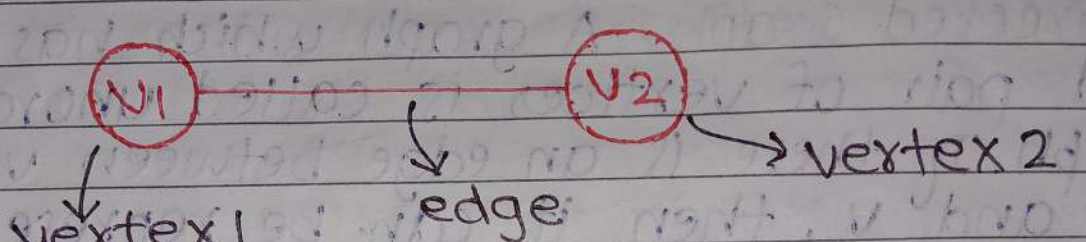
Graph

A Graph consist of two types of elements :- vertices and edges. Each edge has two endpoints, which belong to the vertex set. Edge connects two vertices.

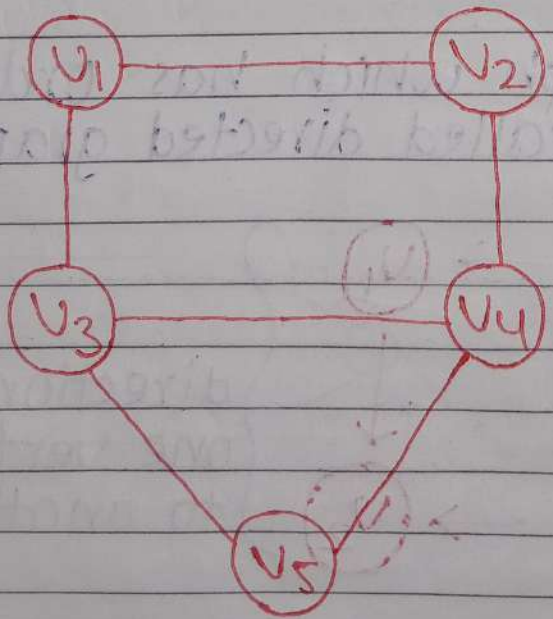


Graph represents a pair of set :-

$G = (V, E)$

$V = \{V_1, V_2, V_3, V_4, V_5\}$

$E = \{(V_1, V_2), (V_1, V_3), (V_3, V_4), (V_2, V_4), (V_3, V_5), (V_4, V_5)\}$

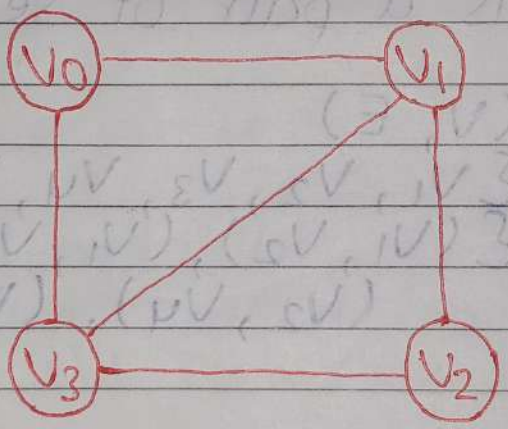


A graph is of 2 types :-

Directed Graph

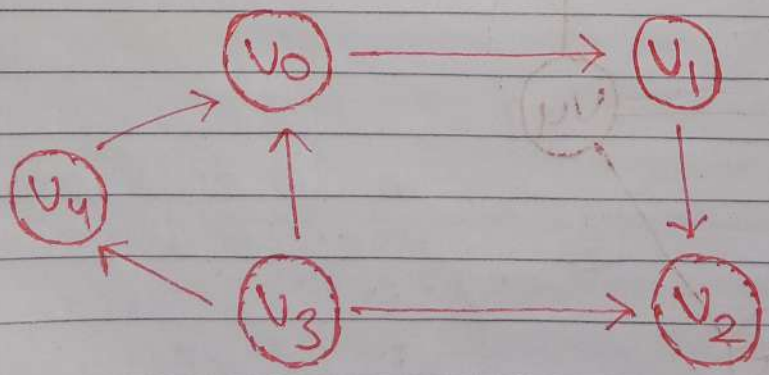
Undirected Graph

* Undirected Graph : A graph which has unorde-red pair of vertices is called unordered graph. If there is an edge between vertices u and v, then it can be represented as either (u, v) or (v, u).

