5 Graphs

Module 5

University Prescribed Syllabus

Introduction, Graph Terminologies, Representation of Graph, Graph Traversals-Depth First Search (DFS) and Breadth First Search (BFS), Graph Application-Topological Sorting.

Syllabus Topic : Introduction

5.1 INTRODUCTION

UQ. 5.1.1 What is a graph ? **MU - May 18, 1 Mark**

We know that there are various data structures available which are mainly categorized as linear and non-linear data structures.

Up till now we have seen various linear data structures like Stack, Queue and Linked list while one non-linear data structure : Tree.

Now we are going to learn another non-linear data structure that is Graph.

Graphs are very powerful and versatile data structures that easily allow us to represent real life relationships between different types of data (nodes).

There are two main parts of a graph:

1. The vertices (nodes) where the data is stored
2. The edges (connections) which connect the nodes

Definition : Graph is a collection of set of vertices V where the actual data is stored and set of edges E which connects those vertices.

The following is a graph with 5 vertices and 6 edges. This graph G can be defined as G = (V , E) Where V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.

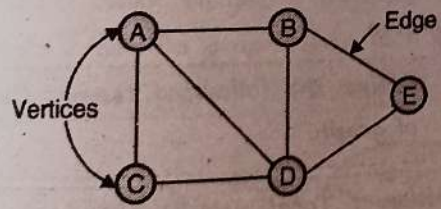


Fig. 5.1.1 : Graph

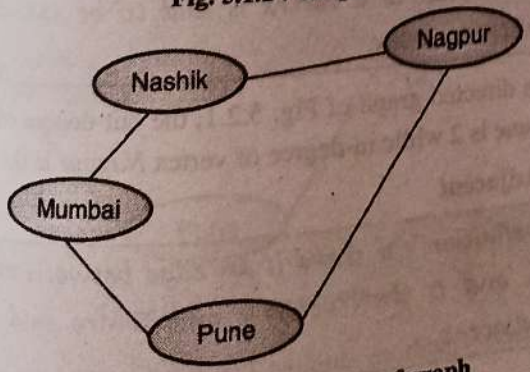


Fig. 5.1.2 : Real life example of graph

In Fig. 5.1.2 the vertices represents the cities while edges represents the road between two cities.

There are number of real life examples of graph. To understand the concept of graph exactly, we will see some of these examples :

1. Representation of road network of cities
2. Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.
3. Using GPS/Google Maps/Yahoo Maps, to find a route based on shortest route.
4. Google, to search for webpages, where pages on the internet are linked to each other by hyperlinks; each page is a vertex and the link between two pages is an edge.
5. On eCommerce websites relationship graphs are used to show recommendations.

Syllabus Topic : Graph Terminologies

5.2 GRAPH TERMINOLOGIES

We will see some important terminologies regarding Graph.

1. Node (Vertex)

Definition : Every individual element in a graph is known as Vertex. Vertex is also called as a Node.

In Fig. 5.1.1 – A, B, C, D & E are vertices.

In Fig. 5.1.2 – Pune, Mumbai, Nasik and Nagpur are vertices.

2. Arc (Edge)

Definition : An Edge is the connecting link between two vertices. Edge is also called as Arc.

An edge is represented as (startingVertex, endingVertex).

For example in Fig. 5.1.1, the link between vertices A and B is represented as (A,B).

In this graph, there are 7 edges : (A,B), (A,C), (A,D), (B,D), (B,E), (C,D) and (D,E).

Module 5

- In Fig. 5.1.2, there are 4 edges : (Pune, Mumbai), (Pune, Nagpur), (Mumbai, Nasik) and (Nasik, Nagpur).

3. Directed Edge

GQ. 5.2.1 *Define directed edge of a tree.* **(1 Mark)**

> **Definition** : The edge which has specific directions, is called as **directed edge.**

4. Undirected Graph

GQ. 5.2.2 *Describe undirected graph with diagram.* **(1 Mark)**

- Up till now we have seen all the undirected graphs. That means the edges do not show any directions.

> **Definition** : The graph in which the edges do not show any direction is called as **Undirected Graph.**

- Fig. 5.1.1 and Fig. 5.1.2 both are examples of undirected graph.
- The edges which do not have directions are called as undirected edges.
- In undirected graph every pair of vertices make two edges.
- For example, in the Fig. 5.1.2 vertices Pune and Mumbai represents two edges as (Pune, Mumbai) and (Mumbai, Pune). Both the edges are considered as same.

5. Directed Graph

GQ. 5.2.3 *Describe directed graph with diagram.* **(1 Mark)**

- Sometimes, there is need to show directions to the edges. For example, in Fig. 5.2.1 we may want to show directions to indicate whether the travelling is from Pune to Mumbai or Mumbai to Pune.

> **Definition** : The graph in which all the edges have specific directions is called as **directed graph.**

- The edges which have directions are called as directed edges.

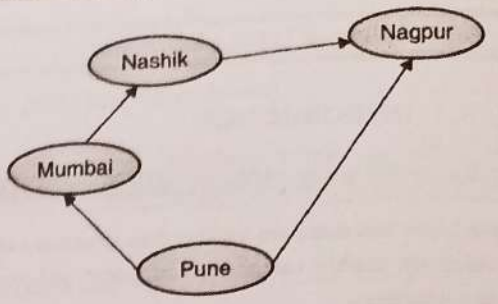


Fig. 5.2.1 : Directed graph

6. Degree

UQ. 5.2.4 *Define Degree of vertex.* **MU - May 16. 1 Mark**

> **Definition** : Total number of edges connected to a vertex is said to be **degree** of that vertex.

- In undirected graph of Fig. 5.1.2, every vertex has degree 2.

7. In-degree

GQ. 5.2.5 *Explain in-degree of a node with example.* **(2 Marks)**

> **Definition** : Total number of incoming edges connected to a vertex is said to be **in-degree** of that vertex.

- In directed graph of Fig. 5.2.1, the in-degree of vertex *pune* is 0 while in-degree of vertex *Nagpur* is 2.

8. Out-degree

GQ. 5.2.6 *Define the following term : Out-degree of graph.* **(1 Mark)**

> **Definition** : Total number of outgoing edges connected to a vertex is said to be **out-degree** of that vertex.

- In directed graph of Fig. 5.2.1, the out-degree of vertex *pune* is 2 while in-degree of vertex *Nagpur* is 0.

9. Adjacent

> **Definition** : If there is an edge between vertices A and B then both A and B are said to be **adjacent.**

In directed graph of Fig. 5.2.1, the vertices *Pune* and *Mumbai* are adjacent.

10. Successor

Definition : A vertex coming after a given vertex in a directed path is called as successor.

- In directed graph of Fig. 5.2.1, the vertex *Mumbai* is successor of vertex *Pune*.

11. Predecessor

Definition : A vertex coming before a given vertex in a directed path is called as predecessor.

- In directed graph of Fig. 5.2.1, the vertex *Pune* is predecessor of vertex *Mumbai*.

12. Relation

Definition : Relation is set of ordered pairs.

13. Weight

Definition : A numerical value, assigned as a label to a vertex or edge of a graph is called as weight.

- The weight may indicate distance between two cities or any other resources such cost, time etc.

14. Weighted Graph

GQ. 5.2.7 *Define Weighted Graph.* **(1 Mark)**

Definition : When weight is assigned to each and every edge of a graph, then such graph is called **Weighted Graph.**

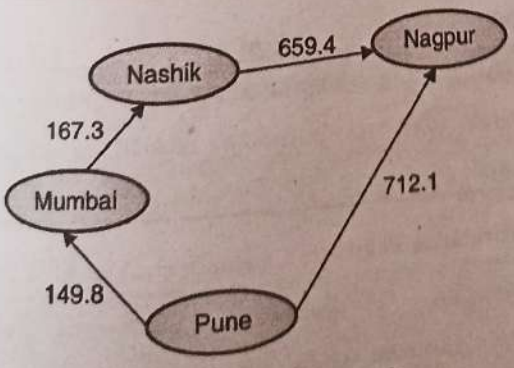


Fig. 5.2.2 : Weighted graph

- In graph of Fig. 5.2.2, the weights (numbers associated to edges) represent distance between two cities.

15. Path

Definition : A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

16. Length

Definition : Length of path is the total number of edges included in that path.

- In graph of Fig. 5.2.3, the length of path (Pune-Nasik) is 2.

17. Linear Path

Definition : The path in which the starting and ending vertices are different, is called as Linear Path.

