Data structure using C++- II

B.Sc(ECS)- II Sem-IV (2024-25)

Unit-II Topic-II. Graph

Introduction:

The only non-linear data structure that we have seen so far is 'tree'. And 'tree' is also special type of 'graph'. However, graph is a non-linear data structure which has wide range of application in real life such as: Analysis of electrical circuit, finding shortest path, Statistical analysis etc.

To be able to understand the graph data structure in details, we must get familiar with some definitions & terms associated with graph. These discussed bellow;

Graph:

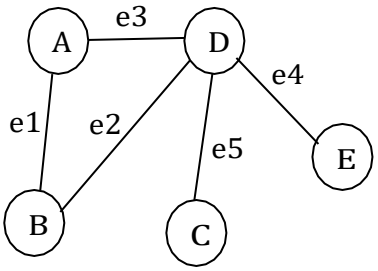
Graph is a non-linear data structure which consists from two sets viz:

1) Set of vertices say 'V' that contains finite and non-empty vertices.
2) Set of edges say 'E' that contains pair of vertices.

That is graph 'G' is given as:

G = {V, E}

Following figure shows graph:



Above graph can be represented as set of vertices 'V' and set of edges 'E' as follows:

V (G) = {A, B, C, D, E}

E (G) = {e1, e2, e3, e4, e5}

Above set of edge is also represented as;

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

Note: 1) Edge is nothing but pair of vertices which is also called 'arc'.
2) 'Vertex' is nothing but 'node'

Types of Graph:

The graph can be of two types depending on ordered or unordered pair of vertices

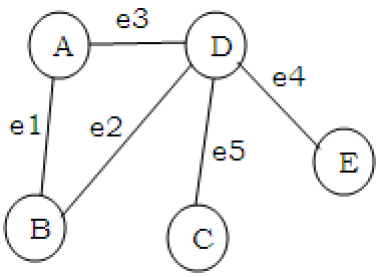
1) Undirected Graph
2) Directed Graph (Digraph)

Let us see in briefly;

1) Undirected Graph:

"The graph is said to be undirected graph if pair of vertices represents **edge is unordered**" i.e. undirected graph contains unordered pair of vertices

Following figure shows undirected graph:



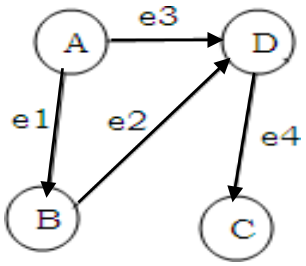
We know that, undirected graph contains unordered pair of vertices therefore an edge 'e1' can be represents as:

e1 = (A, B) = (B, A)

2) Directed Graph (Digraph):

"The graph is said to be directed graph if pair of vertices represents **edge is ordered**" i.e. directed graph contains ordered pair of vertices

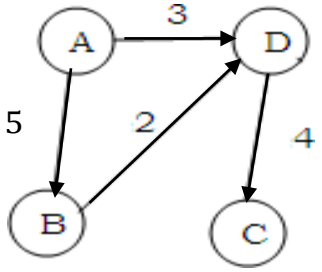
Note that: The edges of directed graph are drawn with arrow from vertex to vertex.
Following figure shows directed graph:



We know that, directed graph contains ordered pair of vertices therefore an edge 'e1'can be represents as: e1 = <A, B> which is not same as <B, A>
Also, e1=<A, B> here, 'A' is tail vertex and 'B' is head vertex.

3) Weighted Graph:

The graph is said to be weighted graph, if we assign non-negative value for every edges of graph.
The assigned non-negative value of edge is called 'Weight' or 'Cost' of edge.
Following Figure shows weighted graph:



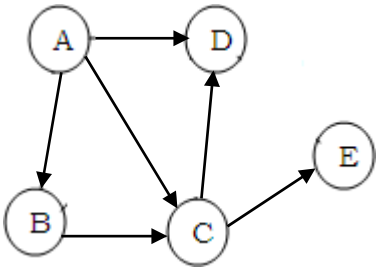
Adjacent Vertices:

The vertex 'V_0' is said to adjacent to vertex 'V_1' if there is an edge from 'V_0' to 'V_1' or 'V_1' to 'V_0' in case of undirected graph.
In above example,
Vertex A and Vertex B are adjacent.
Vertex D and Vertex C are adjacent. Etc.

Path:

Path is nothing but sequence of all vertices from source vertex 'V_0' to destination vertex 'V_n'.

Consider following graph:

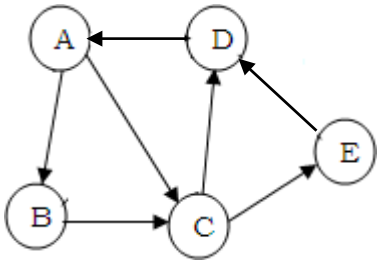


In above graph following are the paths from vertex 'A' to vertex 'E'

1) A-C-E
2) A-B-C-E

Closed Path:

The Path is said to be closed if first and last vertex of path is same.

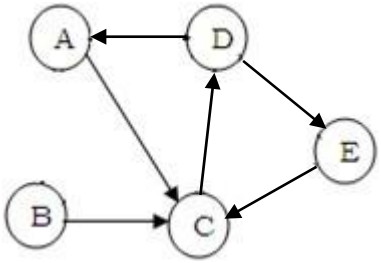


In above graph closed paths are:

1) A-B-C-D-A
2) A-C-D-A
3) A-B-C-E-D-A

Simple Path:

The simple path is a path in which all vertices are different exception that first and last vertex may be same.



