Traversing of a Graph

Traversing of a graph means visiting all vertices in G that are reachable from v. (i.e, all vertices that are connected to v).

There are two ways for Traversing of a Graph

1) Breadth - first Search (BFS)
2) Depth - first search (DFS).

1. Breadth - First Search :-

In breadth first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are visited. unvisited vertices to these newly visited vertices are then visited and so on.

We use queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

Algorithm :-

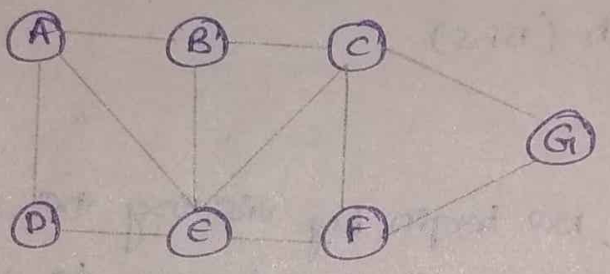
1. Define a queue (size is total number of vertices in the graph).
2. Select any vertex as a starting point for traversal (i.e., v).
3. visit the selected vertex (v) and insert into queue.
4. visit all the adjacent vertices of vertex (v) which is at front of the queue which is not visited and insert them into queue.
5. When there is no new vertex to be visit from the vertex (v) at front of the queue then delete the vertex from the queue.

6) Repeat step 4 & step 5 until queue becomes Empty

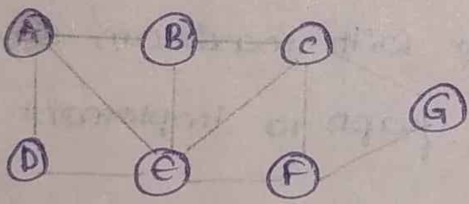
7) When queue becomes Empty, then produce a final tree by removing unused edges from the graph.

Example

consider the following Example to perform BFS traversal.



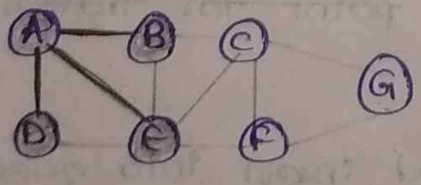
step 1 :- select the vertex 'A' as starting point (visit A) and insert A into the queue.



Queue.

A						

Step 2 :- visit all adjacent vertices of A which are not visited (D, E, B) and insert newly visited vertex into the queue and delete A from the queue.

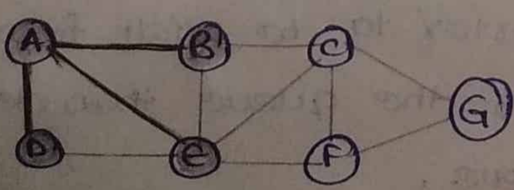


queue

	D	E	B			

Step 3 :- visit all vertices of D which are not visited (there is no vertex of D).

SO, Delete D from the queue.

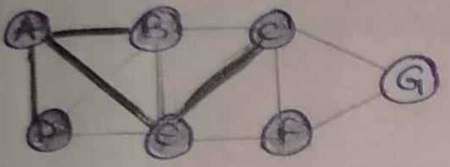


Queue

		E	B			

Step 4 :- Visit all the adjacent vertices of E (unvisited) (C, F).

Insert newly visited vertices into the queue and delete E from the queue.

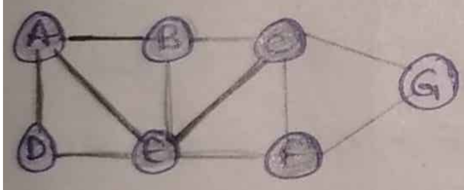


Queue

			B	C	F		

Step 5 :- Visit all the adjacent vertices of ~~wh~~ B (which are not visited)

But, there are no vertices. so, Delete B from the queue.



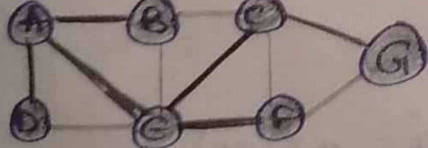
Queue

				C	F	

Step 6 :- Visit all the adjacent vertices of C which are not visited (G).

Insert newly visited vertex into the queue and delete 'C' from the queue.

Queue

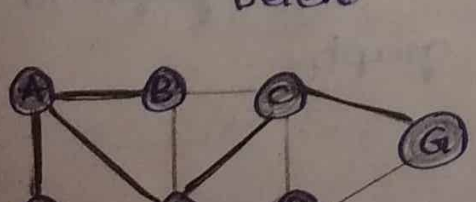


					F	G

Step 7 :- Visit all the adjacent vertices of F which are not visited yet (there are no vertices).