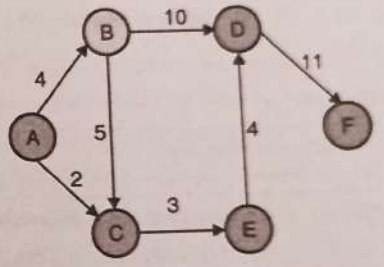


Fig. 5.2.3 : Linear path

18. Cycle

Definition : The path, in which the starting and ending vertex is same, is called as Cycle.

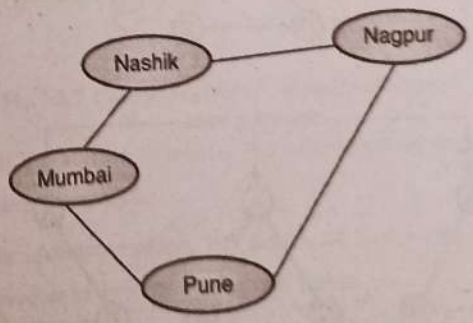


Fig. 5.2.4 : Cycle

19. Sub-graph

Definition : A subgraph of a graph G is another graph formed from a subset of the vertices and edges of G

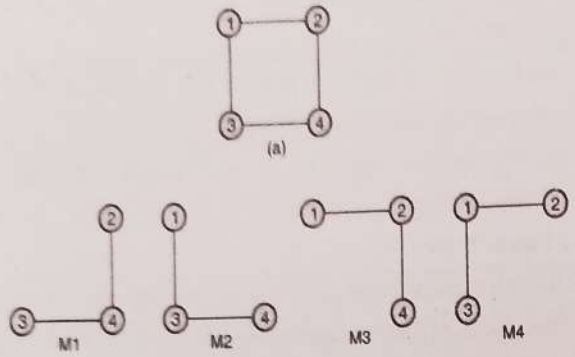


Fig. 5.2.5 : Subgraph

– In Fig. 5.2.5, MI, M2, M3 and M4 are sub-graphs of Graph – (a).

20. Spanning Tree

GQ. 5.2.8 Define spanning tree. **(1 Mark)**

Definition : A spanning tree T of an undirected graph G is a sub-graph that is a tree which includes all of the vertices of G, with minimum possible number of edges.

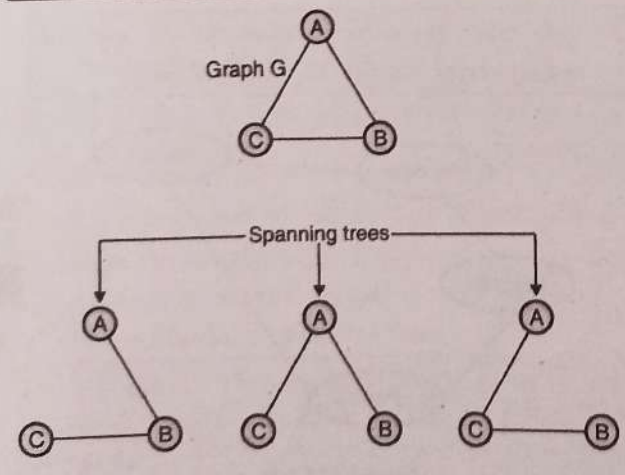


Fig. 5.2.6 : Spanning tree

21. Source

Definition : A source, in a directed graph, is a vertex with no incoming edges (in-degree equals 0).

22. Minimum Cost Spanning Tree

GQ. 5.2.9 Define Minimum cost spanning tree. **(1 Mark)**

– The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees.

Definition : Minimum cost spanning tree is the spanning tree where the cost is minimum among all the spanning trees.

– Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are :

- Cluster Analysis
- Handwriting recognition
- Image segmentation

23. Isolated Node

Definition : An isolated vertex of a graph is a vertex whose degree is zero, that is, a vertex with no incident edges.

24. Sink

Definition : A sink, in a directed graph, is a vertex with no outgoing edges (out-degree equals 0).

25. Articulation Point

Definition : A vertex in a connected graph whose removal would disconnect the graph, is called as Articulation Point.

Syllabus Topic : Representation of Graph

5.3 REPRESENTATION OF GRAPH

UQ. 5.3.1 Explain various techniques of graph representations.

MU - Dec. 13, Dec. 16, May 18, 5 Marks

UQ. 5.3.2 Explain with examples different techniques to represent the graph data structure on a computer. Give 'C' language representations for the same.

MU - Dec. 14, 5 Marks

UQ. 5.3.3 What is a graph ? Explain methods to represent a graph.

MU - May 15, 5 Marks

UQ. 5.3.4 What are the various techniques to represent Graphs in memory ?

MU - May 17, Dec. 19, 5 Marks

UQ. 5.3.5 What are different ways of representing a graph data structure on a computer ?

MU - Dec. 18, 5 Marks

- A graph can be represented in two ways :

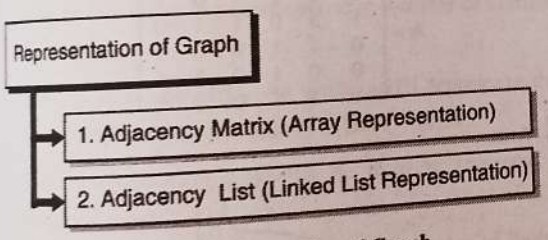


Fig. 5.3.1 : Representation of Graph

5.3.1 Adjacency Matrix (Array Representation)

GQ. 5.3.6 Explain array representation of graph. **(4 Marks)**

GQ. 5.3.7 Explain Adjacency Matrix of Graph. **(4 Marks)**

- This is a simple way to represent a graph. 2D array is used for this representation.
- In **Adjacency Matrix** Graph is represented using a matrix of size total number of vertices by total number of vertices.
- That means if a graph has 5 vertices, then a matrix of 5*5 will be used.
- In this matrix, rows and columns both represent vertices.
- All the values of this matrix are either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.
- Let the 2D array be arr[][], a slot arr[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric : the row i , column j entry is 1 if and only if the row j , column i entry is 1..
- With an adjacency matrix, we can find out whether an edge is present in constant time, by just looking up the corresponding entry in the matrix.
- For example, if the adjacency matrix is named graph, then we can query whether edge (i,j) is in the graph by looking at graph[i][j].

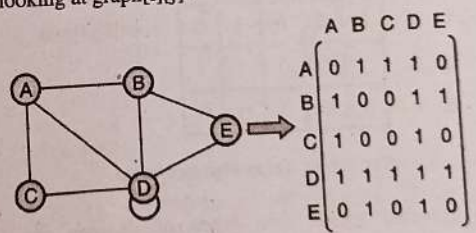


Fig. 5.3.2 : Adjacency Matrix of undirected graph

- In the adjacency matrix, the calculation of degree is very simple task.
- In undirected graph, the degree of any vertex is the number of 1s in the row of that vertex.
- Table 5.3.1 shows degrees of all the vertices of graph in Fig. 5.3.1.

Module 5

Table 5.3.1 : Degree of vertices in undirected graph

Vertex	Degree
A	3
B	3
C	2
D	5
E	2

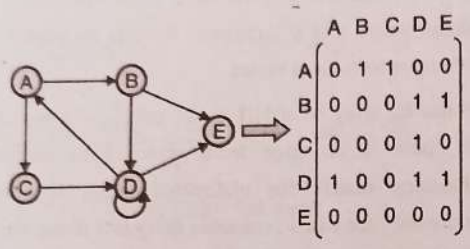


Fig. 5.3.3 : Adjacency Matrix of directed graph

- Actual array representation will be as follows :

	A	B	C	D	E
	0	1	2	3	4
A 0	0	1	1	0	0
B 1	0	0	0	1	1
C 2	0	0	0	1	0
D 3	1	0	0	1	1
E 4	0	0	0	0	0

arr[i][j]

arr[0][0] = 1

arr[2][4] = 0

Fig. 5.3.4 : Array representation

- In directed graph, two types of degrees can be calculated : In-degree and Out-degree

☞ **In-degree**

It is the total number of 1s in column of the vertex

☞ **Out-degree**

- It is the total number of 1s in row of the vertex
- Table 5.3.2 shows degrees of all the vertices of graph in Fig. 5.3.3.

Table 5.3.2 : Degree of vertices in directed graph

Vertex	In-degree	Out-degree
A	1	2
B	1	2
C	1	1
D	3	3
E	2	0

☞ **Pros of Adjacency Matrix**

Representation is easier to implement and follow. Removing an edge takes less time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient.