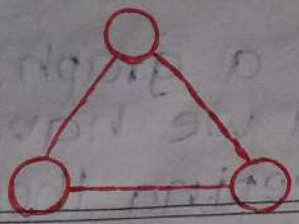


no direction from one vertex to another.

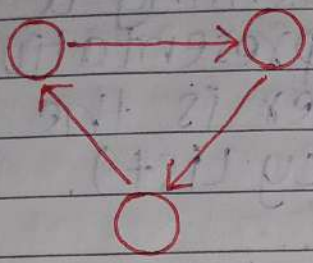
* directed Graph : A graph which has ordered pair of vertices is called directed graph.



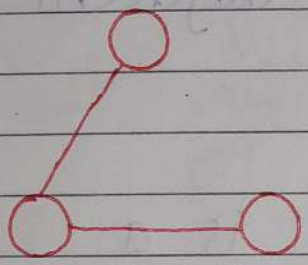
directions from one vertex to another.



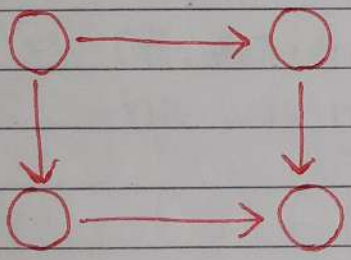
→ cyclic undirected graph



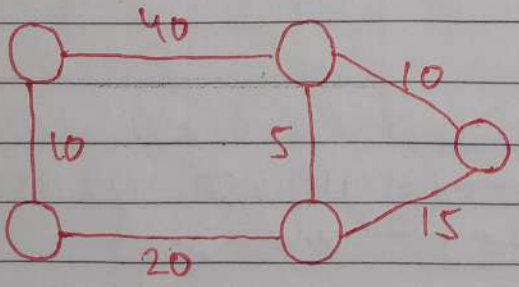
→ cyclic directed graph



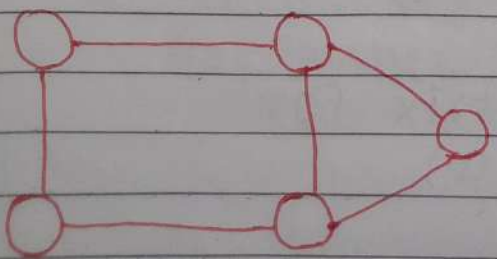
→ acyclic undirected graph



→ acyclic directed graph



→ weighted graph

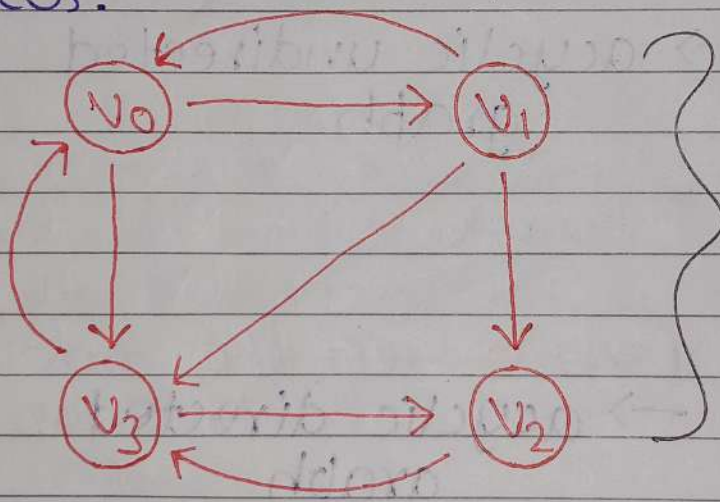


→ unweighted graph

* Representation of a Graph :-

We have mainly two parts in a graph, i.e., vertices and edges, and we have to design a data structure keeping these parts in mind.
There are two ways of representing a graph, one is Sequential Representation (Adjacency Matrix) and other is the Linked Representation (Adjacency List).

