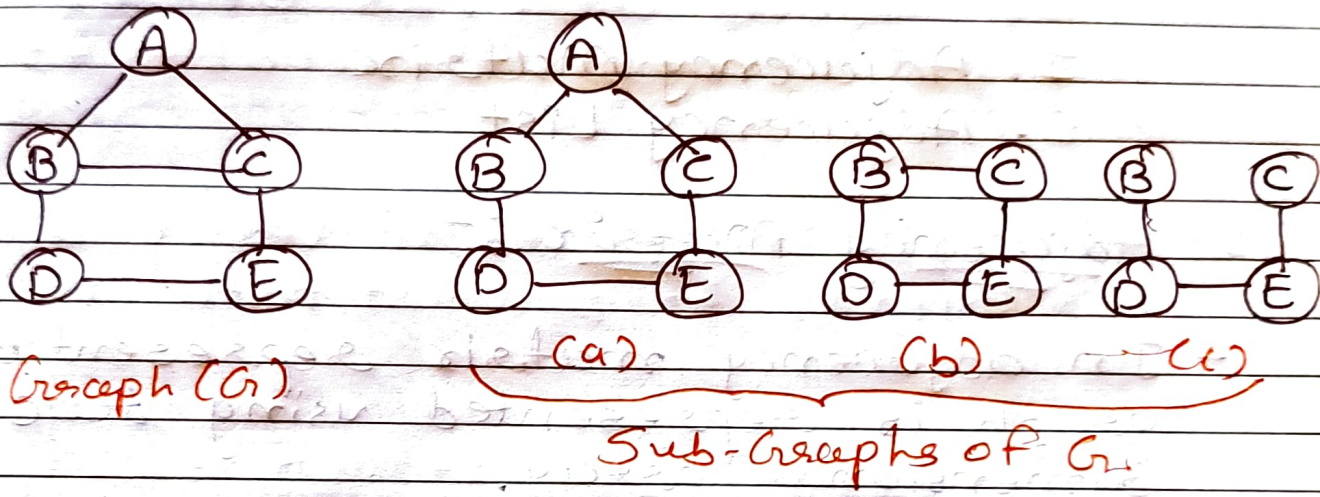
✓



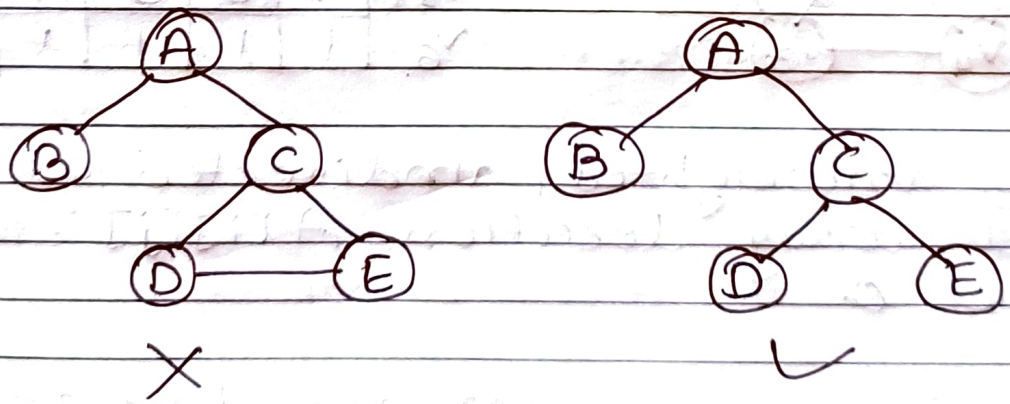
✗

(10) Sub Graph:- Vertex and edge sets are subsets of those of G.

- A super graph of a graph G is a graph that contains G as a sub-graph.



(11) Trees:- An undirected graph is a tree if it is connected and does not contain a cycle. (Connected Acyclic Graph.)



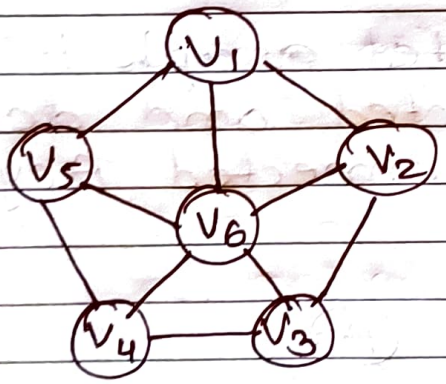
* Representation of Graph :-

- Graph can be represented using many different ways. Here we will discuss here two commonly used representations.

1. Adjacency Matrix
2. Adjacency List

[1] Adjacency Matrix :-

- In adjacency matrix representation, graph is represented using two dimensional array.