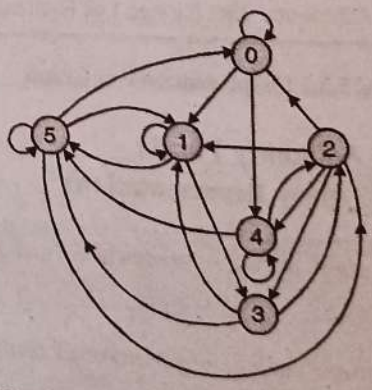
☞ **Cons of Adjacency Matrix**

Consumes more space. Even if the graph is sparse (contains less number of edges), it consumes the same space.

GQ. 5.3.8 *Draw the graph structure for following matrix* **(3 Marks)**

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

☑ **Ans. :**



GQ. 5.3.9 *The adjacency matrix is symmetric for which type of graph ?* **(1 Mark)**

☑ **Ans. : Undirected graph.**

5.3.2 Adjacency List (Linked List Representation)

GQ. 5.3.10 Explain Adjacency List of Graph. (4 Marks)

This representation is an alternative to Adjacency Matrix. The memory required by this representation is very less.

In this representation, every vertex of graph contains list of its adjacent vertices.

Example

For example, consider the following directed graph representation implemented using linked list.

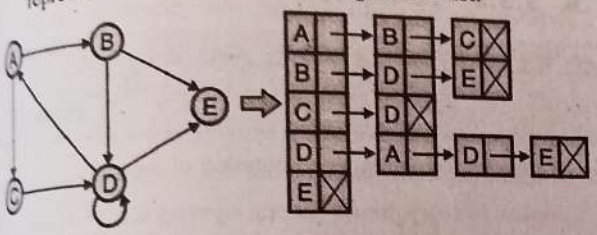


Fig. 5.3.5 : Adjacency list

- In this representation an array of linked lists is generally used.
- Size of this array is exactly equal to number of vertices in the graph.
- Let the array be arr[]. An entry arr[i] represents the linked list of vertices adjacent to the i^{th} vertex.
- This representation can also be used to represent a weighted graph.
- The weights of edges can be stored in nodes of linked lists. Fig. 5.3.6 shows the implementation of this representation using array :

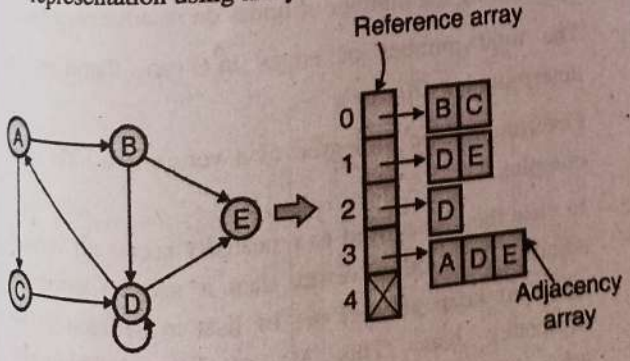


Fig. 5.3.6 : Adjacency list using array

- It is not compulsory that vertex numbers in an adjacency list to be appeared in any particular order, though it is usually convenient to list them in increasing order, as in the given example.
- It is possible to get each vertex's adjacency list in constant time, since there is only need to index into an array.
- To search whether an edge (i,j) exist in the graph, we go to i's adjacency list in constant time and then look for j in i's adjacency list.
- The degree of vertex i could be as high as |V|-1 (if i is adjacent to all the other |V|-1 vertices) or as low as 0 (if i is isolated, with no incident edges).
- In an undirected graph, vertex j is in vertex i's adjacency list if and only if i is in j's adjacency list.
- In case of weighted graph, each item in each adjacency list is either a two-item array or an object, providing the vertex number and the edge weight.

Degree of vertex

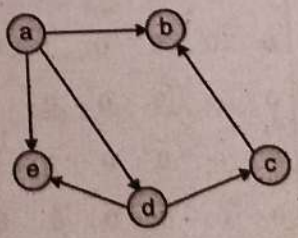
GQ. 5.3.11 Consider the following adjacency matrix.

	a	b	c	d	e
a	0	1	0	1	1
b	0	0	0	0	0
c	0	1	0	0	0
d	0	0	1	0	1
e	0	0	0	0	0

Draw the Graph. Find in-degree and out-degree of all vertices. Draw the adjacency list. (5 Marks)

☑ **Soln.**

(I) Graph of given adjacency matrix is,

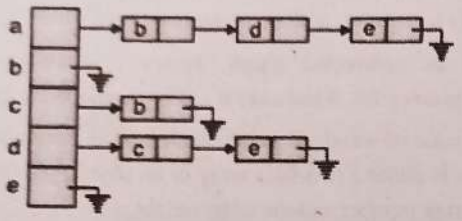


Module 5

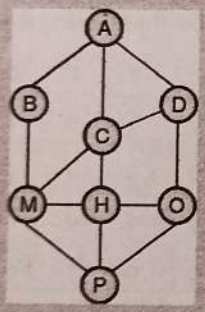
(ii) In-degree and out-degree of all vertices

Vertex	In-degree	Out-degree
a	0	3
b	2	0
c	1	1
d	1	2
e	2	0

(iii) Draw the adjacency list.



GQ. 5.3.12 *Consider the following Graph :*



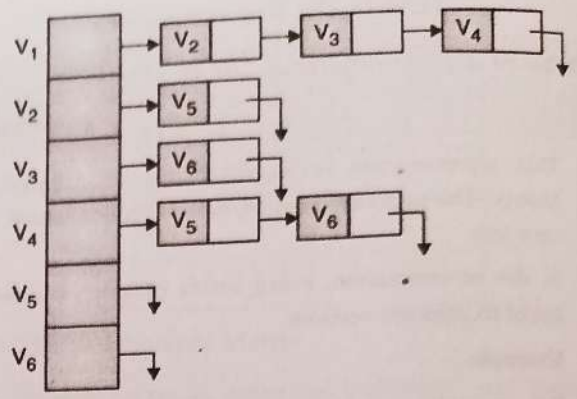
(i) Write the adjacency matrix

(ii) Draw adjacency list.

(i) Adjacency matrix

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	1	1	1	0	0
V_2	0	0	0	0	1	0
V_3	0	0	0	0	0	1
V_4	0	0	0	0	1	1
V_5	0	0	0	0	0	0
V_6	0	0	0	0	0	0

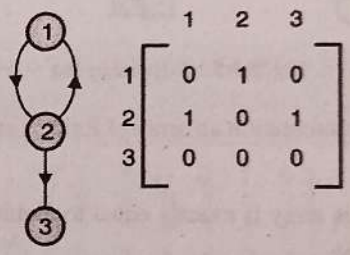
(ii) Adjacency list



5.3.3 Inverse Adjacency List

GQ. 5.3.13 *Draw the Inverse Adjacency List.* **(5 Marks)**

– Consider following representation of a graph :



– The degree of any vertex in an undirected graph may be determined by just counting the number of nodes in its adjacency list.

– The total number of edges in G may, therefore, be determined in time $O(n + e)$.

– In the case of a digraph the number of list nodes is only e. The out-degree of any vertex may be determined by counting the number of nodes on its adjacency list.

– The total number of edges in G can, therefore, be determined in $O(n + e)$.

– Determining the in-degree of a vertex is a little more complex.

– In case there is a need to repeatedly access all vertices adjacent to another vertex then it may be worth the effort to keep another set of lists in addition to the adjacency lists. This set of lists, called *inverse adjacency lists*, will contain one list for each vertex.

each list will contain a node for each vertex adjacent to the vertex it represents

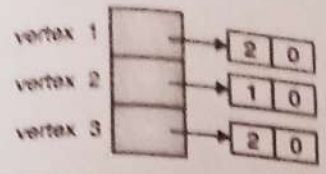


Fig. 5.3.7 : Inverse adjacency list

5.3.4 Adjacency MultiList

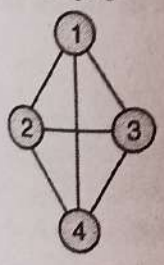
GQ. 5.3.14 *Explain graph representation using adjacency multilist with example.* **(4 Marks)**

- In the adjacency list representation of an undirected graph each edge (v_i, v_j) is represented by two entries, one on the list for v_i and the other on the list for v_j.
- As we shall see, in some situations it is necessary to be able to determine the second entry for a particular edge and mark that edge as already having been examined.
- This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists).
- For each edge there will be exactly one node, but this node will be in two lists, i.e., the adjacency lists for each of the two nodes it is incident to.