In above graph simple paths are:

1) A-C-D-A
2) A-C-D-E

And the path A-**C-D**-E-**C-D**-A is not simple path, because vertex C and D are repeated.

Cycle:

The simple path in which first and last vertex is same then it is called 'Cycle'

(OR)

The closed-simple path is called 'Cycle'.

In above graph cycles are:

1) A-C-D-A
2) C-D-E-C
3) C-D-A-C

Length of path:

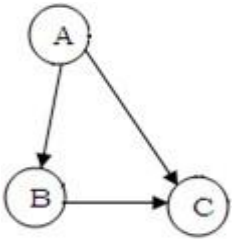
The number of edges along within the path is called as "Length of Path"

Cyclic Graph:

The graph is said to be cyclic graph if it contains the cycles.

Acyclic Graph:

The graph is said to be acyclic graph if it does not contains the cycles.

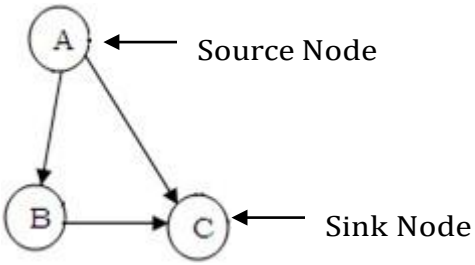


Source Node:

The node that does not have any incoming edges but it has outgoing edges then it is called as 'Source node'.

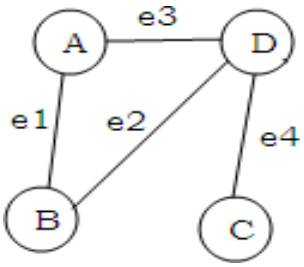
Sink Node:

The node that does not have any outgoing edges but it has incoming edges then it is called as 'Sink node'.



Degree of Node:

The number of edges attached or incident to particular node is 'Degree' of that node.
Consider given graph:

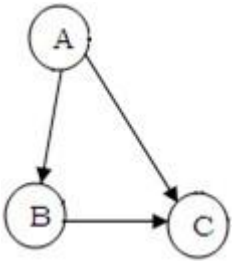


In above graph: Degree (A)= 2 Degree(D)= 3 Degree(C)= 1

In case of directed graph, node having two kinds of degrees viz,

1) **In degree:**
   The number of edges coming towards the node is the 'Indegree' of that node.
2) **Out degree:**
   The number of edges going out from that node is the 'outdegree' of that node.

Consider the graph:



In given graph, Indegree(A)= 0 Indgree(B)= 1 outdegree(A)=2

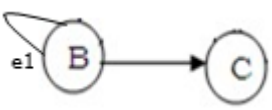
Pendant Node:

The node whose in-degree is 1 and out-degree is 0 then it is called 'Pendant node'

In above graph Node 'C' is Pendant node.

Loop (Self Edge):

An edge is said to be Loop or self edge if it starts and ends with same vertex.



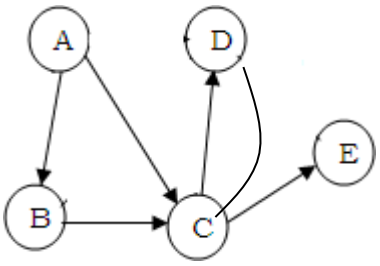
In above graph edge **e1** is Loop or self edge.

MultipleEdge:

If pair of vertices contains more than one edges then it is called as 'Multiple Edge'

Multigraph:

If any graph contains loop or multiple edges then it is called 'Multigraph'



In above graph, the vertex C and vertex D contains more than one vertex therefore it is "multigraph".

Representation of Graph:

There are two standard ways of representing graph in memory of computer. We can represent graph by using,

1) Adjacency Matrix (Sequential Representation/Array representation)
2) Adjacency List (Linked list representation).

However, we know that there are two basic components of graph viz. vertex (node) and edge (arc). To represent graph in computer memory, we have to think about these two components.

Let's see these representations briefly…

1) Adjacency Matrix: (Bit or Boolean Matrix)

Adjacency matrix 'A' is a square matrix having order $n \times n$. Here 'n' represents number of vertices in a graph. And

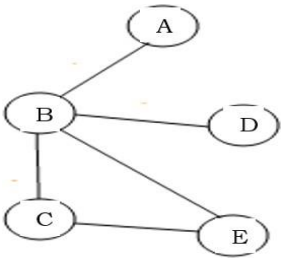
The element , A[i][j] = 1 if there is edge from vertex 'i' to vertex 'j'
The element , A[i][j]= 0 if there is no edge from vertex 'i' to vertex 'j'

That is all entries of Adjacency matrix is either 1 or 0 and it depends on presence or absence of edge between vertices.

Since, Adjacency matrix contains 1 or 0 entries only therefore it is also called as 'Bit matrix' or 'Boolean Matrix'.