(1) Adjacency Matrix = It is the matrix that maintains the information of adjacent vertices.

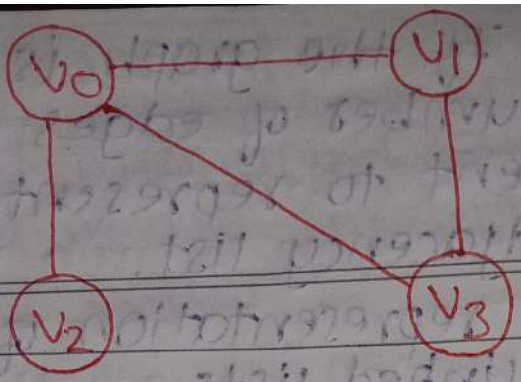


this is a directed graph.

a) Directed Graph

	V_0	V_1	V_2	V_3
V_0	0	1	0	1
V_1	1	0	1	1
V_2	0	0	0	1
V_3	1	0	1	0

b) Adjacency Matrix for Graph (a)

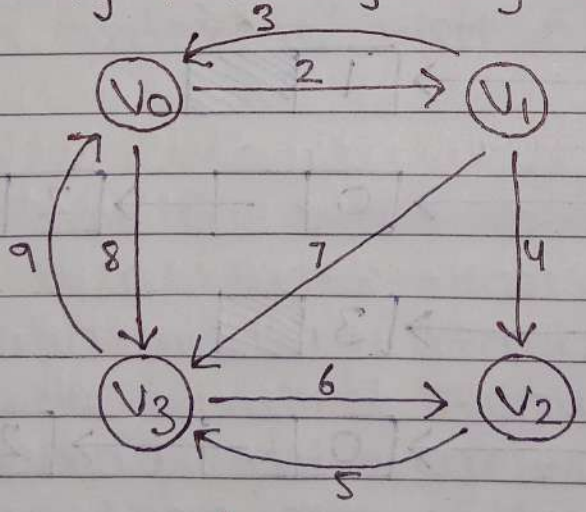


c) Undirected Graph

	V_0	V_1	V_2	V_3
V_0	0	1	1	1
V_1	1	0	0	1
V_2	1	0	0	0
V_3	1	1	0	0

d) Adjacency Matrix for Graph (c)

Ques Given the directed weighted graph, create the weighted adjancy matrix for it.