Delete F from the queue.

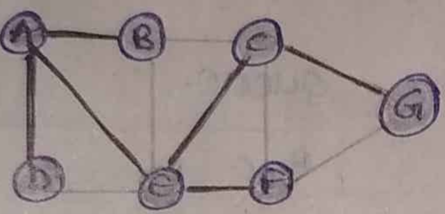
Queue

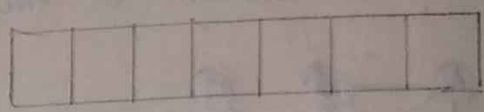


						G

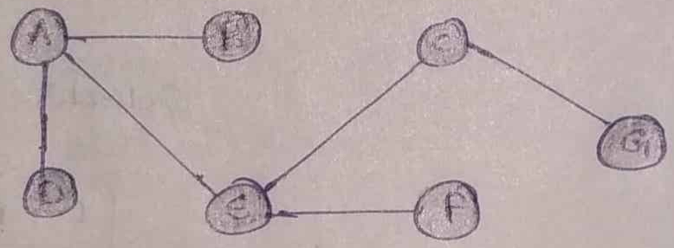
Step 8 :- visit all adjacent vertices of G which are not visited (there is no vertex).
Delete G from the Queue.

Queue





→ Queue is Empty now, so stop the BFS process.
→ Final result of BFS is a spanning tree as shown in fig below



ii) Depth - first search :-
→ In DFS begin the search by visiting the start vertex (v).
→ If v has an unvisited neighbour, traverse it recursi-vely.
→ otherwise backtrack.

2) Time complexity → Adjacent list : O(|E|)
→ Adjacency matrix : O(N|).

3) we use stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

Algorithm :-

1) Define a stack of size total no. of vertices in the graph

2) Select any vertex as starting point for traversal visit that vertex (v) and push it onto the stack.

3) visit any one of the adjacent vertex of the vertex (v) which is at top of the stack which is not visited and push it on to the stack.

4) Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

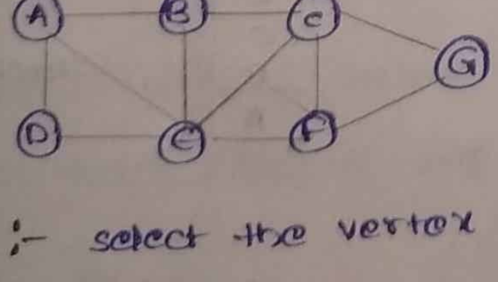
5) when there is no new vertex to be visit then use back-tracking and pop one vertex from the stack.

6) Repeat step 3, 4 and 5 until stack becomes Empty.

7) when stack becomes Empty then produce final spanning tree by removing unused edges from the graph.

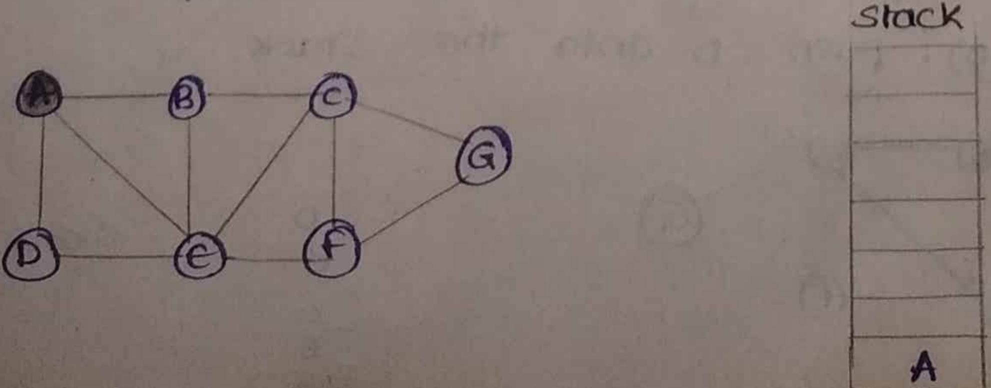
Example :-

Consider the following Example graph to perform DFS Traversal.