- Adjacency Matrix for Undirected graph:

Consider following undirected graph,

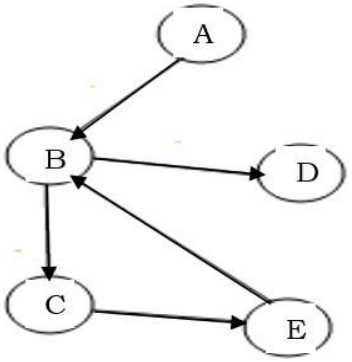


Now, we can write Adjacency matrix for above undirected graph:

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

- Adjacency Matrix for Directed graph:

Consider following directed graph,



Now, we can write Adjacency matrix for above directed graph:

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

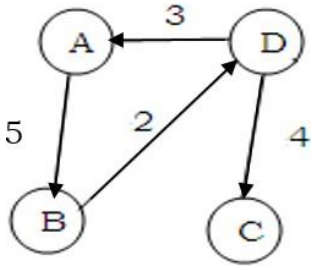
- Adjacency Matrix for Weighed graph:

We know that, Adjacency matrix is square matrix having order $n \times n$. Here, 'n' represents number of vertices in a graph. And in case of weighed graph

The element, A[i][j] = weight or cost if there is edge from vertex 'i' to vertex 'j'

The element, A[i][j] = 0 if there is no edge from vertex 'i' to vertex 'j'

Consider following weighted graph,



Now, we can write Adjacency matrix for above weighted graph as:

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 \end{pmatrix} \end{array}$$

```
// Adjacency matrix for undirected graph
#include<iostream.h>
#include<conio.h>
int    adj[10][10];
void main()
{
   int    n,edges,sc,de,i,j;
   clrscr();
   cout<<"\nEnter total nodes= ";
   cin>>n;
   cout<<"\nEnter total edges=";
   cin>>edges;
   for(i=1;i<=edges;i++)
   {
      cout<<"\nEnter "<<i<<" source vertex= ";
      cin>>sc;
      cout<<"\nEnter "<<i<<" destination vertex= ";
      cin>>de;
      if(sc>n||de>n||sc<=0||de<=0)
      {
           cout<<"\nInvalid edge ";
           i--;
      }
      else
      {
           adj[sc][de]=1;
           adj[de][sc]=1;
      }
   }
   cout<<"\nAdjacency Matrix=\n";
   for(i=1;i<=n;i++)
   {
   for(j=1;j<=n;j++)
   {
     cout<<"\t"<<adj[i][j];
   }
   cout<<"\n";
   }
   getch();
}
```

```
//Adjacency matrix for Directed graph
#include<iostream.h>
#include<conio.h>
int   adj[10][10];
void main()
{
 int   n,edges,sc,de,i,j;
 clrscr();
 cout<<"\nEnter total Vertices=";
 cin>>n;
 cout<<"\nEnter total Edges= ";
 cin>>edges;
 for(i=1;i<=edges;i++)
 {
    cout<<"\nEnter "<<i<<" source Vertex=";
    cin>>sc;
```

```
    cout<<"\nEnter "<<i<<" Destination Vertex=";
    cin>>de;
    if(sc>n||de>n||sc<=0||de<=0)
    {
      cout<<"\nInvalid Edge..";
      i--;
    }
    else
    {
       adj[sc][de]=1;
    }
 }
 cout<<"\nAdjacency Matrix=\n";
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=n;j++)
  {
    cout<<"\t"<<adj[i][j];
  }
  cout<<"\n";
 }
 getch();
}
```

```
//Adjacency matrix for weighted graph
#include<iostream.h>
#include<conio.h>
int   adj[10][10];
void main()
{
 int   n,edges,sc,de,i,j,wt;
 clrscr();
 cout<<"\nEnter total Vertices=";
 cin>>n;
 cout<<"\nEnter total Edges= ";
 cin>>edges;
 for(i=1;i<=edges;i++)
 {
    cout<<"\nEnter "<<i<<" source Vertex=";
    cin>>sc;
    cout<<"\nEnter "<<i<<" Destination Vertex=";
    cin>>de;
    cout<<"\nEnter Weight="; cin>>wt;
    if(sc>n||de>n||sc<=0||de<=0||wt<=0)
    {
      cout<<"\nInvalid Edge..";
      i--;
    }
    else
    {
       adj[sc][de]=wt;
    }
 }
 cout<<"\nAdjacency Matrix=\n";
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=n;j++)
```

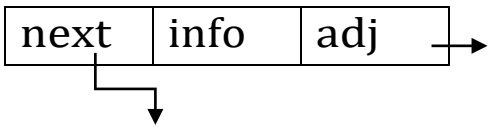
```
  {
    cout<<"\t"<<adj[i][j];
  }
  cout<<"\n";
 }
 getch();
}
```

2) Adjacency List (Linked List Representation of Graph):

In adjacency list representation of graph, we maintain two lists.

- First list will stores all the vertices (Header nodes) of graph.
- Second list will stores all the adjacent nodes for each vertex (header node) of graph.

We can define class for Header node as follow;




```
class    node
{
        int        info;
        node       *next;
        edge       *adj;
    public:
        // Declaration of operations of graph
} *start;
```

In above node structure;

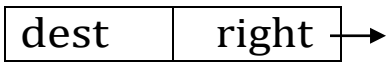
'info' is used to store actual data of header node (vertex)

'next' is **node** pointer which holds address of next header node.

'adj' is **edge** pointer which holds address of first node adjacent to header node.

'start' is **node** pointer which holds address of first header node.