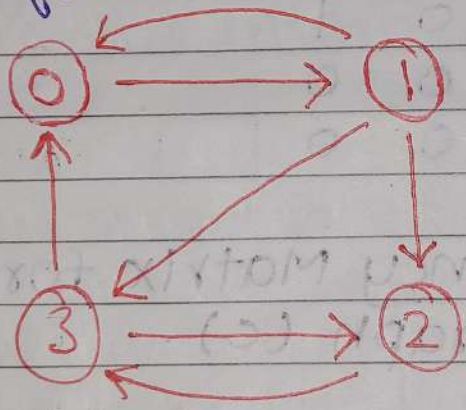
weighted Adjacency Matrix

	V_0	V_1	V_2	V_3
V_0	0	2	0	8
V_1	3	0	4	7
V_2	0	0	0	5
V_3	9	0	6	0

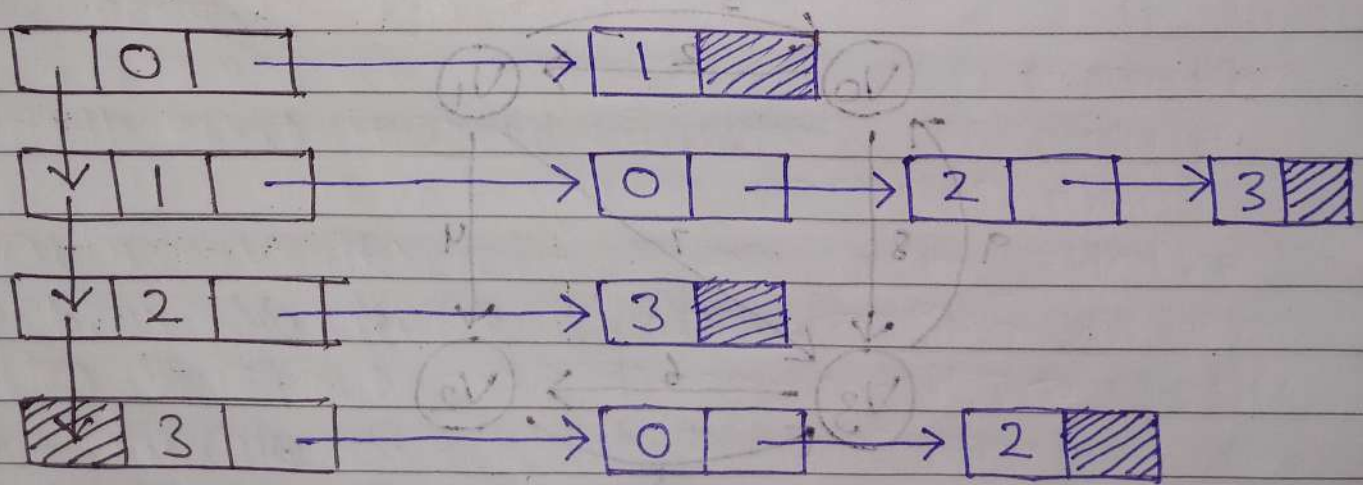
(2) Adjacency List : If the graph is not dense, i.e., the number of edges is less, then it is efficient to represent the graph through adjacency list.

In adjacency list representation of graph, we maintain 2 linked lists.

The first linked list is the vertex list that keeps track of all the vertices in the graph and second linked list is the edge list that maintains a list of adjacent vertices for each vertex.



a) Directed Graph



b) Adjacency List for graph (a)

Graph Traversal

Traversal in Graph is different from traversal in tree or list because of the following reason :-

i) There is no first vertex or root vertex in a graph, hence the traversal can start from any vertex. we can choose any vertex as the starting vertex. A traversal algorithm will produce different sequences for different starting vertices.

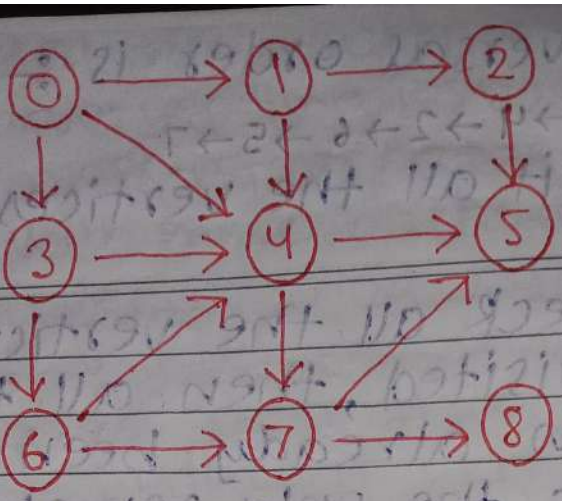
ii) In tree or list, when we start traversing from the first vertex, all the elements are visited but in graph only those vertices will be visited which are reachable from the starting vertex.
So, if we want to visit all the vertices of the graph, we have to select another starting vertex from the remaining vertices in order to visit all the vertices left.

iii) In tree or list while traversing, we never encounter a vertex more than once while in graph we may reach a vertex more than once. This is because in graph a vertex may have cycles and there may be more than one path to reach a vertex. So, to ensure that each vertex is visited only once, we have to keep the status of each vertex whether it has been visited or not.

iv) In tree or list we have uniaue traversals. For example, if we are traversing a binary tree in inOrder, there can be only one seauence in which vertices can be visited. This is because there is no natural order among the successor of vertex and thus the successor may be visited in different orders producing different seauences. The order in which successors are visited may depend on the implementation.