Step1 :- select the vertex A as starting point (visit A)

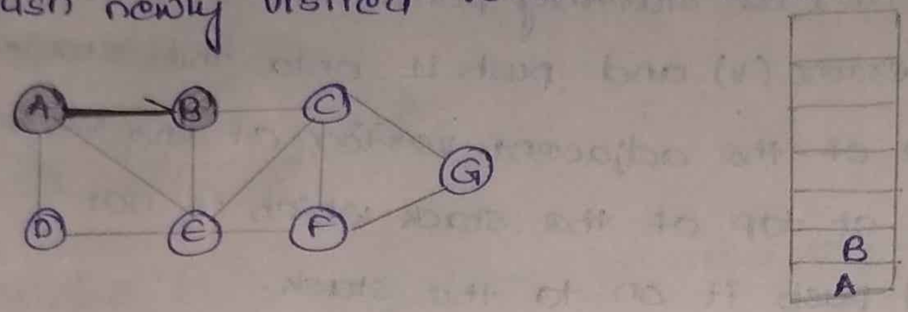
-push A onto the stack.



Step 2 :-

→ visit any adjacent vertex of A which are not yet visited.

→ push newly visited vertex 'B' onto the stack.

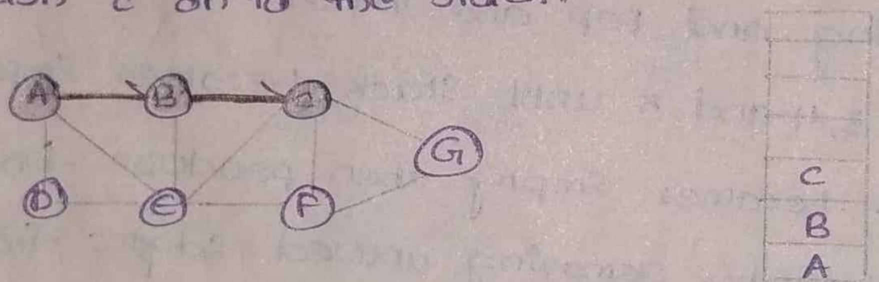


stack

Step 3 :-

→ visit any adjacent vertex of B which are not yet visited. (c)

→ push c on to the stack.

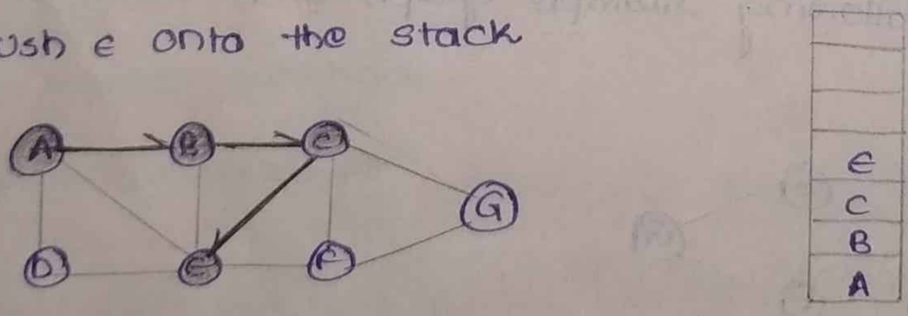


stack

Step 4 :-

→ visit any adjacent vertex of c which is not yet visited (E)

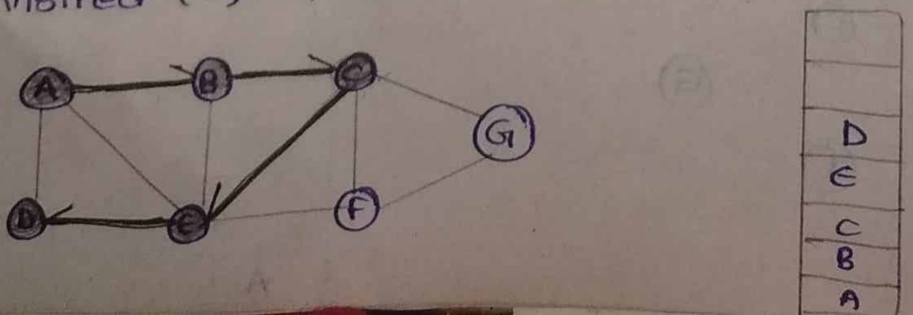
→ push E onto the stack



stack

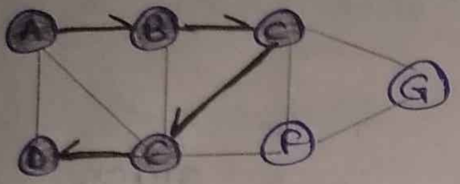
Step 5 :-

→ visit any adjacent vertex of E which is not visited (D). push D onto the stack.



stack

step 6 :- there is no new vertex to be visited from D. so use back track and pop D from the stack.

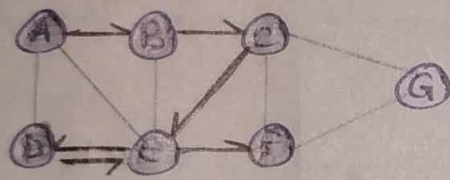


E
C
B
A

Stack

step 7 :-

visit any adjacent vertex of E which is not visited (F) and push F onto the stack.