We can define class for Edge as follow;



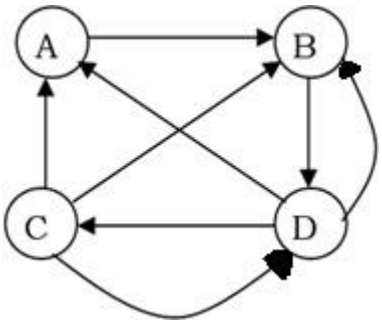

```
class          edge
{
        int       dest;
        edge      *right;
};
```

In above node structure;

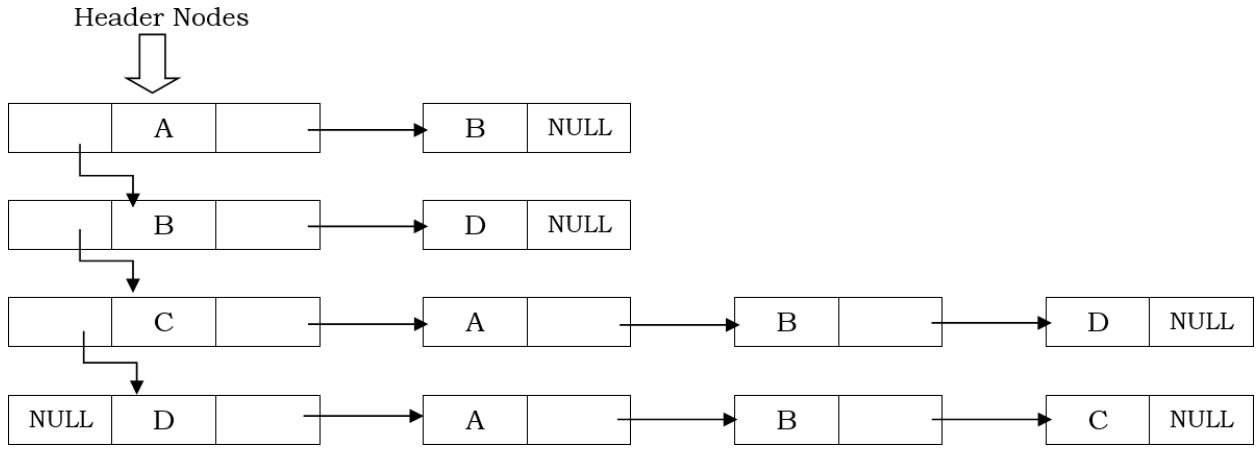
'dest' is used to store actual data of destination node

'right' is **edge** pointer which holds address of next node adjacent to header node.

Consider following directed graph,



The adjacency list representation (Linked list representation) of above graph is shown in following figure:



Operations on Graph:

1) Insert Vertex (Node):

This operation is used to insert new vertex i.e. header node in graph.

Operation:

```
void    ins_vertex(int x)
{
        node *p,*temp;
        p=new   node;
        p→next = NULL;
        p→info = x;
        p→adj = NULL;
        if ( start == NULL)       // if graph is empty
        {
                start = p;
        }
        else
        {
                temp = start;
                while(temp→next != NULL)
                {
                        temp = temp→next;
                }
```

```
                temp→next = p;
        }
}
```

2) Display Vertices:

This operation is used to display all vertices i.e. header nodes of graph.
Operation:

```
void    disp_vertex( )
{
        node *p=start;
        while( p !=NULL)
        {
                cout<<"\t"<<p→info;
                p = p→next;
        }
}
```

3) Search Vertex:

This operation is used to search whether particular vertex i.e. header node is present in graph or not.
Operation:

```
void    search(int      srch)
{
        node   *p;
        p = start;
        int f=0;
        while( p !=NULL)
        {
                if( p→info == srch)
                {
                        f =1;
                        break;
                }

                p = p→next;
        }
        if ( f ==1)
        {
                cout<<"\n Vertex is Present in Graph";
        }
        else
        {
                cout<<"\n Vertex is NOT Present in Graph";
        }

}
```

4) Insert Edge:

This operation is used to insert new edge in graph.To insert new edge in graph, first of all check source and destination vertices arevalid (i.e. they are belongs to the header nodes or not). If both source and destination vertices are valid then insert edge between them.

Operation:

```
void   ins_edge(int  sc, int   de)
{
        node *p,*q;
        int     f =0,k =0;
        p = start;
        q = start;
        while(p != NULL)              //check for source vertex
        {
                if( p→info = = sc)
                {
                        f=1;
                        break;
                }
                p = p→next;
        }
        while(q != NULL)              // check for destination vertex
        {
                if( q→info = = de)
                {
                        k=1;
                        break;
                }
                q = q→next;
        }
        if ( f == 1 && k == 1)              // if both source & destination are valid
        {
                edge   *z, *r;
                z = new   edge;
                z→dest = de;
                z→ right = NULL;
                if( p→adj==NULL)
                {
                        p→adj= z;
                }
                else
                {
                        r = p →adj;
                        while(r →right!=NULL)
                        {
```

```
                        r=r→right;
                    }
                    r→right=z;
                }
            }
            else
            {
                cout<< "\n Vertices are invalid";
            }
    }
```

5) Find adjacent vertices:

This operation is used to find all adjacent vertices of entered vertex.

Operation:

```
    void    find_adj(int    srch)
    {
        node   *p;
        int     f=0;
        p=start;
        while(p!=NULL)
        {
            if(p→info==srch)
            {
                f=1;
                break;
            }
            p=p→next;
        }
        if(f==1)
        {
            if(p→adj==NULL)
            {
                cout<<"\n Entered vertex has no adjacent vertices";
            }
            else
            {
                edge   *r;
                r = p→adj;
                cout<<"\n Adjacent vertices= ";
                while(r!=NULL)
                {
                    cout<<"\t"<<r→dest;
                    r=r→right;
                }
            }
        }
        else
        {
            cout<<"\n Vertex not found";
        }
    }
```

6) Display Graph:

This operation is used to display the graph i.e. all vertices and all edges of graph.
Operation:

```
void    disp_graph( )
{
        node   *p=start;
        edge    *r;
        while(p!=NULL)
           {
                        cout<<"\n"<<p→info;
                        r=p→adj;
                        while(r!=NULL)
                        {
                          cout<<"-->"<<r→dest;
                          r=r→right;
                        }
                p=p→next;
           }
}
```

Graph Traversal:

We know that traversal is nothing but visiting each node of graph in some systematic manner, after visiting retrieve 'info' part of visited node and display onto console.