Like Binary Trees, in graph also there can be many methods by which a graph can be traver-sed, but two of them are stan-dard and are known as Breadth First Search and Depth First Search.

* Breadth First Search = In this techniaue, first we visit the starting vertex and then visit all the vertices adjacent to the starting vertex. After this we pick these adjacent vertices one by one and visit their adjacent vertices and this process goes on. This traversal is eauivalent to level order traversal of trees.



Here, we start from vertex 0 as the starting vertex.
First we will visit the vertex 0. Then we will visit all vertices adjacent to vertex 0, i.e., 1, 4, 3. we can visit these vertices in any order.
Suppose we visit the vertices in order 1, 3, 4. So, our traversal order is :-

$0 \rightarrow 1 \rightarrow 3 \rightarrow 4$

Now, after choosing the order, we again start visiting the vertices in the same order, we choose previous vertices. Means, first we visit all the vertices adjacent to 1, then visit all the vertices adjacent to 3 and then all the vertices adjacent to 4.

First we will visit vertices adjacent to 1, i.e., 2, we will not visit 4 because 4 is already visited, so we select only 2.
Then we visit vertices adjacent to 3, i.e., 6, we will not visit 4 because 4 is already visited, so we select only 6.
Then we visit vertices adjacent to 4, i.e, 5 and 7.

So, our new traversal order is :

$0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 7$

Now, we will visit all the vertices adjacent to 2, 6, 5, 7.

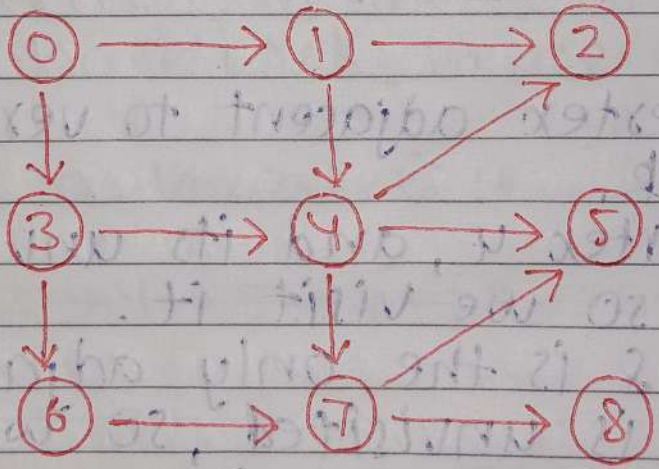
So when we check all the vertices of 2 has already been visited, then all the vertices of 6 has already been visited, 5 has no vertex, the only remaining vertex to be visited is 8, which is adjacent to 7. Finally, we visit the 8 vertex.

So, our final traversal order is :-

$0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 7 \rightarrow 8$

* Depth First Search :- In this technique, we travel along a path in the graph & when a dead end comes we backtrack. This technique is named so because search proceeds deeper in the graph, i.e., we traverse along the graph path as deep as we can.
First the starting vertex will be visited & then we will pick up any path that starts from the starting vertex & visit all the vertices in this path till we reach a dead end.
Dead end means, that we reach a vertex which does not have any adjacent vertex or all of its adjacent vertices have been visited.
After reaching the dead end we will backtrack along the path that we have

visited till now. Suppose the path that we've traversed is V1 → V2 → V3 → V4 → V5. After traversing V5 we reach a dead end. Now we will move backwards till we reach a vertex that has any unvisited adjacent vertex. we move back and reach V4 but see that it has no unvisited adjacent vertices so we will reach vertex V3. Now if V3 has an unvisited vertex adjacent to it we will pick up a path that starts from V3 & visit it until we reach a dead end. Then again we will backtrack. This process finishes when we reach the starting vertex and there are no vertices adjacent to it which have to be visited.