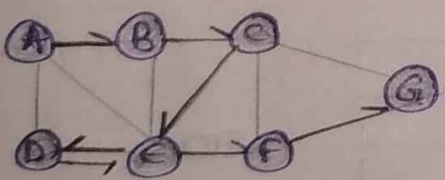


F
E
C
B
A

Stack

step 8 :-

visit any adjacent vertex of F which is not visited (G) and push G onto the stack

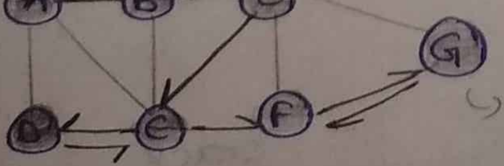


G
F
E
C
B
A

Stack

step 9 :-

there is no new vertex to be visited from G. so, use back track. Delete (POP) G from stack.



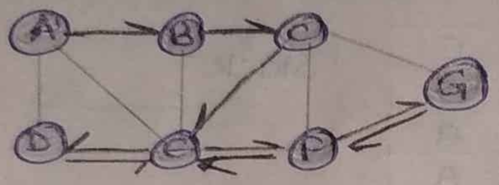
poped element

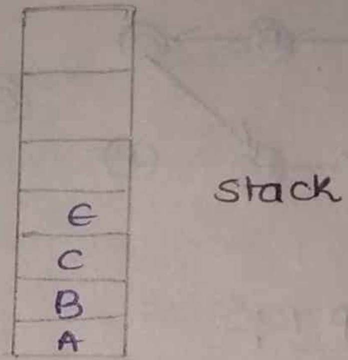
F
E
C
B
A

stack

step 10 :-

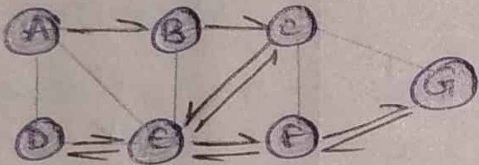
there is no new vertex to be visited from F. so, we back track. pop F from the stack.

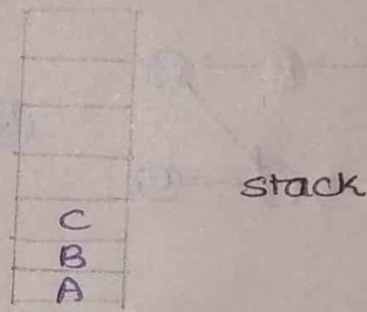




step 11 :-

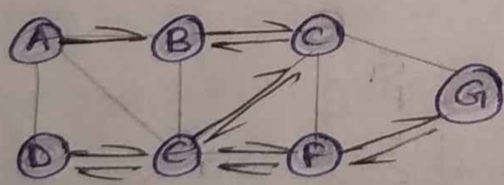
there is no new vertex to be visited from E. so, we back track. pop E from the stack.

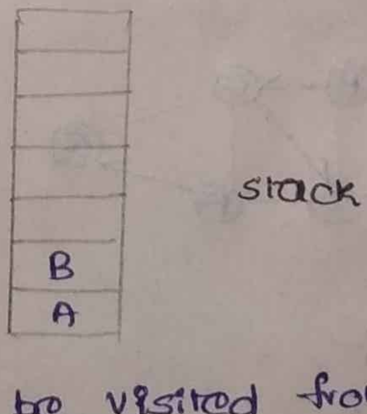




step 12 :-

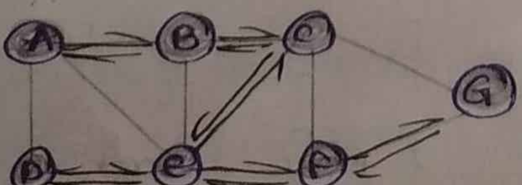
there is no new vertex to be visited from C. so, use back track. pop c from the stack.

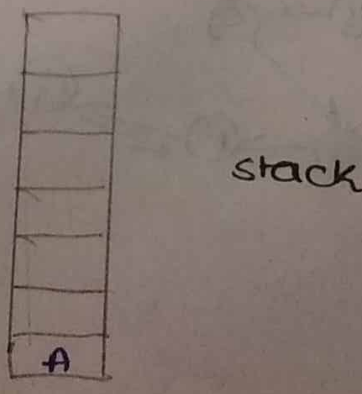




step 13 :-

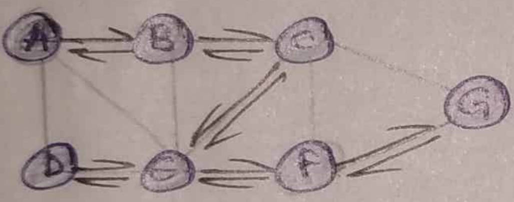
there is no new vertex to be visited from B. so, use back track. pop B from the stack.