M	V_1	V_2	LINK 1 for V_1	LINK 2 for V_2

- The node structure now becomes where M is a one bit mark field that may be used to indicate whether or not the edge has been examined.
- The storage requirements are the same as for normal adjacency lists except for the addition of the mark bit M.

Consider graph



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Fig. 5.3.8 : Graph G

- Then Adjacency Multilist for Graph G :

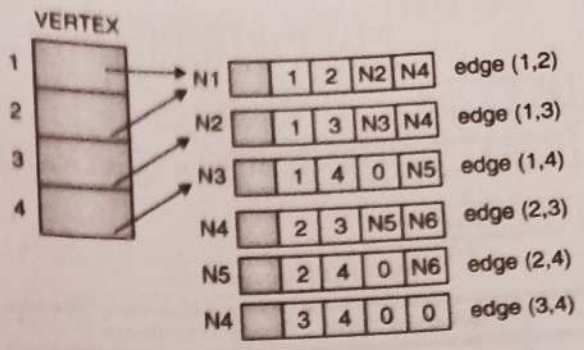


Fig. 5.3.9 : Adjacency multi list

5.3.5 Orthogonal List

GQ. 5.3.15 *How can a graph be represented as an orthogonal list ? Explain with example.* **(4 Marks)**

- Sometimes, there is need of both indegree as well as outdegree of a vertex to perform some tasks.
- For this purpose we have to maintain two separate lists which may be cumbersome.
- To solve this problem, DS provides concept of orthogonal list representation in which we can get both in-degree as well as out-degree of a vertex.
- Node structure for any directed edge is:

tail	head	Column link for head	Row link for tail

Consider Graph

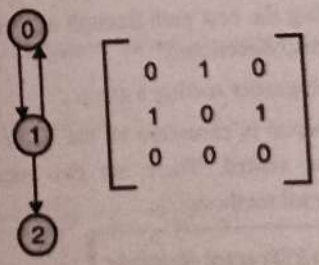


Fig. 5.3.10 : Graph G1

Orthogonal list for Graph G1

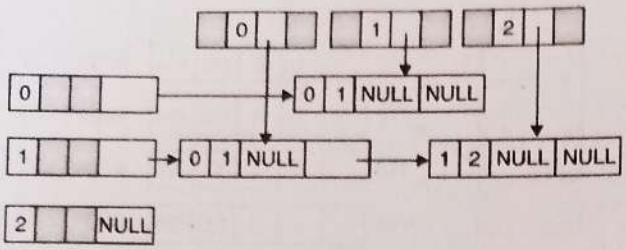


Fig. 5.3.11 : Orthogonal list

Syllabus Topic : Graph Traversals

5.4 GRAPH TRAVERSALS

UQ. 5.4.1 *Explain various graph traversal techniques with examples.* **MU - Dec. 13, Dec. 17, 8 Marks**

UQ. 5.4.2 *What are different methods for traversing the graph ? Explain DFS in detail with an example. Write a function for DFS.* **MU - Dec. 15, 10 Marks**

UQ. 5.4.3 *Explain the following : Graph Traversal Techniques.* **MU - May 18, 10 Marks**

- Graph traversal refers to the process of visiting each vertex in a graph at least once.
- There are various reasons for which we may have to traverse a graph :
  1. Finding all reachable nodes.
  2. Finding the best reachable.
  3. Finding the best path through a graph (for routing and map directions).
  4. Topologically sorting a graph.
- Graph traversal is classified by the order in which the vertices are visited. There are two most frequently graph traversal methods :

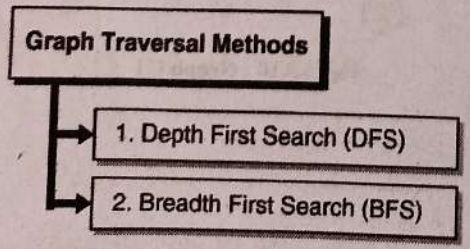


Fig. 5.4.1 : Graph Traversal Methods

Syllabus Topic : Depth First Search (DFS)

5.5 DEPTH FIRST SEARCH (DFS)

UQ. 5.5.1 *Write a function for DFS traversal of graph. Explain its working with an example.* **MU - Dec. 14, 5 Marks**

UQ. 5.5.2 *Explain the following : Graph Traversal Techniques.* **MU - May 18, 10 Marks**

UQ. 5.5.3 *Explain Depth First search (DFS) Traversal with an example.* **MU - May 19, 5 Marks**

- A **Depth-First Search (DFS)** is an algorithm for traversing a finite graph.
- DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth.
- A stack is generally used while implementing this algorithm.
- The algorithm begins with a chosen "root" vertex; it then iteratively transitions from the current vertex to an adjacent, unvisited vertex, until it can no longer find an unexplored vertex to transition to from its current location.
- The algorithm then backtracks along previously visited vertices, until it finds a vertex connected to yet more uncharted territory.
- It will then proceed down the new path as it had before, backtracking as it encounters dead-ends, and ending only when the algorithm has backtracked past the original "root" vertex from the very first step.

5.5.1 Recursive Algorithm of Depth First Search (DFS)

GQ. 5.5.4 *Write algorithm to traverse graph using Depth First Search(DFS).* **(3 Marks)**

- ***Input*** : A graph G and a vertex v of G.
- ***Output*** : A labeling of the edges in the connected component of v as discovery edges and back edges.

```
procedure DFS(G, v):
    label v as explored
    for all edges e in G.incidentEdges(v) do
        if edge e is unexplored then
            w ← G.adjacentVertex(v, e)
            if vertex w is unexplored then
                label e as a discovered edge
                recursively call DFS(G, w)
            else
                label e as a back edge
```

5.5.2 Algorithm of Non-Recursive DFS

Q. 5.5.5 *Write algorithm of non-recursive DFS.* **(3 Marks)**

- In non- recursive traversal, we use concept of stack to hold some elements.

Step 1: Initially a stack will be defined with size same as total number of vertices in the graph.

Step 2: Select the root vertex as starting node for traversal. Visit that vertex and push it in the Stack.

Step 3: Visit any nearest (adjacent) vertex of the vertex which is located at top of the stack and which is not visited and push it in the stack.

Step 4: Repeat step 3 until there are no any adjacent vertex to visit from the vertex on top of the stack.

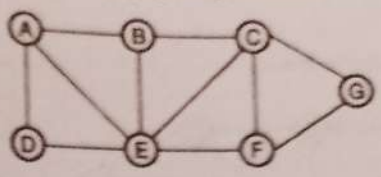
Step 5: When there is no any vertex remain to visit then use the method of back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes completely empty, then generate final spanning tree by removing unused edges from the graph.

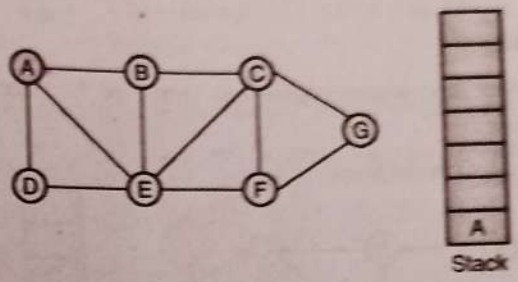
Example

- Consider following graph to perform DFS traversal.



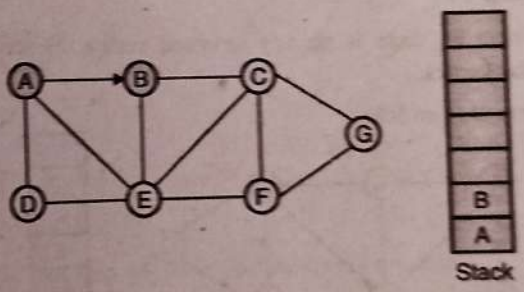
Step I

- Select the vertex A as starting point (visit A)
- Push A in the Stack.



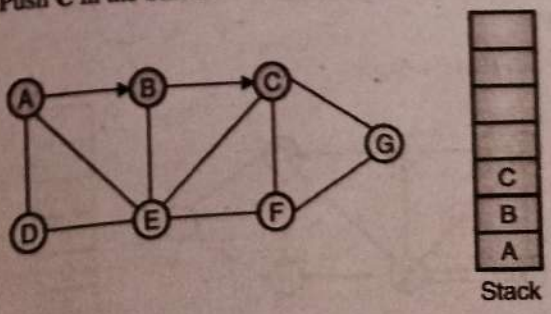
Step II

- Visit any adjacent unvisited vertex of A. Here we visit B.
- Push B in the Stack.