First, we will visit vertex 0. Vertices adjacent to vertex 0 is 1 and 3. Suppose, we visti vertex 1. Now, we look at the adjacent vertices of 1. Suppose, we visit 2.
Till now, our traversal is :-

0 → 1 → 2

There is no vertex adjacent to vertex 2. It means we have reached the dead end of the path or the dead end from where we can't go forward. So, we will move backward.
we reach vertex 1 and see if there is any vertex adjacent to it, which is not visited yet.
vertex 4 is such a vertex and therefore we visit it.
Now, vertices 5 and 7 are adjacent to 4, and they are unvisited & from these we choose to visit vertex 5.
Till now our traversal is :-

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$$

There is no vertex adjacent to vertex 5 so we will backtrack.
we reach vertex 4, and its unvisited adjacent vertex is 7, so we visit it.
Now, vertex 8 is the only adjacent vertex to 7 which is unvisited, so we visit it.
Till now our traversal is :-

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8$$

vertex 8 has no unvisited vertex so we backtrack & reach vertex 7.
Now, vertex 7 also has no unvisited adjacent vertex so we backtrack & reach vertex 4. vertex 4 also has no adjacent vertex so we backtrack & reach vertex 1.
vertex 1 also has no adjacent vertex so we

backtrack and reach vertex 0.
vertex 3 is adjacent to vertex 0 which is unvisited. so, we visit vertex 3.
vertex 6 is adjacent to vertex 3 & is unvisited. so, we visit vertex 6.
Till now, our traversal is :-

$$0 \to 1 \to 2 \to 4 \to 5 \to 7 \to 8 \to 3 \to 6$$

Now, vertex 6 has no adjacent unvisited vertex left so we backtrack and reach vertex 3.
vertex 3 also has no unvisited vertex left. so we backtrack and reach vertex 0.
vertex 0 has no adjacent unvisited vertex left, and vertex 0 is the start vertex so we can't backtrack, hence our traversal finishes.
The final traversal is :-

$$0 \to 1 \to 2 \to 4 \to 5 \to 7 \to 8 \to 3 \to 6$$

Breadth First search (BFS), implements using Queue.
Depth First search (DFS), implements using stack.

Implementation of Breadth First search using Queue.

There are basically 3 states of any vertex - Initial, waiting & visited.

At the start of the algorithm, all vertices will be in initial state, when a vertex will be inserted in the Queue its state will change from initial to waiting. when a vertex will be deleted from Queue & visited, its state will change from waiting to visited

Algorithm of BFS using Queue :-

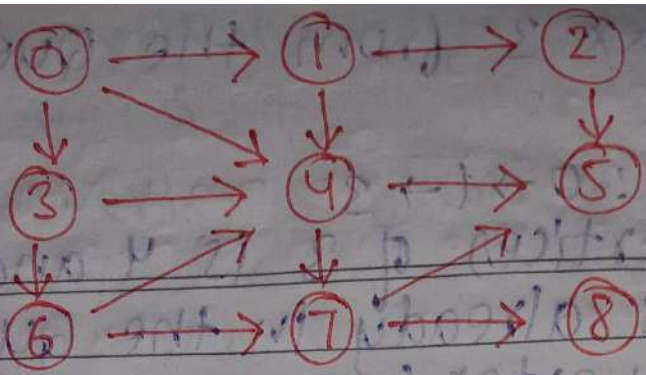
Initially Queue is empty and all the vertices are in initial state.

i) Insert the starting vertex into the Queue, change its state to waiting.

ii) Delete front element from the Queue and visit it, change its state to visited.

iii) Look for the adjacent vertices of the deleted element & from these inserted only those elements into the queue which are in the initial state. change the state of all these inser -ted vertices from initial to waiting.

iv) Repeat step 2 & 3, until the Queue is empty.