A graph can be traversed by two ways namely -

1) Breadth First Search: **(BFS Method)**
   This traversal method uses queue which keeps the records of nodes for next processing.
2) Depth First Search: **(DFS Method)**
   This traversal method uses stack which keeps the records of nodes for next processing.

Difficulties in Graph traversal:

Traversal in graph is different from traversal in tree or linked list because of following reasons:

1) In case of graph, there is no fix starting vertex (node) hence the traversal can start from any node.
2) In tree or linked list when we traverse from first node then there is surety to traverse all the node.
   But in graph only those nodes will be traversed which are reachable from starting node. Therefore, if we want to traverse all reachable nodes, we again have to select another node as starting node for traversing the remaining nodes.
3) In a graph there is possibility to visit (traverse) a single node multiple times, but it is useless. Therefore we have to maintain the status (record) of visited node.
4) In tree or linked list, there are unique traversals. For example: preorder, inorder, postorder traversals of binary tree results in fix specific sequence.
   But in graph, the traversal may result in different sequences.

Breadth First Traversal: (Breadth First Search)

→ **Requirements:**

1) One directed graph
2) One queue which keeps the records of nodes for next processing.
3) One array say 'visited' to store the status of visited node.
4) Starting vertex to traverse the graph.

Algorithm for BFS:

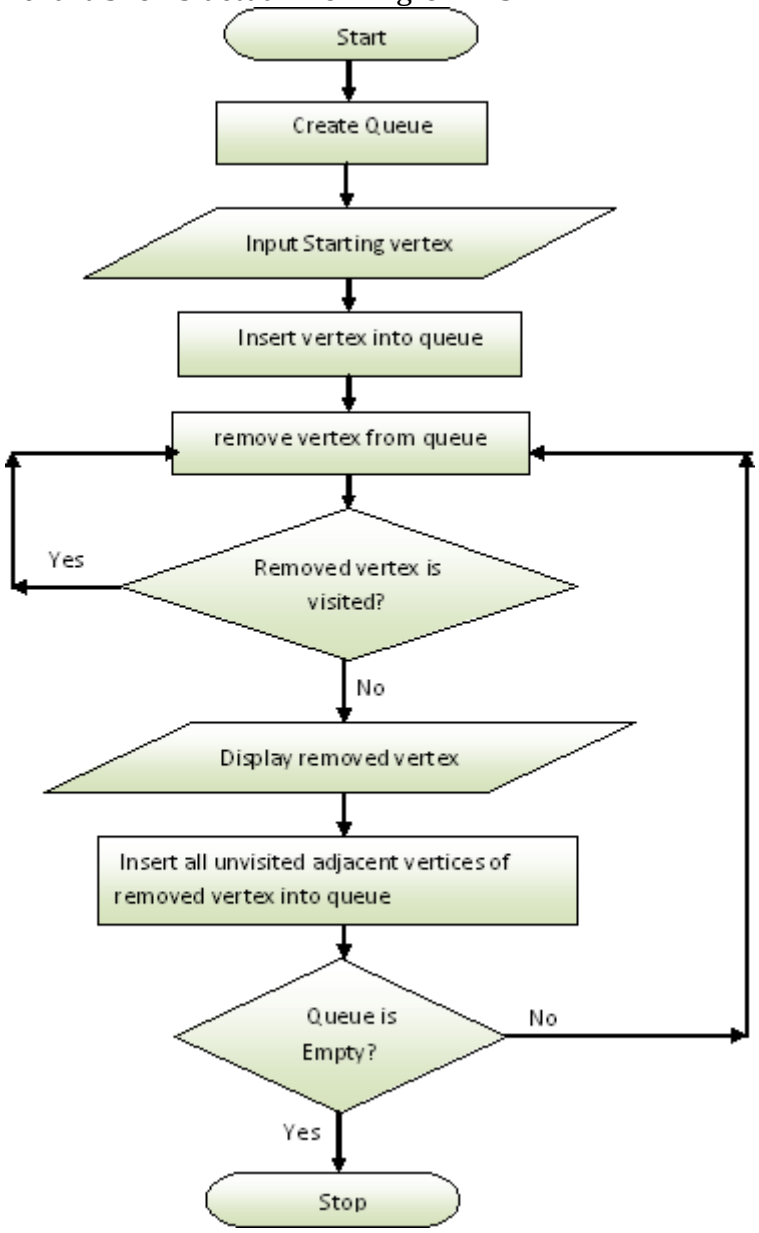
Step 1: Start
Step 2: Create Queue.
Step 3: Input starting vertex from user.
Step 4: Insert vertex into queue.
Step 5: Remove vertex from queue.
        Check removed vertex is traversed or not
            If removed vertex is not traversed then

Traverse it (Display it)
goto Step 6
Else
goto Step 5

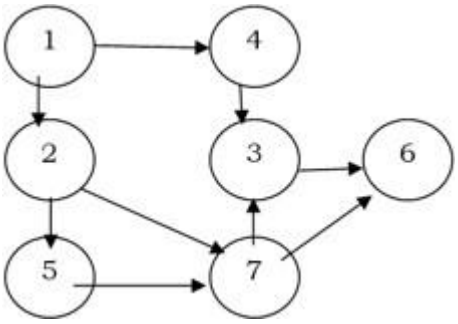
Step 6: Also, insert all unvisited adjacent vertices of removed vertex into queue.
Step 7: Repeat the step 5 until queue becomes empty.
Step 8: Stop.