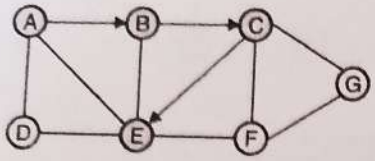
Step III

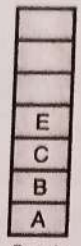
- Visit any adjacent unvisited vertex of B. Here we visit C.
- Push C in the Stack.



▶ **Step IV**

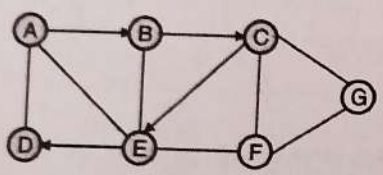
- Visit any adjacent unvisited vertex of C. Here we visit **E**.
- Push **E** in the Stack.

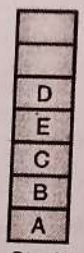




▶ **Step V**

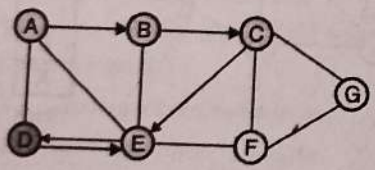
- Visit any adjacent unvisited vertex of E. Here we visit **D**.
- Push **D** in the Stack.

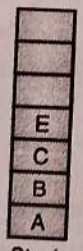




▶ **Step VI**

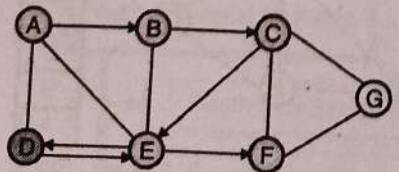
- From **D**, there is no any unvisted vertex. Hence use **Backtrack**.
- Pop **D** from Stack.

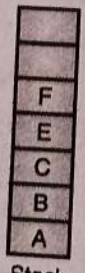




▶ **Step VII**

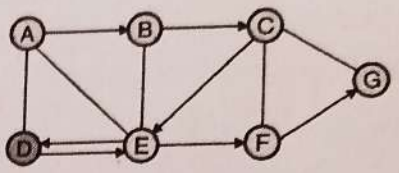
- Visit any adjacent unvisited vertex of E. Here we visit **F**.
- Push **F** in the Stack.

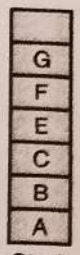




▶ **Step VIII**

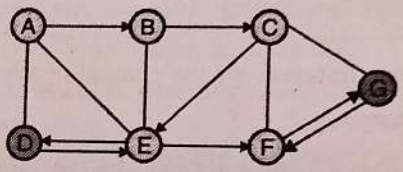
- Visit any adjacent unvisited vertex of F. Here we visit **G**.
- Push **G** in the Stack.





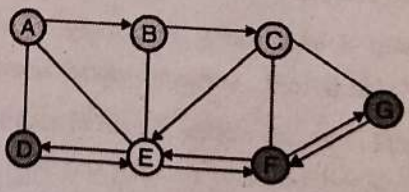
▶ **Step IX**

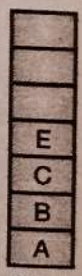
- From **G**, there is no any unvisted vertex. Hence use **Backtrack**.
- Pop **G** from Stack



▶ **Step X**

- From **F**, there is no any unvisted vertex. Hence use **Backtrack**.
- Pop **F** from Stack.

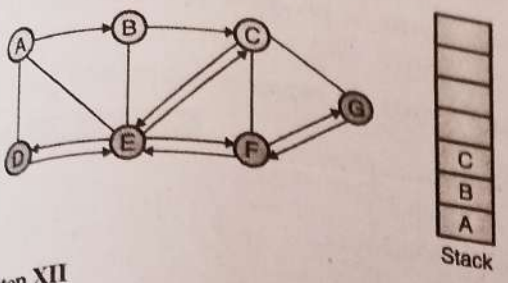




▶ **Step XI**

- From **E**, there is no any unvisted vertex. Hence use **Backtrack**.

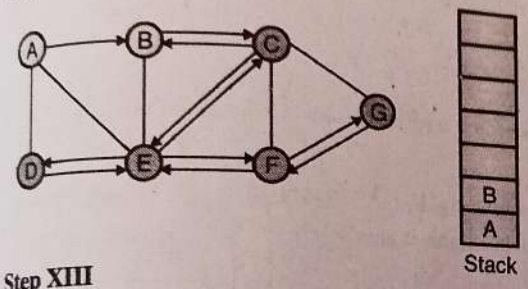
Pop **E** from Stack.



Step XII

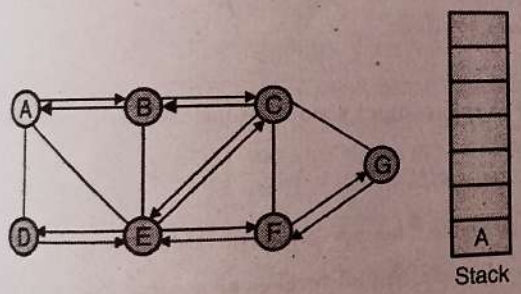
From **C**, there is no any unvisted vertex. Hence use **Backtrack**.

Pop **C** from Stack.



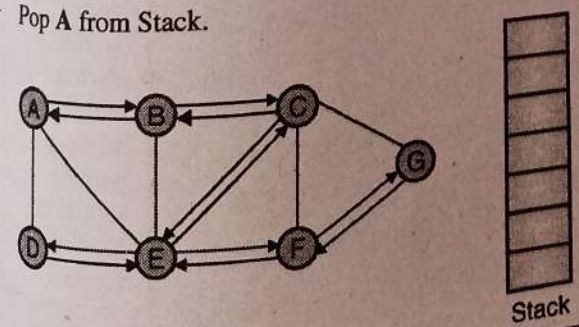
Step XIII

- From **B**, there is no any unvisted vertex. Hence use **Backtrack**.
- Pop **B** from Stack.

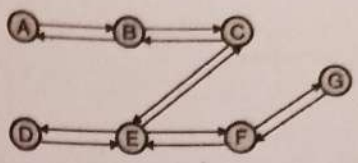


Step XIV

- From **A**, there is no any unvisted vertex. Hence use **Backtrack**.
- Pop **A** from Stack.



- Now Stack becomes empty, hence stop traversal.
- Final result of DFS is spanning tree :



The output of DFS traversal is : **A, B, C, E, D, F, G**

5.5.3 Program on DFS using Adjacency Matrix

GQ. 5.5.6 *Write a program on DFS traversal of graph using Adjacency Matrix.* **(4 Marks)**

```
#include<stdio.h>
#include<conio.h>
void DFS(int v);
typedef enum boolean{false,true}bool;
int n,arr[10][10];
bool visited[10];
void main()
{
int i,j,v;
printf("Enter total number of nodes in the graph : ");
scanf("%d",&n);
printf("Enter the adjacency matrix for the graph \n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
scanf("%d",&arr[i][j]);
}
}
printf("Enter the starting node for DFS : ");
scanf("%d",&v);
for(i=1;i<=n;i++)
visited[i]=false;
DFS(v);
```

Values for whether edge is present (1) or not (0)

```
getch();
}
void DFS(int v)
{
int i,stack[10],top=-1,pop;
top++;
stack[top]=v;
while(top>=0)
{
pop=stack[top];
top--;
if(visited[pop]==false)
{
printf("%d",pop);
visited[pop]=true;
}
else
continue;
for(i=n;i>=1;i--)
{
if(arr[pop][i]==1 && visited[i]==false)
{
top++;
stack[top]=i;
}
}
}
}
```

Extract elements from stack and print if not visited

Push elements in the stack

Output

```
C:\dfs.exe
Enter total number of nodes in the graph : 5
Enter the adjacency matrix for the graph
1 0 1 0 0
1 1 1 0 0
0 1 0 1 1
0 0 0 1 1
0 1 1 1 0
Enter the starting node for DFS : 1
13245
```

5.5.4 Program on DFS using Adjacency List

GQ. 5.5.7 Write a program on DFS traversal of graph using Adjacency List. **(4 Marks)**

UQ. 5.5.8 Write the recursive function for DFS.