Take vertex 0 as the starting vertex for traversal.

i) Initial vertex is 0. So, insert vertex 0 in the Queue.

Queue : 0

ii) Delete vertex 0 from the Queue and visit it.

Traversal : 0

Insert the adjacent vertices of vertex 0 into the Queue, i.e., 1, 3, 4.

Queue : 1, 3, 4

iii) Delete vertex 1 from the Queue & visit it

Traversal : 0 → 1

Adjacent vertices of 1 is 2 & 4. vertex 4 is already in the Queue. So, we insert 2 in the Queue.

Queue : 3, 4, 2

iv) Delete vertex 3 from the Queue & visit it.

Traversal: 0 → 1 → 3

Adjacent vertices of 3 is 4 and 6. vertex 4 is already in the Queue. So, we insert vertex 6.

Queue: 4, 2, 6

v) Delete vertex 4 from the Queue & visit.

Traversal: 0 → 1 → 3 → 4.

Adjacent vertices of 4 is 5 & 7. Insert both vertices in Queue.

Queue: 2, 6, 5, 7

vi) Delete vertex 2 from the Queue & visit it.

Traversal: 0 → 1 → 3 → 4 → 2

Adjacent vertex of 2 is 5 & 5 is already in the Queue.

Queue: 6, 5, 7

vii) Delete vertex 6 from Queue & visit it.

Traversal: 0 → 1 → 3 → 4 → 2 → 6.

Adjacent vertex of 6 is 4 & 7. 4 is already visited & 7 is in the Queue.

So, nothing will insert in the Queue.

Queue: 5, 7

viii) Delete vertex 5 from Queue & visit it.

Traversal: 0→1→3→4→2→6→5

vertex 5 has no adjacent vertex.

Queue: 7

ix) Delete 7 from Queue & visit it.

Traversal: 0→1→3→4→2→6→5→7

Adjacent vertices of 7 is 5 and 8.
5 is already visited so we insert 8 in the Queue.

Queue: 8

x) Delete vertex 8 from the Queue & visit.

Traversal: 0→1→3→4→2→6→5→7→8

No vertex is adjacent to 8.
Queue is empty, we stop our process.

Traversal complete.

Implementation of Depth First Search using Stack.

During the algorithm any vertex will be in one of the two states – Initial or visited.

At the start of the algorithm, all vertices will be in initial state, and when a vertex will be popped from stack its state will change to visited.

Algorithm of DFS using stack :-

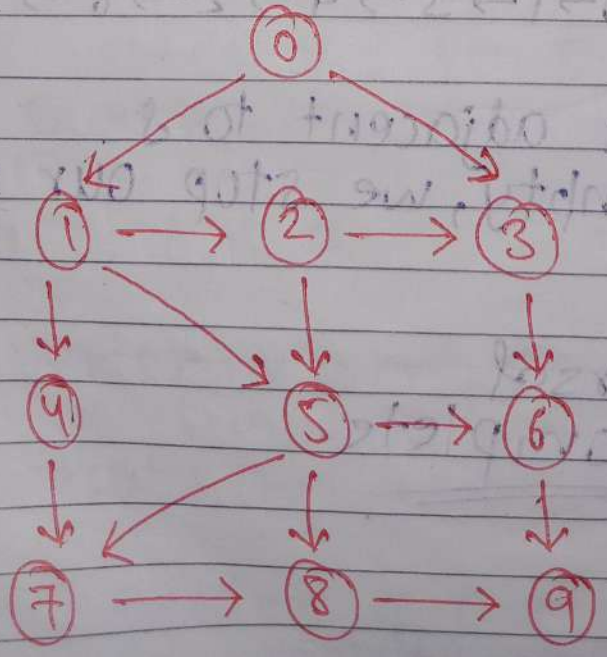
Initially stack is empty, and all vertices are in initial state.

i) Push starting vertex on the stack.

ii) Pop a vertex from the stack.

iii) If popped vertex is in initial state, visit it & change its state to visited. Push all unvisited vertices adjacent to the popped vertex.

iv) Repeat step 2 and 3 until stack is empty.