Flowchart for BFS:

Following flowchart shows actual working of BFS:



E.g. Consider following graph, We traverse it according to BFS method by taking different starting vertices then we get,



Starting Node	BFS result
1	1, 2, 4, 5, 7, 3, 6
2	2, 5, 7, 3, 6
3	3, 6
4	4, 3, 6
5	5, 7, 3, 6
6	6
7	7, 3, 6

Operation of BFS:

```
void    bfs(int v)
{
    cout<<"\nBFS=>";
    int   queue[20], i, front, rear;
    front=rear=-1;
    ++rear;
    queue[rear]=v;
    while(front!=rear)
        {
            ++front;
            v=queue[front];
          if (visited[v] = = false)
          {
            cout<<"\t"<<v;
            visited[v]=true;
            for(i=1;i<=n;i++)
            {
                if(adj[v][i]==1 && visited[i]==false)
                {
                    ++rear;
                    queue[rear]=i;
                }
            }
          }
        }
}
```

/*Program to implement Breadth First Search traversal of graph*/

```
#include<iostream.h>
#include<conio.h>
#define false 0
#define true   1
int   adj[20][20];
int   n,edges;
int visited[20];
void bfs(int);
void main()
{
   int      i,source,dest,v;
   clrscr();
   cout<<"\nEnter number of Vertices= ";
   cin>>n;
   cout<<"\nEnter number of Edges=";
  cin>>edges;
  for(i=1;i<=edges;i++)
   {
     cout<<"\n Enter "<<i<<" Edge: ";
    cout<<"\nEnter Source Vertex= ";
    cin>>source;
    cout<<"\nEnter Destination Vertex= ";
    cin>>dest;
if(source>n||dest>n||source<=0||dest<=0)
        {
          cout<<"\n Invalid Edge.....";
          i--;
        }
```

```
void    bfs(int v)
{
cout<<"\nBFS=>";
int      queue[20], i, front, rear;
front=rear=-1;
++rear;
queue[rear]=v;
while(front!=rear)
      {
            ++front;
            v=queue[front];
         if (visited[v] == false)
      {
            cout<<"\t"<<v;
            visited[v]=true;
            for(i=1;i<=n; i++)
            {
          if(adj[v][i]==1 && visited[i]==false)
                  {
                     ++rear;
                     queue[rear]=i;
                  }
            }
          }
     }
}
```

```
        else
        {
            adj[source][dest]=1;
        }
    }
    cout<<"\nEnter starting vertex=";
    cin>>v;
    bfs(v);  //call
    getch();
}
```

Depth First Traversal: (Depth First Search)

➔

Requirements:

1) One directed graph
2) One stack which keeps the records of nodes for next processing.
3) One array say 'visited' to store the status of visited node.
4) Starting vertex to traverse the graph.

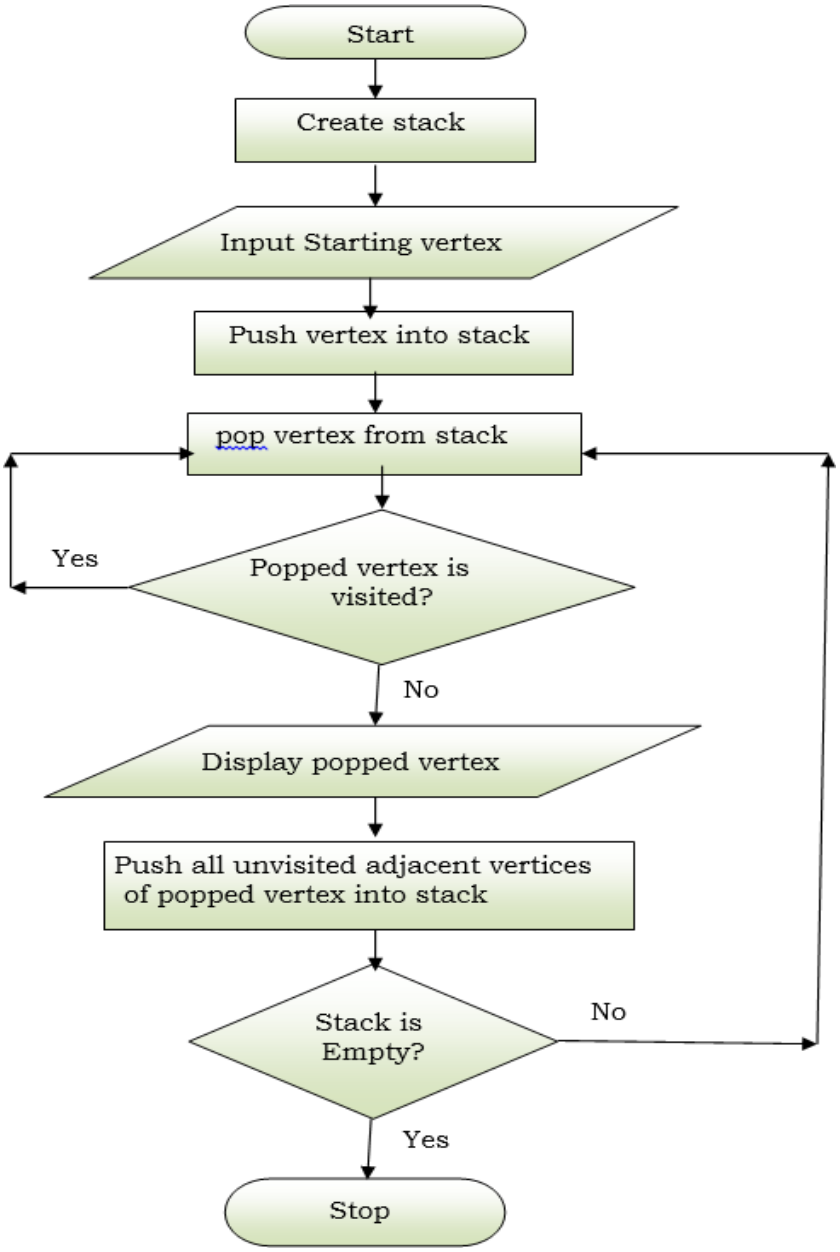
Algorithm for DFS:

Step 1: Start
Step 2: Create stack.
Step 3: Input starting vertex from user.
Step 4: Push vertex into stack.
Step 5: Pop vertex from stack.
Check popped vertex is traversed or not
If popped vertex is not traversed then
Traverse it (Display it)
goto Step 6
Else
goto Step 5

Step 6: Also, push all unvisited adjacent vertices of popped vertex into stack.
Step 7: Repeat the step 5 until stack becomes empty.
Step 8: Stop.

Flowchart for DFS:

Following flowchart shows actual working of DFS:



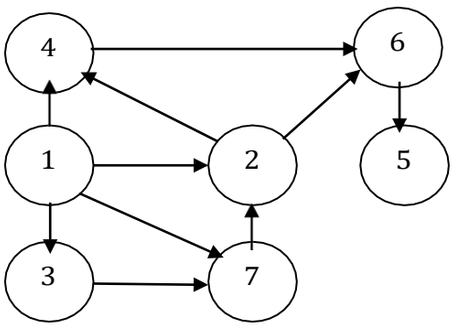
Operation for DFS:

```
void    dfs(int v)
{
    cout<<"\nDFS=>";
    int          stack[20], top, i;
        top=-1;
        ++top;
        stack[top]=v;
        while(top != -1)
            {
                    v=stack[top--];
                    if(visited[v]==false)
                    {
                        cout<<"\t"<<v;
                        visited[v]=true;
                        for(i=n;i>=1;i--)
                       {
                            if(adj[v][i]==1 && visited[i]==false)
                            {
                                ++top;
                                stack[top]=i;
                            }
                       }
                    }

            }
}
```

Program to implement Depth First Search traversal of graph (Home Work)

E.g. Consider following graph, We traverse it according to DFS method by taking different starting vertices then we get,

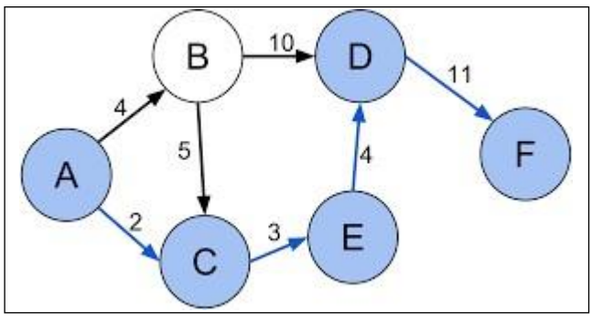


Starting Node	DFS result
1	1,2,4,6,5,3,7
2	2,4,6,5
3	3,7,2,4,6,5
4	4,6,5
5	5
6	6,5
7	7,2,4,6,5

Shortest Path:

- If total cost of a path is minimum among all other paths then that path is called "Shortest Path" **OR**
- "The shortest path is the path in which the sum of weights (costs) of the included edges is minimum" **OR**
- Shortest path is such a path whose sum of weights of edges is minimum to reach destination vertex from source vertex.
- We know that, everyone always think about shortest path to reach the destination place from source place.
- Suppose, one person wants to go from one station to another then he needs to know the shortest path from one station to another. Here, station represents thenode and tracks represent edges.
- In computer, shortest path is very useful in network for routing concepts.

E.g. consider following graph:



Here, Shortest path (from Source A to Destination F) is A-C-E-D-F having path value= 20 which is minimum among all remaining paths.

Dijkstra's Algorithm to find shortest path:

- Dijkstra's algorithm is used to find shortest path from source vertex V_1 to all vertices.
- In Dijkstra's algorithm each node of graph is labeled with dist, predecessor and status.
  - 'dist' of node represents the shortest distance of node from source node.
  - 'predecessor' of a node represents the node which precedes the given node in shortest path.
  - 'status' of node may be either 'permanent' or 'temporary'.
- (If status is 'permanent' then it suggest that it has been included in shortest path. If status is 'temporary' then it is not included in shortest path)

Procedure to find shortest path using Dijkstra's method:

I) Initially make source node as 'permanent' and make it as current working node. All other nodes are made 'temporary'.
II) Find all 'adjacent temporary' nodes of current working node.
III) Find distance of all 'adjacent temporary' nodes of current working node. Using following formula:

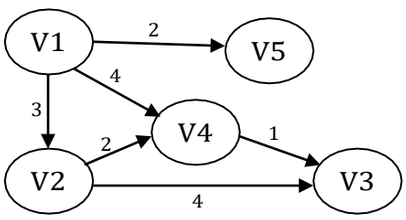
$$\text{distance(node)} = \min\left[\text{distance(node)}, \text{distance(CWN)} + \text{weight(CWN,node)}\right]$$

Here, CWN= Current Working Node

IV) Now, from all distances, select such temporary node which has minimum value of distance for next processing. Make selected node as 'permanent' and make it as current working node.
V) Repeat step II to Step IV until all nodes are made 'permanent'

Example:

Consider, following graph and find shortest path of every vertex from 'V1' vertex.