MU - May 19. 5 Marks

```
#include <stdio.h>
#include <stdlib.h>

struct snode {
    int data;
    struct snode * next;
};

struct snode * add(struct snode * head, int num)
{
    struct snode * newNode = (struct snode *)
malloc(sizeof(struct snode));

    newNode->data = num;
    newNode->next = head;

    return newNode;
}
void DFSExplore(struct snode * adjacencyList[], int parent[],
int vertex)
{
    struct snode * temp = adjacencyList[vertex];
    // recursively visit all vertices accessible from this Vertex
    while (temp != NULL) {
        if (parent[temp->data] == -1) {
        parent[temp->data] = vertex;
```

We use Head insertion for

We started exploring from Vertex - 'vertex', so the Vertex - temp->data, it's parent should be our initial vertex

```
        DFSExplore(adjacencyList, parent, temp->data);
    }

    temp = temp->next;
}

DFS(struct snode * adjacencyList[], int length, int
parent[])

{
int i;

for (i = 1; i <= length; ++i) {

    if (parent[i] == -1) {
        parent[i] = 0;
        // It is a completely un-visited vertex and we start
        // our DFS from here, so it has no parent, but just
        // to mark it that we visited this node, we assign 0

        DFSExplore(adjacencyList, parent, i);

    }
}

main()
{

int vertices, edges, i, j, v1, v2;

printf("Enter the Number of Vertices : ");
scanf("%d", &vertices);

printf("Enter the Number of Edges : ");
scanf("%d", &edges);

struct snode * adjacencyList[vertices + 1];
int parent[vertices + 1];

// Must initialize the array
for (i = 0; i <= vertices; ++i) {
    adjacencyList[i] = NULL;
    parent[i] = -1;
}

printf("\nEnter edges : \n");

for (i = 1; i <= edges; ++i) {
    scanf("%d%d", &v1, &v2);
    adjacencyList[v1] = add(adjacencyList[v1], v2);
}

// Printing Adjacency List
printf("\nAdjacency List -\n\n");
for (i = 1; i <= vertices; ++i) {
    printf("adjacencyList[%d] -> ", i);

    struct snode * temp = adjacencyList[i];

    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }

    printf("NULL\n");
}

DFS(adjacencyList, vertices, parent);

getch();
}
```

Then we recursively visit everything from the child vertex

After finishing, move on to next Vertex adjacent to the vertex - 'vertex'

By this we begin to explore all the vertices reachable from Vertex i

Size is made (vertices + 1) to use the array as 1-indexed, for simplicity.

Adding edge v1 ------> v2

Output

```
C:\dfs2.exe
Enter the Number of Vertices : 3
Enter the Number of Edges : 4

Enter edges :
1 2
1 3
2 3
3 4

Adjacency List -

adjacencyList[1] -> 3 -> 2 -> NULL
adjacencyList[2] -> 3 -> NULL
adjacencyList[3] -> 4 -> NULL
```

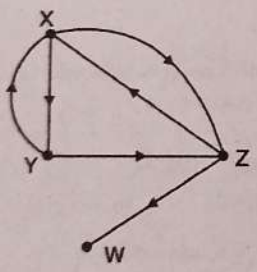
GQ. 5.5.9 *Consider the graph G.*

i) *Write Adjacency matrix representation.*

ii) *Depth first traversal sequence.*

iii) *Find all simple path from X to W.*

iv) *Find indegree (X) and outdegree (W).*



(i) Adjacency matrix representation

	W	X	Y	Z
W	0	0	0	0
X	0	0	1	1
Y	0	1	0	1
Z	1	1	0	0

(ii) Depth first traversal sequence.

X, Y, Z, W

(iii) All simple paths from X to W.

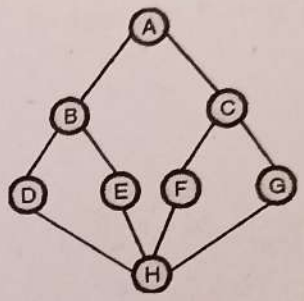
(a) $X \rightarrow Z \rightarrow W$

(b) $X \rightarrow Y \rightarrow Z \rightarrow W$

(iv) Indegree (X) and outdegree (W).

- indegree (X) - 2
- outdegree (W) – 0

GQ. 5.5.10 *Using DFS technique, list the vertices of the following graph, the way they are visited. (Let starting vertex is 'A').* **(1 Mark)**



A B D H E F C G

Syllabus Topic : Breadth First Search (BFS)

5.6 BREADTH FIRST SEARCH (BFS)

UQ. 5.6.1 *Explain BFS algorithm with example.* MU - May 14, 10 Marks

UQ. 5.6.2 *Explain the following : Graph Traversal Techniques.* MU - May 18, 10 Marks

- A **Breadth First Search (BFS)** is another technique for traversing a finite graph.
- BFS visits the neighbour vertices before visiting the child vertices, and a queue is used in the search process.
- This algorithm is often used to find the shortest path from one vertex to another.
- BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without any loops.
- We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

5.8.1 Algorithm of BFS Traversal of Graph

Q. 5.8.3 Write an algorithm for BFS traversal of a Graph. **(3 Marks)**

Input : A graph G and a vertex v of G.

Output : The closest vertex to v satisfying some conditions, or null if no such vertex exists.

```
procedure BFS(G, v):
create a queue Q
enqueue v onto Q
mark v
while Q is not empty:
    t ← Q.dequeue()
    if t is what we are looking for:
        return t
    for all edges e in G.adjacentEdges(t) do
        o ← G.adjacentVertex(t, e)
        if o is not marked:
            mark o
            enqueue o onto Q
return null
```

We use the following steps to implement BFS traversal :

Step 1: A Queue of size total number of vertices in the graph is defined.

Step 2: Any vertex as starting point for traversal is selected. Visit that vertex and insert it into the Queue.

Step 3: Visit all the nearest (adjacent) vertices of the vertex which is at front of the Queue which is not visited and insert these vertices in the Queue.

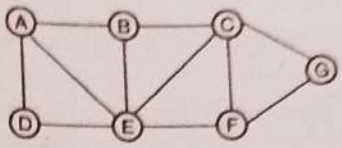
Step 4: When there is no any unvisited vertex remain from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes completely empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph.

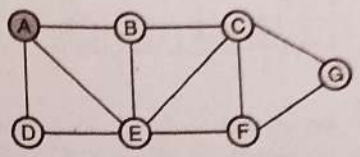
Example

Consider following graph :



▶ **Step I**

Select the vertex **A** as starting vertex. Insert **A** into Queue.



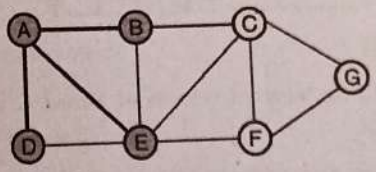
Queue

A						

▶ **Step II**

Visit all the adjacent vertices of Vertex **A** which are unvisited **(D, E, B)**.

Insert the vertices **D, E, B** into Queue and remove vertex **A** from the queue.



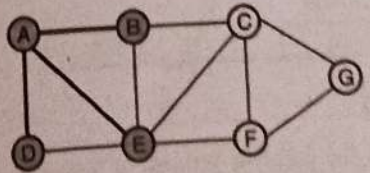
Queue

	D	E	B			

▶ **Step III**

Visit all the adjacent vertices of Vertex **D** which are unvisited **(No any vertex)**.

Remove **D** from queue.



Queue

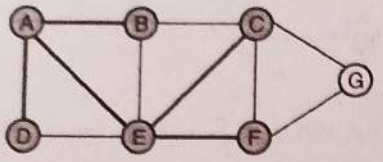
		E	B			

Module 5

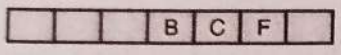
▶ **Step IV**

Visit all the adjacent vertices of Vertex **E** which are unvisited (**C, F**).

Insert the vertices **C** and **F** into Queue and remove vertex **E** from the queue.



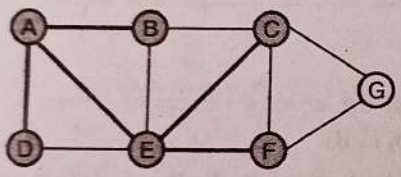
Queue



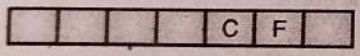
▶ **Step V**

Visit all the adjacent vertices of Vertex **B** which are unvisited (**No any vertex**).