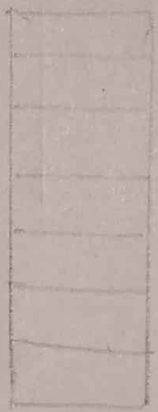




step 14 :-

there is no new vertex to be visited from A. so, use back track and pop A fom the stack.

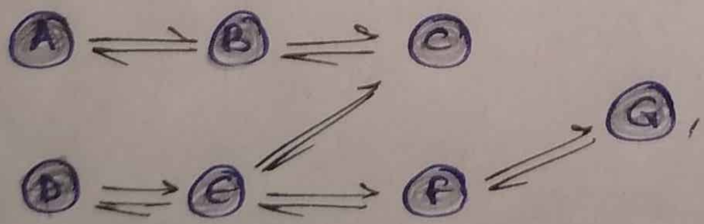




stack

→ stack became Empty, stop DFS traversal

→ final result of DFS traversal is following spanning tree.



Differences between BFS and DFS

BFS	DFS
1) BFS stands for Breadth first search	DFS stands for Depth first search.
2) BFS uses queue data structure to find the shortest path	DFS uses the stack data structure to find the shortest path.
3) BFS is better when target is closer to the source	DFS is better when target is far from the source.
4) As BFS considers all neighbours, so it is not suitable for decision tree based in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to tranverse further to augment the decision. If we reach the conclusion we won.
5) BFS is slower than DFS	DFS is faster than BFS.
6) the time complexity of BFS = $O(V+E)$, where V is vertices and E is Edges	the time complexity of DFS = $O(V+E)$, where V is vertices and E is Edges.

Connected components :-

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

A graph G is said to be disconnected, if doesn't contain atleast two connected vertices.

* connected components are of two types. they are
  1. Directed graph
  2. undirected graph.

Algorithm for undirected graph :-

1) for each vertex in v in G.v, Assign -1 (or) Assign a flag value -1 to whole the vertices of the graph.

2) Do following for every vertex 'v'
   a) If v is not visited before
   b) print newline character.

3) call the DFS (v) function, pass v to it
   i) Mark 'v' as visited
   ii) print 'v'
   iii) Do following for every adjacent 'u' of v
   If 'u' is not visited, then recursively call DFS (u).

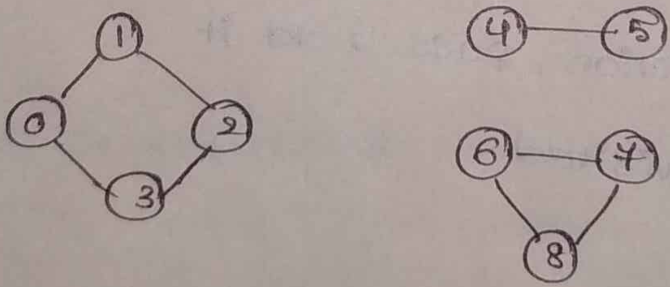
Program using DFS traversal :-

```
Connect Component (G, N) {
    for each vertex in v in G.v
        do flag [v] ← -1
```

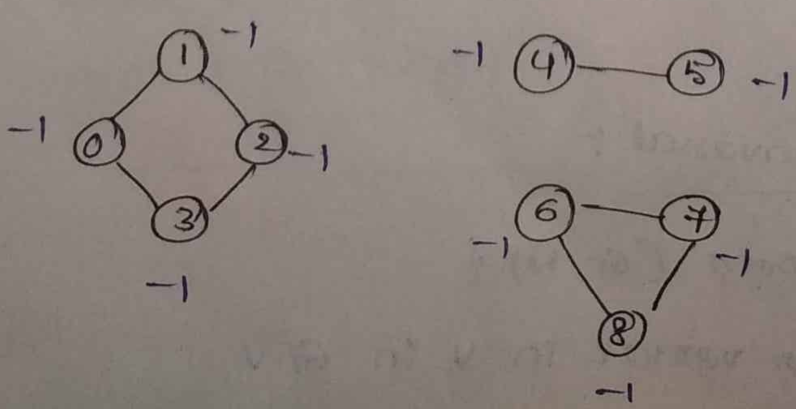
```
for each vertex in G.V
        do if ( Flag[v] = = -1)
            DFS (v, flag)
            count ++
    return count
}
DFS (v, flag) {
    print v
    Flag[v] ← 1
    for each adjacent node u of v
        do if ( Flag[u] = = -1)
            DFS (u, flag)
}
```

* Example :-

consider the following example.