	V1	V2	V3	V4	V5	V6
V1	0	1	0	0	1	1
V2	1	0	1	0	0	1
V3	0	1	0	1	0	1
V4	0	0	1	0	1	1
V5	1	0	0	1	0	1
V6	1	1	1	1	1	0

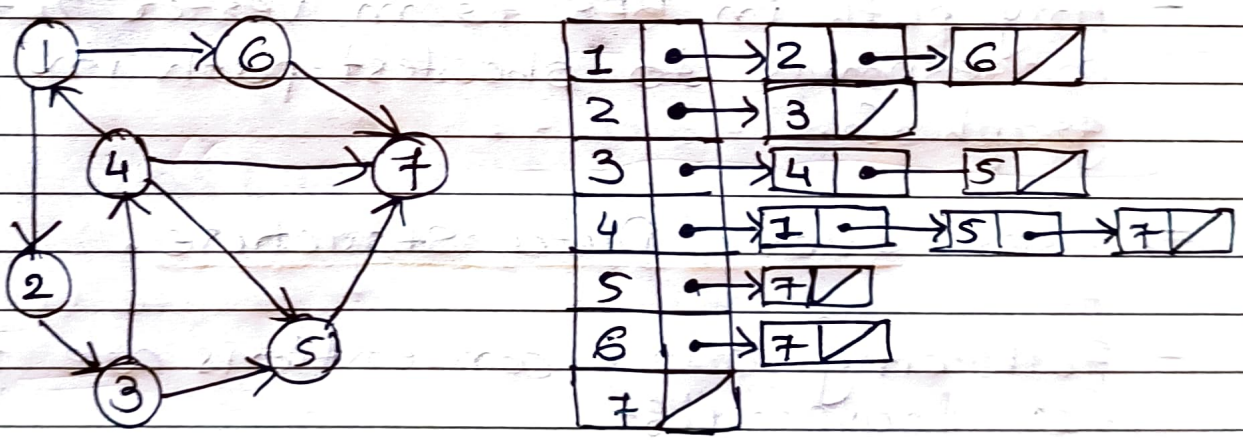
- For unweighted, undirected graph, each array location A[i][j] stores 1 or 0.

- If there exists an edge between vertices Vi and Vj, set A[i][j] = 1, else set it 0.

- For weighted graph, if there exists an edge between vertices V_i and V_j, set $A[i][j] = W_{ij}$, else set it 0. where, W_{ij} indicates weight of edges joining vertices $V_i V_j$.

[2] Adjacency List :-

- Adjacency list uses linked list to represent graph.



* Traversing Graphs :-

- Like tree, we can traverse graph in various way. ~~Traversing~~
- Traversing order has different applications
- Traversal of graph creates a tree as Forest.
- Two types of algorithm :- (1) DFS
  (2) BFS

(I) Breath First Search :-

⇒ Applications :- (1) Prim's algorithm
(2) Dijkstra's algorithm.

- For graph $G = (V, E)$, BFS starts from some source vertex s and explores it's all neighbors reachable from s.

- Any path in BFS from vertex u to v corresponds to shortest path in graph G.

⇒ Uses Queue Data structure.

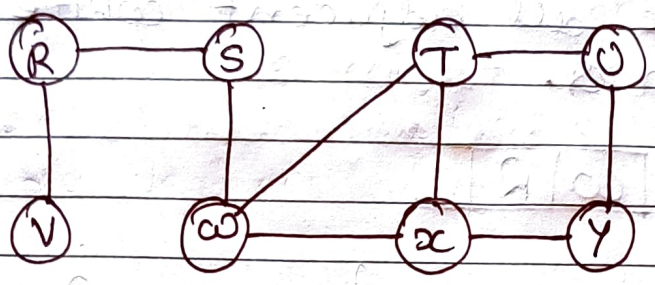
- Following color conventions are used for traversal :-

→ White :- Undiscovered state. Initially all nodes are in undiscovered state and hence white.

→ Gray :- When algorithm encounters node first time, it is inserted in queue and converted white to gray.

→ Black :- Fully visited. When node v is removed from queue and all its neighbours are inserted in queue, node v is called fully explored / visited and converted gray to Black.

⇒ Example :-



- We use ○ sign for white node.
- We use ◎ sign for gray node.
- We use ⊘ sign for Black node.

Step 1 :- Choosing ● as starting node,

color [u] = white
π[u] = nil [∵ π is used to represent parent of particulars selected node]
d[u] = ∞ [∵ d is used to represent distance]

Q | |

- Initially all vertices are white (unvisited), parent values are nil and distance is ∞.