Start vertex is 0, so first push 0 in stack:

Pop	Visit	Push	Stack
		0	0
0	0	3, 1	3, 1
1	1	5, 4, 2	3, 5, 4, 2
2	2	5, 3	3, 5, 4, 5, 3
3	3	6	3, 5, 4, 5, 6
6	6	9	3, 5, 4, 5, 9
9	9	no adjacent vertex	3, 5, 4, 5
5	5	8, 7	3, 5, 4, 8, 7
7	7	8	3, 5, 4, 8, 8
8	8	adjacent vertex already visited	3, 5, 4, 8
8	already visited	adjacent vertex already visited	3, 5, 4
4	4	—	3, 5
5	already visited	—	3
3	already visited	—	empty

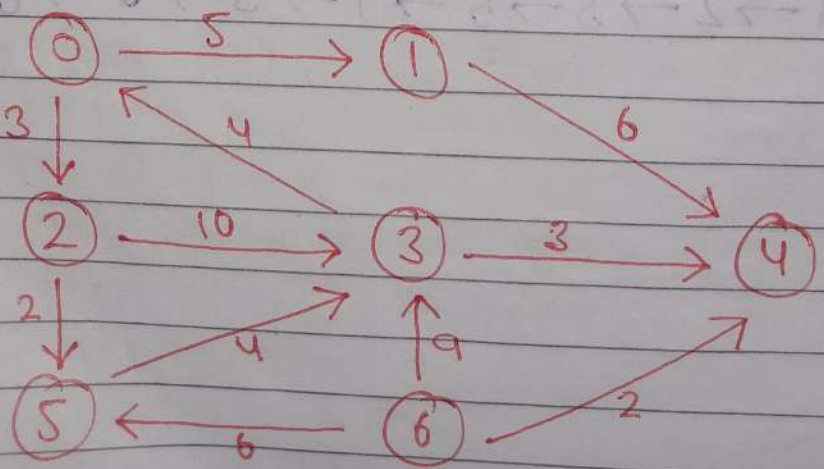
Traversal complete.

Final DFS Traversal is :-

$0 \to 1 \to 2 \to 3 \to 6 \to 9 \to 5 \to 7 \to 8 \to 4$

In BFS, if a vertex was already present in the queue then it was not inserted again in the queue. So there we changed the state of vertex from initial to waiting as soon as it was inserted in the queue, and never inserted a vertex in the queue that was in waiting state.
In DFS, we don't have the concept of waiting state & so there may be multiple copies of a vertex in the stack.
If we don't insert a vertex already present in the stack, then we will not be able to visit the vertices in depth first search order.

* Shortest Path Problem :- There can be several paths possible for going from one vertex to another vertex in a weighted graph, but the shortest path in which the sum of weights of the included edges is the minimum.



Suppose we have to go from vertex 0 to vertex 4. we can take the path $0 \rightarrow 2 \rightarrow 3 \rightarrow 4$, and the length of this path would be 16. But this is not the only path from vertex 0 to vertex 4, there are other paths also which may be shorter. we have to find the shortest of these paths.
The other two paths are $0 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 4$, (length is 12) and $0 \rightarrow 1 \rightarrow 4$, (length is 11).
So, from all 3 paths, the shortest one is $0 \rightarrow 1 \rightarrow 4$.

There are basically 3 algorithms for finding out the shortest path in a weighted graph :-

i) Dijkstra Algorithm.
ii) Bellman Ford Algorithm.
iii) Floyd's or Modified Warshall's Algorithm.