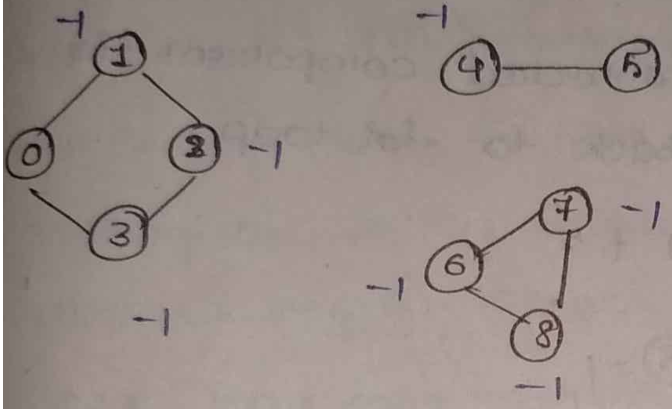


Step 1 :- Assign a flag value -1 for each vertex belong to N (9) 0 to 8.



ep2 :- starting from '0' vertex · Here the vertex v
0 is -1, then we enter into DFS function.

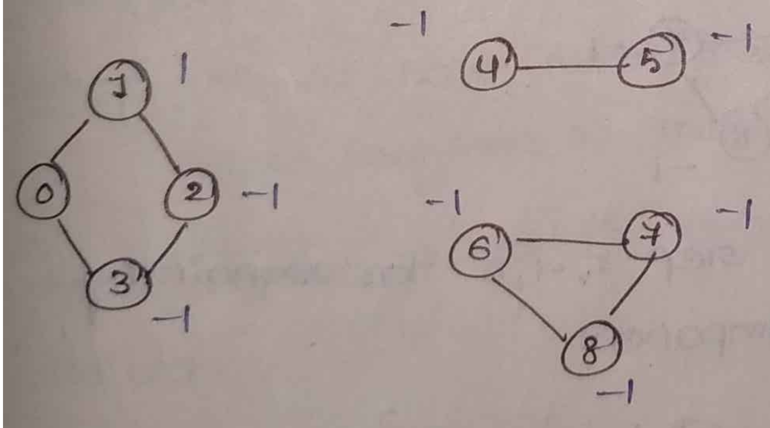
p3 :- After entering into dfs function, print all the vertex 0 and set the 1 to the '0' vertex.

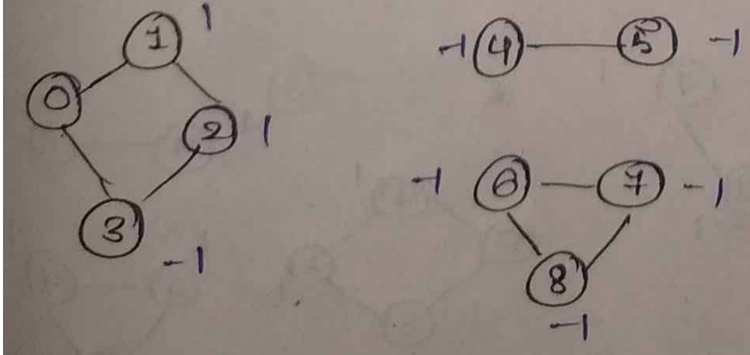


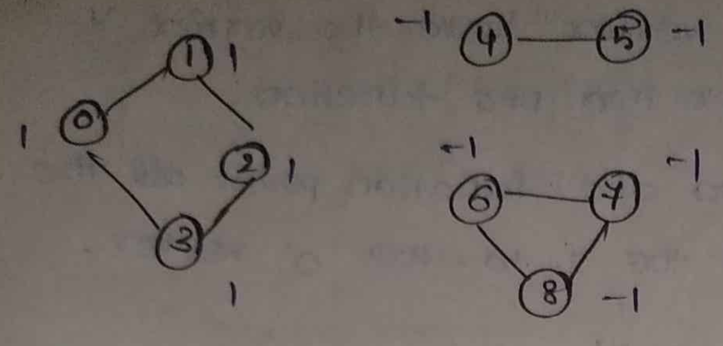
p4 :- For each adjacent vertex u of v is not visited suppose i.e vertex 1) DFS is called recursively passing u and flag value (here value is 1)
until all the vertices are

Step 5 :- Repeat step 3 & 4

visited.