→ Consider S as source node,

color [s] = Grey
π[s] = nil
d[s] = 0

Q | S |

We enqueue the source vertex s in the queue Q.

Step 2:- Dequeue s from Q.
→ Enqueue all adjacent white vertices of s in Q.

Q [ω | R]

color[v] = Grey
d[v] = d[s] + 1
= 0 + 1
= 1
π[v] = s

} values of neighbours of S - (ω & R)

↳ color[s] = Black.

Step 3:- Dequeue ω from Q.
→ Enqueue all adjacent white vertices of ω in Q.

Q [R | T | x]

color[v] = Grey
d[v] = d[ω] + 1
= 1 + 1
= 2
π[v] = ω

} values of neighbours of ω (T & x)

↳ color[ω] = Black

Step 4:- Dequeue R from Q
→ Enqueue all adjacent white vertices of R in Q.

Q [T | x | v]

color[v] = Grey
d[v] = d[R] + 1
= 1 + 1
= 2
π[v] = R

} values of neighbours of R (v)

↳ color[R] = Black

Step 5:- Dequeue T from Q.
→ Enqueue all adjacent white vertices of T in Q.

Q | x | v | u |

color[v] = Grey
d[v] = d[T] + 1
= 2 + 1
= 3
π[v] = T

} values of neighbours of T (u)

↳ color[T] = Black.

Step 6:- Dequeue x from Q
→ Enqueue all adjacent white vertices of x in Q.

Q | v | u | y |

color[v] = Grey
d[v] = d[x] + 1
= 2 + 1 = 3
π[v] = x

} values of neighbours of x (y)

↳ color [x] = Black

Step 7:- Dequeue V from Q
↳ There is no adjacent white vertices to V.

↳ color [V] = Black.

Step 8:- Dequeue U from Q.
→ There are no adjacent white vertices to U

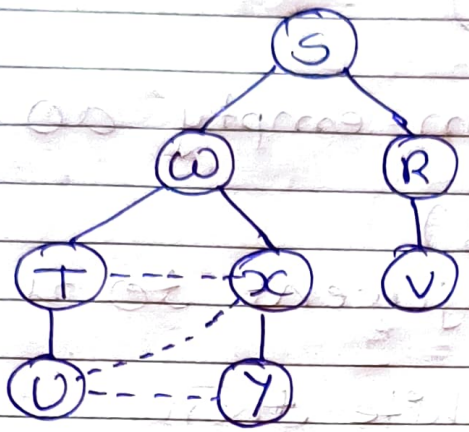
↳ color [U] = Black

Step 9:- Dequeue Y from Q
→ There are no adjacent white vertices to Y.

↳ color [Y] = Black

⇒ → Now all vertices are visited. Make BFS tree By visited sequence.

step 10 :- BFS tree.



⇒ BFS Sequence :- SWRTXVUY

→ '—' Solid line is consides as Tree edge.

→ '---' Cross Edge (not covered in a tree but given in graph)

⇒ Algorithm :-

Algorithm BFS (G, S) ← S: starting point; G: given graph

// for each vertex u ∈ G.V - {S} do

color[u] ← white
d[u] ← ∞
π[u] ← NIL

color[s] ← Grey
d[s] ← 0
π[s] ← NIL

Q ← { }
ENQUEUE (Q, s)

//while Q is non-empty do

u ← DEQUEUE (Q)
//for each v adjacent to u do

if color[v] ← white then
color[v] ← Grey
d[v] ← d[u] + 1
π[v] ← u
ENQUEUE (Q, v)

color[u] ← Black

⇒ Time complexity :- O(|V| + |E|)

(2) Depth First Search (DFS) :-

⇒ Uses stack data structure.

⇒ Classification of DFS edges :-

(1) Tree edge :- A tree edge is an edge that is included in DFS tree.

(2) Back edge :- An edge from a vertex 'u' to one of its ancestors 'v' and grey at that time is called as a back edge.

- A self loop is considered as a back edge.

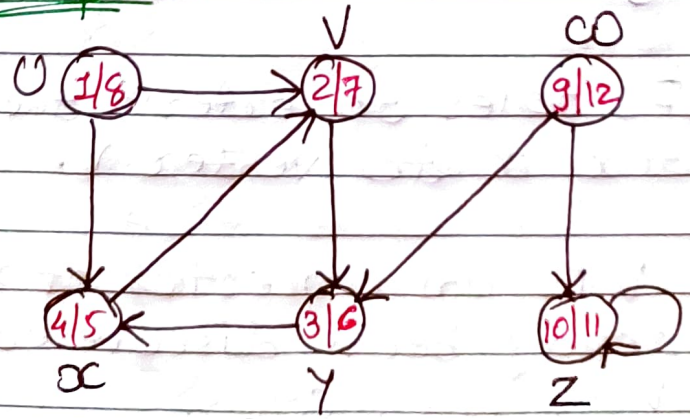
③ Forward edge :- An edge from a vertex 'u' to one of its descendents 'v' is called as a forward edge.