Remove **B** from queue.



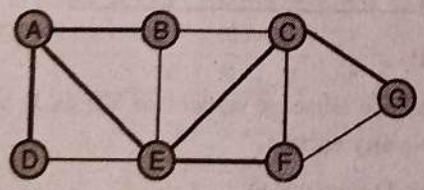
Queue



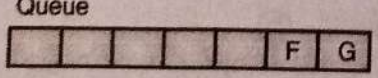
▶ **Step VI**

Visit all the adjacent vertices of Vertex **C** which are unvisited (**G**).

Insert the vertex **G** into Queue and remove vertex **C** from the queue.



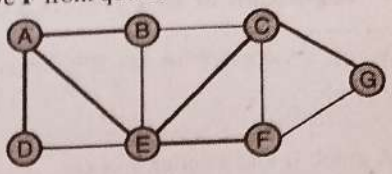
Queue



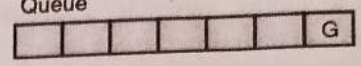
▶ **Step VII**

Visit all the adjacent vertices of Vertex **F** which are unvisited (**No any vertex**).

Remove **F** from queue.

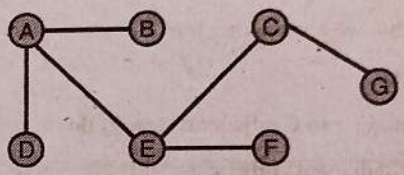


Queue



▶ **Step VIII**

- Visit all the adjacent vertices of Vertex G which are unvisited (No any vertex).
- Remove G from queue.
- As now queue is completely empty, stop the process.
- Final result of BFS is spanning tree :



- The output of BFS traversal is :

A, D, E , B, C, F, G

5.6.2 Program of BFS Traversal on Graph

UQ. 5.6.4 *Write a program in C to implement the BFS traversal of a graph. Explain the code with an example.*

MU - May 16, 10 Marks

UQ. 5.6.5 *Write a function for BFS traversal of graph.*

MU - Dec. 16, 10 Marks

UQ. 5.6.6 *Give C function for breadth first search traversal of a graph. Explain the code with an example.*

MU - Dec. 18, 10 Marks

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define SIZE 5

struct VX {
  char data;
```

```
bool visited;
};

Q[SIZE];
rear = -1;
front = 0;
count = 0;
struct VX* lstVertices[SIZE];
adj_Matrix[SIZE][SIZE];
vertexCount = 0;

void insert(int data)

Q[++rear] = data;
count++;

removeEle() {
count--;
return Q[front++];

bool isQEmpty()

return count == 0;

void insertVertex(char data) {
struct VX* VX = (struct VX*) malloc(sizeof(struct VX));
VX->data = data;
VX->visited = false;
lstVertices[vertexCount++] = VX;

void insertEdge(int start,int end)

adj_Matrix[start][end] = 1;
adj_Matrix[end][start] = 1;

void displayVertex(int vertexIndex)

printf("%c ",lstVertices[vertexIndex]->data);
```

Graph Variables

Adjacent Matrix

Add vertex in list

Add edge to edge array

```
int getAdjUnvisitedVertex(int vertexIndex)
{
  int i;

  for(i = 0; i<vertexCount; i++) {
    if(adj_Matrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
      return i;
  }

  return -1;
}

void BFS() {
  int i;
  lstVertices[0]->visited = true;

  displayVertex(0);

  //insert VX index in Q
  insert(0);

  int unvisitedVertex;

  while(!isQEmpty()) {
      int tempVertex = removeEle();

    while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {
      lstVertices[unvisitedVertex]->visited = true;
      displayVertex(unvisitedVertex);
      insert(unvisitedVertex);
    }

  }
    for(i = 0;i<vertexCount;i++) {
```

Get the adjacent unvisited VX

Marks the first node as visited

Insert vertex in Q

Get the unvisited VX of VX which is at front of the Q

No adjacent VX found

```
    lstVertices[i]->visited = false;
  }
}
```

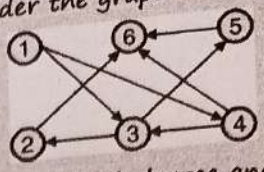
Q is empty, search is complete, reset the visited flag

```
int main() {
  int i, j;

  for(i = 0; i<SIZE; i++) // set adjacency
  {
    for(j = 0; j<SIZE; j++) // matrix to 0
      adj_Matrix[i][j] = 0;
  }

  insertVertex('K'); // 0
  insertVertex('U'); // 1
  insertVertex('N'); // 2
  insertVertex('A'); // 3
  insertVertex('L'); // 4

  insertEdge(0, 1);  // K-U
  insertEdge(0, 2);  // K-N
  insertEdge(0, 3);  //K-A
  insertEdge(1, 4);  //U-L
  insertEdge(2, 4);  //N-L
  insertEdge(3, 4);  // A-L

  printf("\nBreadth First Search: ");

  BFS();

  getch();
}
```

Output

```
C:\bfs.exe
Breadth First Search: K U N A L
```

UQ. 5.6.7 Consider the graph shown below :



(i) Write the indegree and outdegree of each vertex.

(ii) Draw the adjacency list representation.

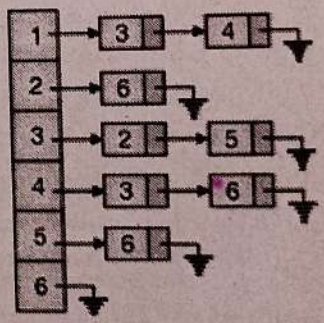
MU - Dec. 13. 5 Marks

☑ **Soln. :**

(i) Write the indegree and outdegree of each vertex.

Vertex	Indegree	OutDegree
1	0	2
2	1	1
3	2	2
4	1	2
5	1	1
6	3	0

(ii) Draw the adjacency list representation.



Syllabus Topic : Graph Applications

5.7 GRAPH APPLICATIONS

GQ. 5.7.1 State any two applications of Graph.

(1 Mark)

– Graphs can be used to model many types of relations and processes in physical, biological, social and information systems. Many practical problems can be represented by graphs.

In computer science, graphs are used to represent networks of communication, data organization, and computational devices.

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

In mathematics, graphs are useful in geometry.

Weighted graphs are used to represent structures in which pair-wise connections have some numerical values. **Ex. :** Road Network.

Graph algorithms are useful for calculating the shortest path in Routing.

Maps – finding the shortest / cheapest path for a car from one city to another, by using given roads.

5.7.1 Dijkstra's Algorithm / Shortest Path Algorithm

GQ. 5.7.2 *Explain Dijkstra's algorithm.* **(3 Marks)**

GQ. 5.7.3 *Explain Shortest Path algorithm.* **(3 Marks)**

Introduction

- **Dijkstra's algorithm** is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.
- It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.
- The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.
- For a given source node in the graph, the algorithm finds the shortest path between that node and every other.
- It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.
- For example, if the nodes of the graph represent cities and edges represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.
- As a result, the shortest path algorithm is widely used in network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF).

Dijkstra's algorithm

- Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*.
- Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

Step 1 : Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.

Step 2 : Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.

Step 3 : For the current node, consider all of its neighbours and calculate their *tentative* distances.

Step 4 : Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one.

For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbour *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.

Step 5 : When we are done considering all of the neighbours of the current node, mark the current node as visited and remove it from the *unvisited set*.

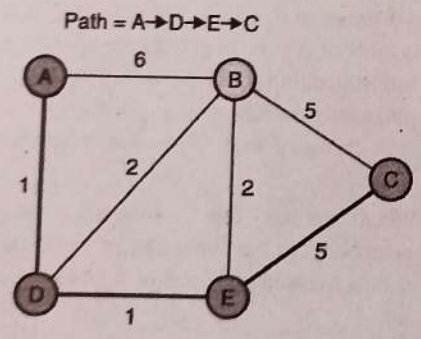
A visited node will never be checked again.

Step 6 : If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.

Step 7 : Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Pseudo code of Dijkstra's algorithm

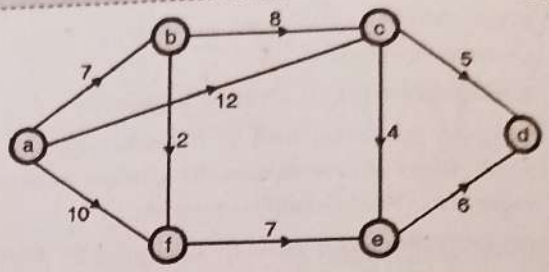
```
function Dijkstra(Graph, source):
 dist[source] ← 0      // Initialization
create vertex set Q
for each vertex v in Graph:
if v ≠ source
dist[v] ← INFINITY // Unknown distance from source to v
prev[v] ← UNDEFINED // Predecessor of v