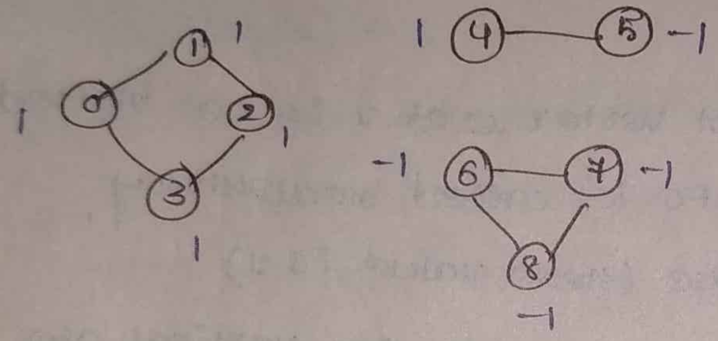


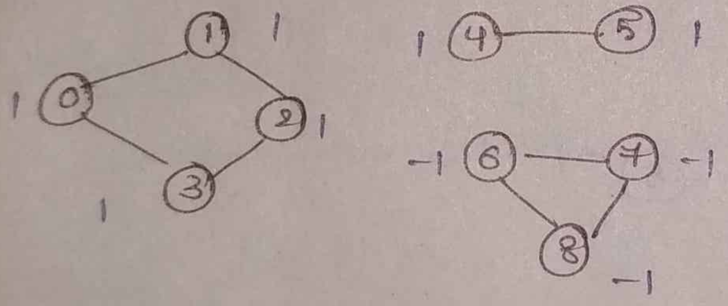




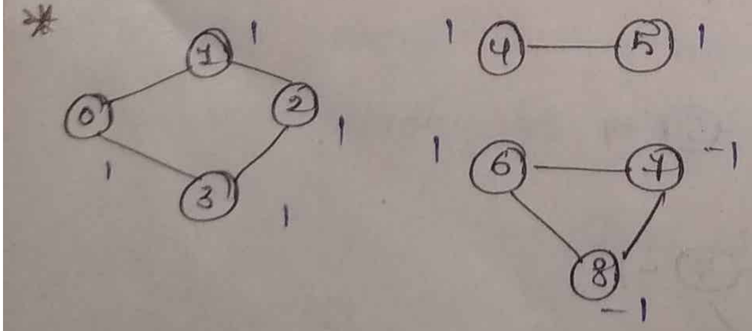
Here traversing of one connected component is completed, then we go back to for loop.

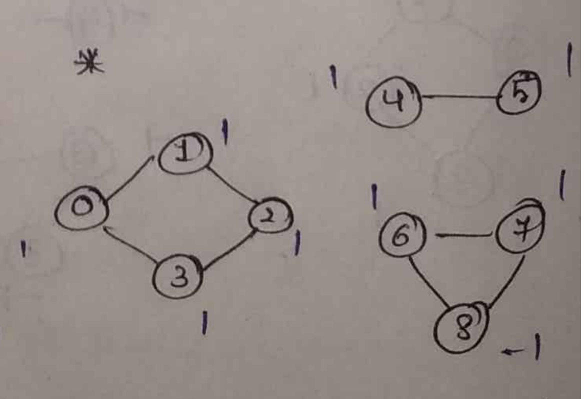
Step 6 :- Repeat step 3, 4 & 5

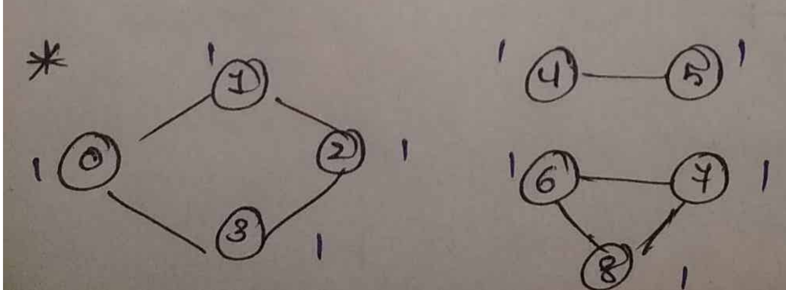




Step 7 :- Again repeat step 3, 4, 5 for remaining connected component.







Topological sort

Topological sort is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex u to vertex v then u comes before v in the ordering.

→ It is important to note that

1) Topological sorting is possible if the graph is directed Acyclic Graph.

2) There may exist multiple different topological orderings for a given directed acyclic graph.