- A forward edge is discovered when DFS tries to extend the visit from a vertex 'u' to a vertex 'v' and finds that color(v) = Black and d(v) > d(u)

④ Cross edge :- An edge from a vertex 'u' to a vertex 'v' that is neither its ancestor nor its descendent is called as a cross edge.

- A cross edge is discovered when DFS tries to extend the visit from a vertex 'u' to a vertex 'v' and finds that color(v) = Black and d(v) < d(u)

⇒ Example :-



Step 1:- Initially, color[u] = white
$\pi[u] = NIL$
time = 0

→ For source vertex U,

color[U] = Grey
time = 0+1 = 1
d[U] = 1

Step 2:- For vertex V,

$\pi[V] = U$
color[V] = Grey
time = 1+1 = 2
d[V] = 2

Step 3:- For vertex Y,

color[Y] = Grey
$\pi[Y] = V$
time = 2+1 = 3
d[Y] = 3

Step 4:- For x,

color[x] = Grey
$\pi[x] = Y$
time = 1+3 = 4
d[x] = 4

Step 5:- When DFS tries to extend the visit from vertex x to vertex V,

(1) Vertex V is an ancestor of x since it has already been discovered.
(2) Vertex V is Grey color.

So, x-V is Back edge.

step 6:- color[x] = Black
time = 4+1 = 5
f[x] = 5 (finishing time)

step 7:- color[y] = Black
time = 5+1 = 6
f[y] = 6

step 8:- color[v] = Black
time = 6+1 = 7
f[v] = 7

step 9:- when DFS tries to extend the visit from v to x,
① color[x] = Black
② d[x] > d[v]

so, v-x is forward edge.

step 10:- color[v] = Black
time = 7+1 = 8
f[v] = 8

step 11:- color[w] = Grey
time = 8+1 = 9
π[w] = Nil
d[w] = 9

step 12:- when we visit w to y
① color[y] = Black
② d[y] < d[w]

so, w-y is cross edge.

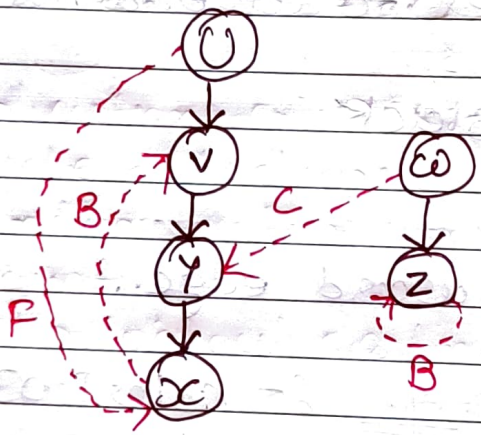
step 13:- color[z] = Grey
π[z] = w
time = 9+1 = 10
d[z] = 10

Step 14:- z-z is self edge, self edges are Back edge.

Step 15:- color[z] = Black
time = 10+1 = 11
f[z] = 11

Step 16:- color[w] = Black
time = 11+1 = 12
f[w] = 12

Step 17:- DFS Tree.



⇒ **Algorithm:-**

DFS (G)

```
for each vertex u in V[G] do
    color[u] ← white } for loop
    π[u] = NIL
time ← 0
for each vertex u in V[G] do
    if color[u] ← white then
        Depth-First-search(G, u)
```

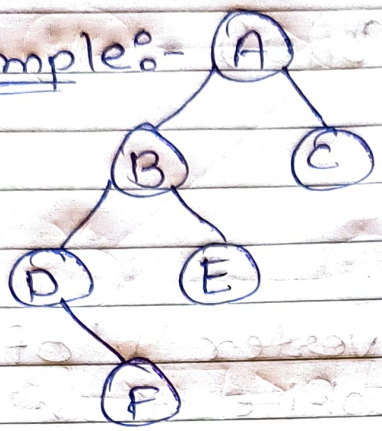
Depth-First-Search (G, u)

time ← time + 1
d[u] ← time
color[u] ← Grey
for each adjacent vertex 'v' of 'u' do
    if color[v] == white then
        set π[v] = u
        Depth-First-Search (G, v)
color[u] = Black
time = time + 1
f[u] = time