Q.add_with_priority(v, dist[v])

 while Q is not empty:      // The main loop
   u ← Q.extract_min()  // Remove and return best vertex
   for each neighbor v of u:  // only v that is still in Q
     alt ← dist[u] + length(u, v)
     if alt < dist[v]
       dist[v] ← alt
       prev[v] ← u
       Q.decrease_priority(v, alt)
return dist, prev
```



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Fig. 5.7.1 : Shortest path

GQ. 5.7.4 *Find the shortest paths using Dijkstra's shortest path algorithm..* **(5 Marks)**



Node	Path	Distance
b	a → b	7
c	a → c	12
d	a → c → d	17
e	a → b → f → e	16
f	a → b → f	9

Syllabus Topic : Graph Application : Topological Sorting / AOV Network

5.7.2 Graph Application : Topological Sorting / AOV Network

GQ. 5.7.5 *Define the following : AOV network.* **(1 Mark)**

AOV Network

- An activity on vertex, or AOV network, is a directed graph G in which the vertices represent tasks or activities and the edges represent the precedence relation between tasks.
- Example : C3 is C1's successor, C1 is C3's predecessor

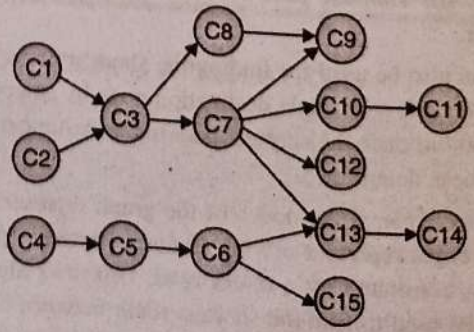


Fig. 5.7.2 : AOV network

GQ. 5.7.6 Define: Topological sort (1 Mark)

A topological order is a linear ordering of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network then i precedes j in the ordering

Example : C1 C2 C4 C5 C3 C6 C8 C7 C10 C13 C12 C14 C15 C11 C9 „

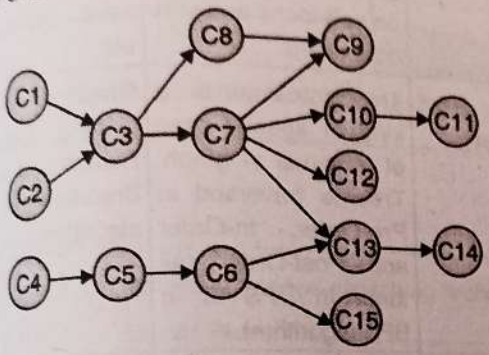


Fig. 5.7.3 : Topological sort

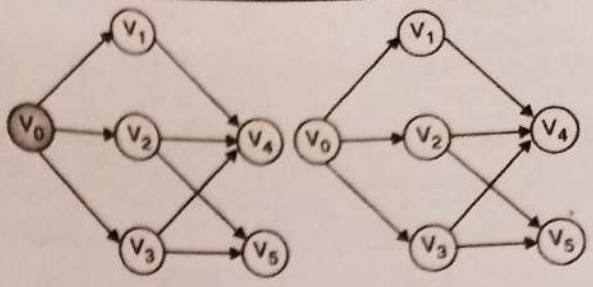
Step 1: Find a vertex v such that v has no predecessor, output it. Then delete it from network

Step 2: Repeat this step until all vertices are deleted,, Time complexity: O(|V| + |E|)

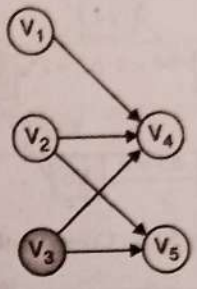
```
...(i = 0; i <n; i++)

...every vertex has a predecessor

...printf(stderr, "Network has a cycle.\n " );
...exit(1);

...ect a vertex v that has no predecessors;
...output v;
...delete v and all edges leading out of v from the
...network;
```

Example

V_0 has no predecessor, output it. Then delete it and three edges



- Choose v_3, output it



- Final result : v0, v3, v2, v1, v4, v5

5.7.3 AOE Network - Critical Path

GQ. 5.7.7 Explain the following : AOE network. (5 Marks)

AOE Network

- AOE network is an activity network closely related to the AOV network. The directed edges in the graph represent tasks or activities to be performed on a project.
- **Directed edge :** tasks or activities to be performed. **Vertex** : events which signal the completion of certain activities.
- **Number :** time required to perform the activity
- **Example :** Activities: a0, a1,... Events : v0,v1,...

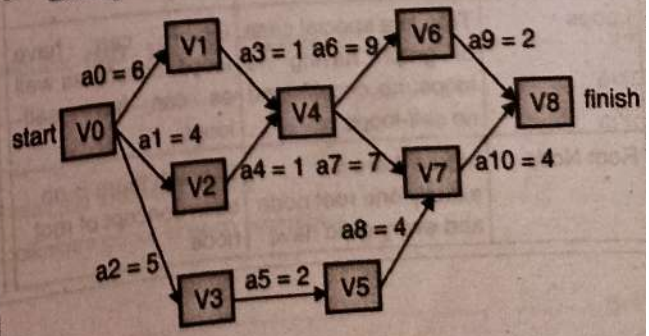


Fig. 5.7.4 : AOE network

Critical Path

GQ. 5.7.8 *Define : Critical path.* **(1 Mark)**

❑ **Definition :** *A critical path is a path that has the longest length.*

– (v0, v1, v4, v7, v8)

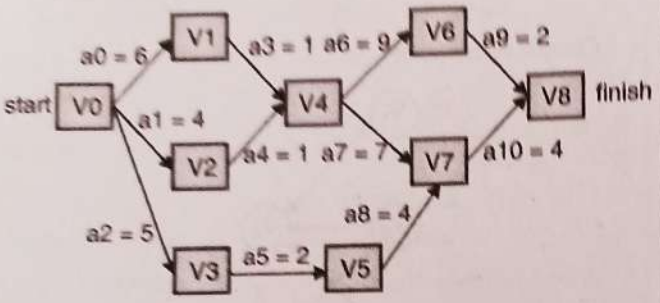


Fig. 5.7.5 : Critical path

Slack Time

GQ. 5.7.9 *Define Slack Time* **(1 Mark)**

– **Slack** is the amount of **time** that an activity can be delayed past its earliest start or earliest finish without delaying the project.

5.7.4 Differentiate between Tree and Graph

GQ. 5.7.10 *Differentiate between tree and graph.* **(4 Marks)**

Parameter	Tree	Graph
Path	Tree is special form of graph i.e. minimally connected graph and having only one path between any two vertices.	In graph there can be more than one path i.e. graph can have uni-directional or bi-directional paths (edges) between nodes
Loops	Tree is a special case of graph having no loops, no circuits and no self-loops.	Graph can have loops, circuits as well as can have self-loops.
Root Node	In tree there is exactly one root node and every child have only one parent.	In graph there is no such concept of root node.
Parent Child relationship	In trees, there is parent child relationship so flow can be there with direction top to bottom or vice versa.	In Graph there is no such parent child relationship.
Complexity	Trees are less complex than graphs as having no cycles, no self-loops and still connected.	Graphs are more complex as compare to trees as they can have cycles, loops etc.
Types of Traversal	Tree traversal is a kind of special case of traversal of graph. Tree is traversed in Pre-Order, In-Order and Post-Order (all three in DFS or in BFS algorithm)	Graph is traversed by DFS: Depth First Search and in BFS : Breadth First Search algorithm
DAG	Trees come in the category of DAG : Directed Acyclic Graphs is a kind of directed graph that have no cycles.	Graph can be Cyclic or Acyclic.
Different Types	Different types of trees are : Binary Tree , Binary Search Tree, AVL tree, Heaps.	There are mainly two types of Graphs : Directed and Undirected graphs.
Applications	Sorting and searching like Tree Traversal & Binary Search.	Coloring of maps, algorithms, Graph coloring, job scheduling, etc.
No. of edges	Tree always has n-1 edges.	In Graph, no. of edges depend on the graph.
Model	Tree is a hierarchical model.	Graph is a network model.

...Chapter Ends

❑❑❑