Algorithm :-

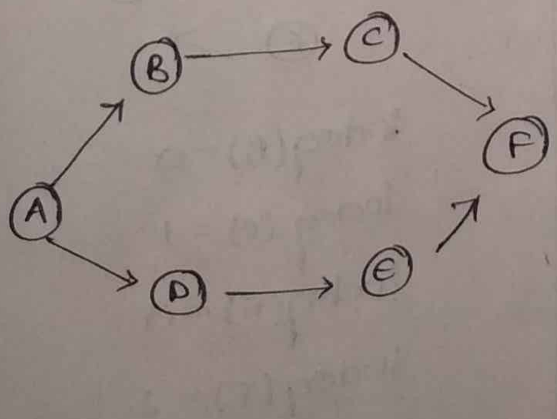
1) Firstly we update indegree of ~~eve~~ each vertex

2) find the least one and remove it and print that one (ans).

3) **Note :-** If we have more than one leasts, we can take any one of them.

4) Again repeat ① and ② steps until all nodes are visited.

Example :-



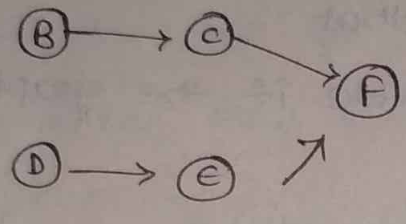
Step1 :- updating indegree of each vertex

indegree (A) = 0 indegree (D) = 1

indegree (B) = 1 indegree (E) = 1

indegree (C) = 1 indegree (F) = 2

Least one is A, print it and delete it



Step2 :- After deleting vertex A, Again updating indegrees

indeg(B) = 0 indeg(E) = 1

indeg(C) = 1 indeg(F) = 2

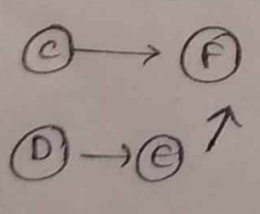
indeg(D) = 0

Here we have two least, we can take either of them, so two cases.

case 1 :- A → B (or) case 2 :- A → D

Here, first delete B

Again updating indegrees.



indeg(C) = 0

indeg(F) = 2

indeg(D) = 0

indeg(E) = 1

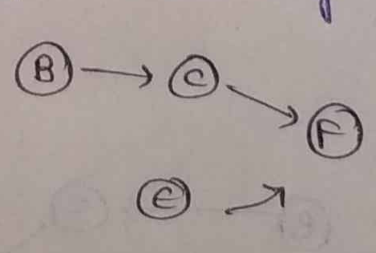
Here, also 2 leasts

A → B → C (or) ①

A → B → D ②

Here, first delete D

Again updating indegrees



indeg(B) = 0

indeg(C) = 1

indeg(E) = 0

indeg(F) = 2

Again 2 leasts.

A → D → B ⓐ

(or)

A → D → E ⓑ

for 1 :-

we update indegrees

indeg(D) = 0 ; indeg(E) = 1 ; indeg(F) = 1

Here least is D, output it and delete it

A → B → C → D

Again updating indegrees

indeg(E) = 0 ; indeg(F) = 1

Here least is E, output it and delete it

A → B → C → D → E

Again updating indegree of F i.e 0

only one so output it and End

1 A → B → C → D → E → F

For 2 :-

we update indegrees

indeg(C) = 0 ; indeg(F) = 2 ; indeg(E) = 0.

Here also we have two cases :

i) Deleting C another ii) Deleting E

case 1 :- A → B → D → C

case 2 :- A → B → D → E

updating indegrees for case (i)

indeg(F) = 1 ; indeg(E) = 0

least is E ; output it and delete it.

Now least is F then output it and End

2 A → B → D → C → E → F

updating indegrees for case ②

indeg(C) = 0 ; indeg(F) = 1

least is C ; output it and then least is F , output it and End

③ A → B → D → E → C → F

from a :-

updating indegrees

indeg(C) = 0 ; indeg(E) = 0 ; indeg(F) = 2

here also, there are two cases.

i) Deleting C another ii) Deleting E

updating indegrees for case i

indeg(E) = 0 ; indeg(F) = 1

least is E ; output it then F

④ A → D → B → C → E → F

updating indegrees for case ii

indeg(C) = 0 ; indeg(F) = 1

least is C ; output it and then F

⑤ A → D → B → E → C → F

from b :-

updating indegrees

indeg(B) = 0 ; indeg(C) = 1 ; indeg(F) = 2

least is B, output it and delete it.

→ updating indegrees

indeg(C) = 0 ; indeg(F) = 1

least is c, output it and delete it.

→ updating indegrees

indeg(F) = 0

output it and delete it

(6) A → B → E → B → C → F

Hence,

we get 6 topological orderings

1. A B C D E F

2. A B D C E F

3. A B D E C F

4. A D B C E F

5. A D B E C F

6. A D E B C F