* Comparision of DFS and BFS :-

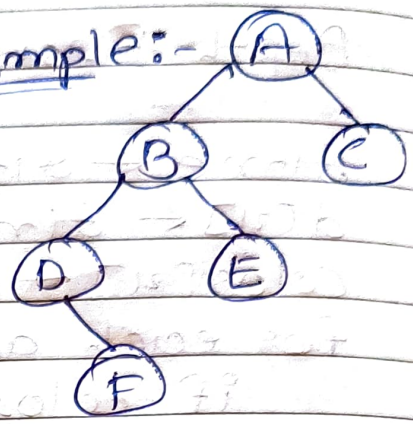
DFS	BFS
1. DFS explores the search as far as possible from the root node.	BFS explores the search level by level as close as possible from the root.
2. DFS is implemented using stack, in LIFO order.	BFS is implemented using queue, in FIFO order.
3. DFS is faster than BFS.	BFS is slower than DFS.
4. DFS needs less memory.	BFS needs more memory.

5. Example:-



Sequence:- ABDFEC

Example:-



Sequence:- ABCDEF

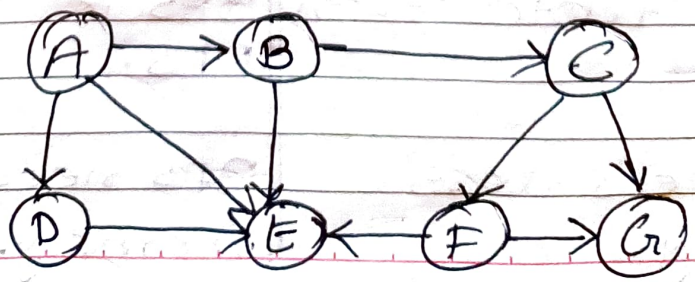
6. Applications of DFS:-
1. Cycle detection
2. Finding articulation point.
3. Topological sorting.

Applications of BFS:-
1. Find shortest path
2. Spanning tree
3. Connectivity checking.

* Topological Sort :-

- Topological sorting is applicable on directed acyclic graphs only.

⇒ Directed Acyclic Graph:- A graph is called a directed acyclic graph if graph is directed graph and it has no directed cycles.



⇒ Topological Sort :-

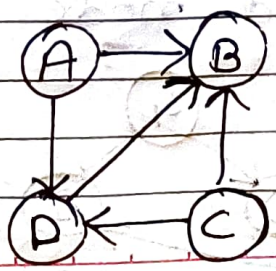
- Topological sorting of directed graph is linear ordering of its vertices.
- In this sorting, for every directed edge from u to v, u comes before v in ordering.
- Two algorithms :-

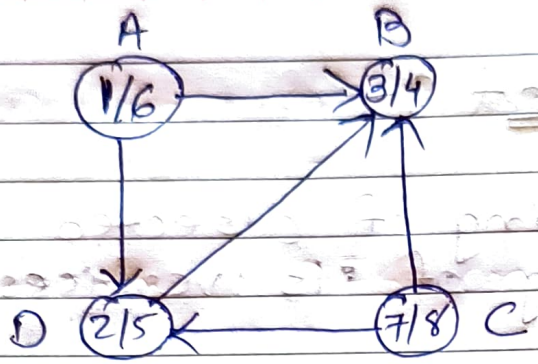
  (1) DFS Based Algorithm
  (2) Source Removal Algorithm

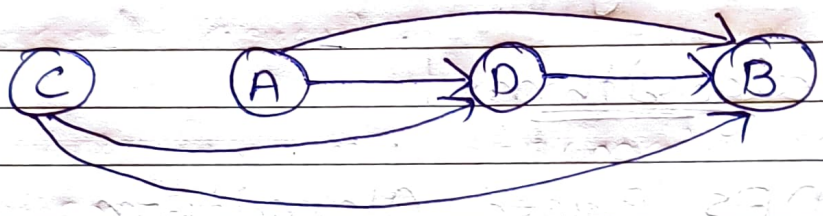
1. DFS Based Algorithm :-

- Call DFS(G) to compute finishing times f[v] for each vertex v.
- As each vertex is finished, insert it onto the front of a linked list
- Return the linked list of vertices.
- Result is just a list of vertices in order of decreasing finish times f[].

⇒ Example :-





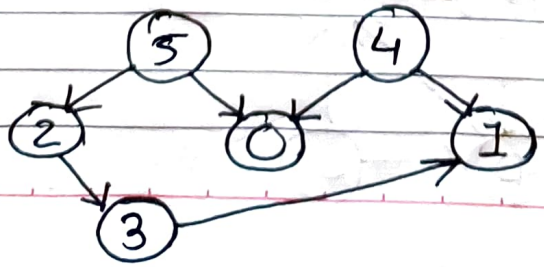


⇒ Applications of topological sorting :-

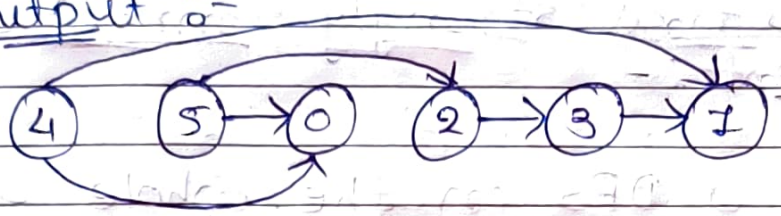
1. Project Management
2. Job Scheduling.
3. Data serialization
4. Ordering of cell evaluation in formulas in spread sheet.

2. Source Removal Algorithm :-

- Identify nodes having in degree '0'.
- Select a node and delete it with its edges then add node to output.

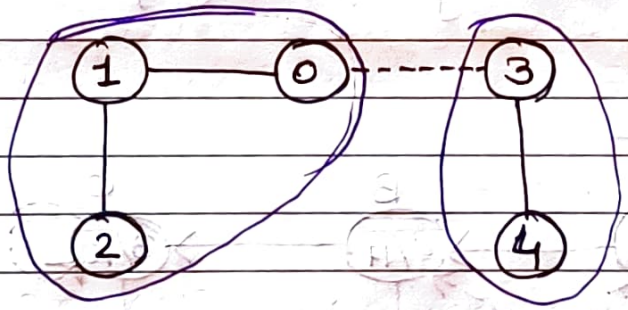


⇒ Output :-



* Connected Components :-

- A connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by paths.



- There are two connected components in the above undirected graph 0 1 2 and 3 4.

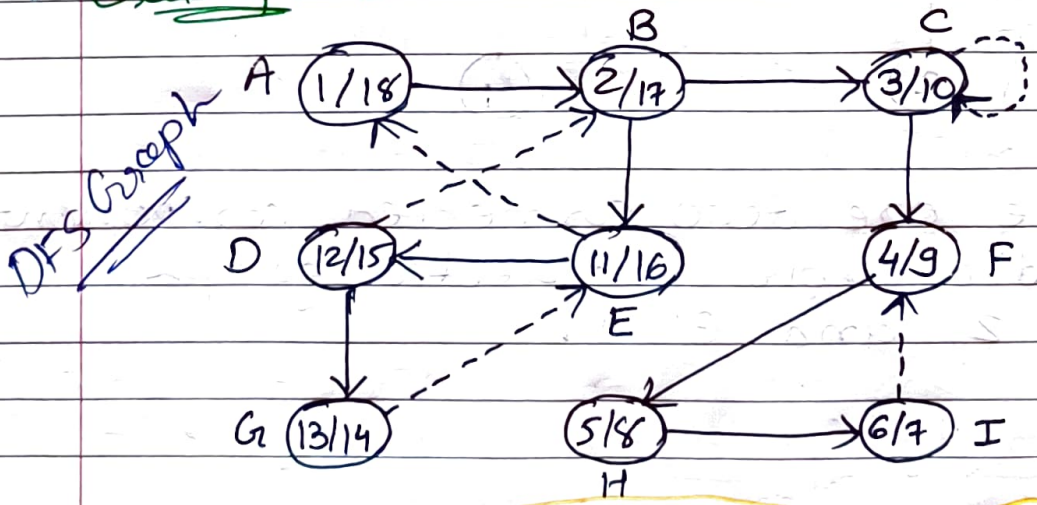
(1) Strongly Connected Components :-

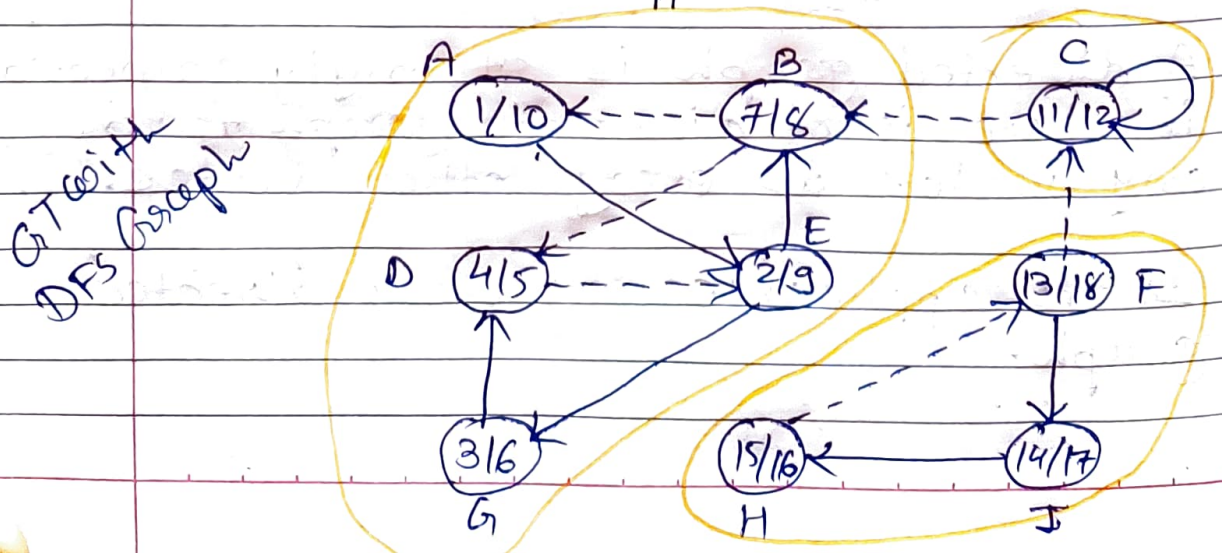
- A directed graph is strongly connected if there is a directed graph from any vertex to every other vertex.
- It is applicable only on a directed graph.

⇒ Steps to find strongly connected components :-

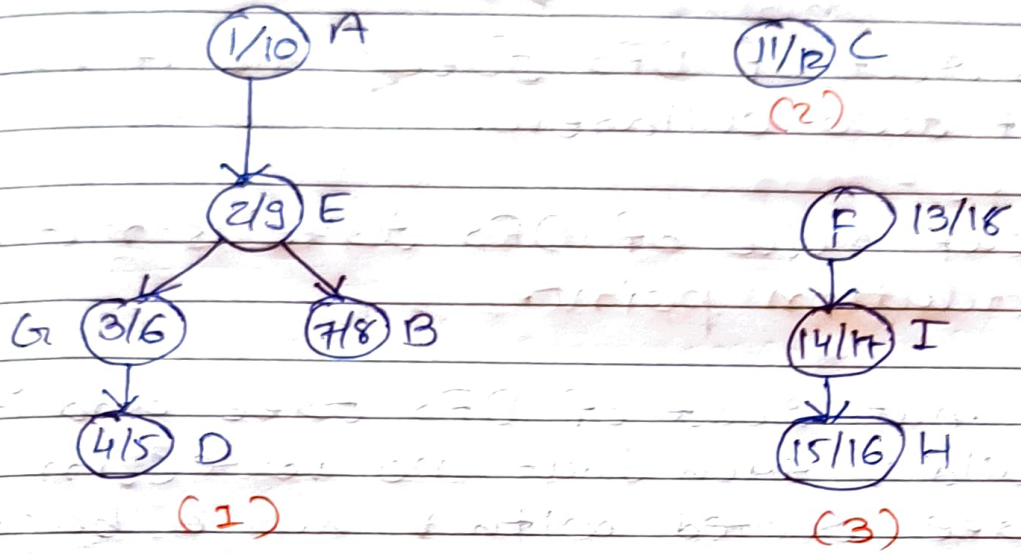
1. Perform a DFS on the whole graph.
2. Find G^T (Reverse the original graph)
3. Perform depth-first search on the reversed graph.
4. Then we finding strongly connected components

➔ Example :-





=> Strongly connected components are :-

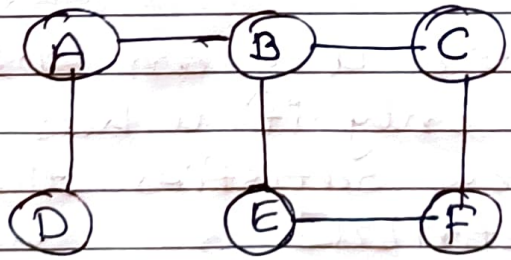


→ (1) { A, E, G, D, B }
(2) { C }
(3) { F, I, H }

(2) Graph Components :-

1 Articulation Point :-

- Articulation point is the vertex $v \in V$ in graph $G = (V, E)$, whose removal disconnects graph G.



- In above example, removal of A or B divides graph in two components, so A & B are articulation points.

⇒ Conditions for articulation point :-

1. u is root of DFS tree and it has at least two children.
2. A leaf node of DFS tree is not an articulation point.
3. u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors of u.

⇒ Formula :-

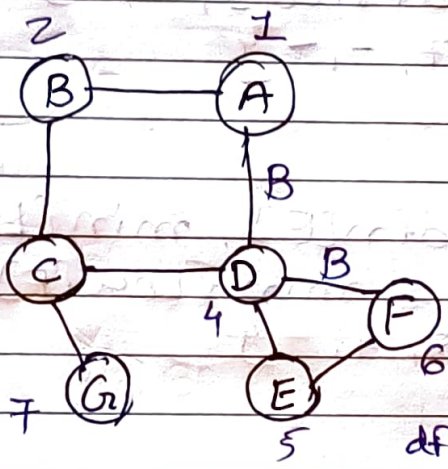
$$Low(u) = \min\{ dfn[u], \min(Low[w]), \min(dfn[w])\}$$

(w is child of u; (u, w is back edge))

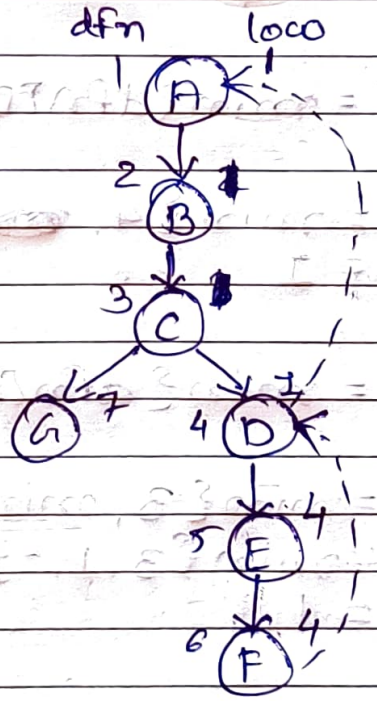
Note :- Root of DFS tree is an articulation point if and only if it has more than one child.

Note :- Any other vertex u is an articulation point if and only if u has some child w which satisfies the condition $Low[w] \geq dfn[u]$

⇒ Example :-



⇒ DFS tree :-



→ Finding loco for every vertex.

$loco[A] = min\{dfn[A], min\{loco[B]\}, -\}$
$= min\{1, loco[B], -\}$
$= 1$

$loco[F] = min\{dfn[F], -, min\{dfn[D]\}\}$
$= min\{6, -, 4\}$
$= 4$

Low[G] = min{dfn[G], -, -}
= min{7, -, -}
= 7

Low[E] = min{dfn[E], min{Low[F]}, -}
= min{5, min{Low[F]}, -}
= min{5, 4, -}
= 4

Low[D] = min{dfn[D], min{Low[E]}, min{dfn[A]}}
= min{4, 4, 1}
= 1

Low[C] = min{dfn[C], min{Low[G], low[D]}, -}
= min{3, min{7, 1}, -}
= min{3, 1, -}
= 1

Low[B] = min{dfn[B], min{Low[C]}, -}
= min{2, 1, -}
= 1

- Now, check conditions for finding articulation point.

→ A is root but it has only one child so it cannot be an articulation point.

→ For B, it should satisfy the condition Low[C] ≥ dfn[B].
here, Low[C] = 1 & dfn[B] = 2
so, B is not articulation point.

→ For C, it should satisfy the condition Low[G] ≥ dfn[C]. as Low[D] ≥ dfn[C].
here, Low[G] = 7 & dfn[C] = 3.
so, C is an articulation point.

→ For D, it should satisfy the condition Low[E] ≥ dfn[D].
here, Low[E] = 4 & dfn[D] = 4.
so, D is an articulation point.

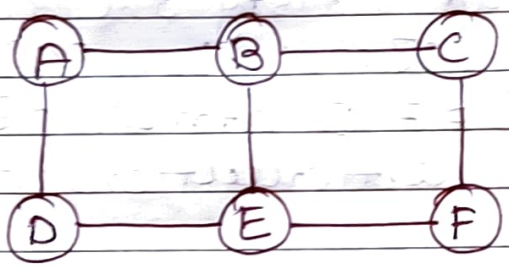
→ For E, it should satisfy the condition Low[F] ≥ dfn[E].
here, Low[F] = 4 & dfn[E] = 5.
so, E is not an articulation point.

→ F & G are leaf node, so they cannot be an articulation point.

⇒ For given graph, there are two articulation points, {C, D}.

2 Bi-Connected Component :-

→ Graph G is called bi-connected graph if it does not have any articulation point.



Given

- In ↓ Graph G, has no articulation point, so removal of any vertex does not factor graph in two parts. so, it is bi-connected graph.

- To disconnect a bi-connected graph, we must remove at least two vertices, if we remove vertex B and E from graph, it becomes disconnected.