- First, select 'V1' as source vertex & make it as current working node. Also, make it as 'permanent'.

For, Dijkstra's algorithm we maintain one table as follows:

Node	Distance	Predecessor	Status
V1	0	0	Permanent
V2	∞	0	Temporary
V3	∞	0	Temporary
V4	∞	0	Temporary
V5	∞	0	Temporary

Current working node is 'V1',

$\therefore$ V1 $\xrightarrow{\text{Adjacent temp. vertices}}$ V2, V4, V5

Now, find distance of all adjacent temporary vertices:-

- distance(V2) = min[distance(V2), distance(V1) +wt(V1,V2)]
  distance (V2) = min [∞ , 0 + 3]
  distance (V2) = 3

- distance(V4) = min[distance(V4), distance(V1) +wt(V1,V4)]
  distance (V4) = min [∞ , 0 + 4]
  distance (V4) = 4
- distance(V5) = min[distance(V5), distance(V1) +wt(V1,V5)]
  distance (V5) = min [∞ , 0 + 2]
  distance (V5) = 2

Now, update previous table with values of distances of V2, V4 & V5. Also, update its predecessor to 'V1'.

Imp.Note: Previous distance and predecessor of node is updated if and only if calculated distance of node is **minimum** than previous distance.

Also, Select minimum distance temporary node for next processing.
'V5' is minimum distance temporary node.
$\therefore$ Take, 'V5' is current working node. Hence, change its status to 'permanent'

Node	Distance	Predecessor	Status
V1	0	0	Permanent
V2	3	V1	Temporary
V3	∞	0	Temporary
V4	4	V1	Temporary
V5	2	V1	Permanent

Now, Current working node is 'V5',

$\therefore$ V5 $\xrightarrow{\text{Adjacent temp. vertices}}$ No such vertex

Now, 'V2' is temporary node having minimum distance. Therefore, select it as current working node & also change its status to 'permanent'

$\therefore$

Node	Distance	Predecessor	Status
V1	0	0	Permanent
V2	3	V1	Permanent
V3	∞	0	Temporary
V4	4	V1	Temporary
V5	2	V1	Permanent

Current working node is 'V2',

$\therefore$ V2 $\xrightarrow{\text{Adjacent temp. vertices}}$ V3, V4

Now, find distance of all adjacent temporary vertices:-

- distance(V3) = min[distance(V3), distance(V2) +wt(V2,V3)]
  distance (V3) = min [∞ , 3 + 4]
  distance (V3) = 7

- distance(V4) = min[distance(V4), distance(V2) +wt(V2,V4)]
  distance (V4) = min [4 , 3 + 2]
  distance (V4) = 4
  Now, we update above table with value of distance of 'V3' only.

[Since, distance (V4) from V1 is 4 which is same here ∴ do not update V4]

Node	Distance	Predecessor	Status
V1	0	0	Permanent
V2	3	V1	Permanent
V3	7	V2	Temporary
V4	4	V1	Temporary
V5	2	V1	Permanent

Now, 'V4' becomes current working node, since it has minimum distance than 'V3'
∴ Change status of 'V4' to permanent, we get;

Node	Distance	Predecessor	Status
V1	0	0	Permanent
V2	3	V1	Permanent
V3	7	V2	Temporary
V4	4	V1	Permanent
V5	2	V1	Permanent

Current working node is 'V4',

∴ V4 $\xrightarrow{\text{Adjacent temp. vertices}}$ V3

- distance(V3) = min[distance(V3), distance(V4) +wt(V4,V3)]
  distance (V3) = min [7 , 4 + 1]
  distance (V3) = 5

Now, update above table with value of distance of 'V3' & its predecessor to 'V4'
We get,

Node	Distance	Predecessor	Status
V1	0	0	Permanent
V2	3	V1	Permanent
V3	5	V4	Temporary
V4	4	V1	Permanent
V5	2	V1	Permanent

Now, from all temporary vertex or node 'V3' has smallest value of distance so make it 'permanent'. ∴ We get,

Node	Distance	Predecessor	Status
V1	0	0	Permanent
V2	3	V1	Permanent
V3	5	V4	Permanent
V4	4	V1	Permanent
V5	2	V1	Permanent

From above table; Shortest path:

1) V1 → V3
⇨
Pred(V3)= V4 ↑
Pred(V4)= V1

∴ V1-V4-V3 is shortest path
⇨ 4 + 1
⇨ 5

2) V1 → V4
⇨
Pred(V4)= V1

∴ V1-V4 is shortest path
⇨ 4

Applications of Graph data structure:

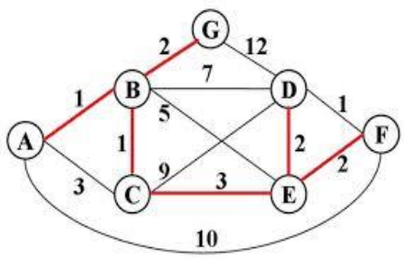
1) In computer science, graphs are used to represent networks of communication, data organization etc.
2) The link structure of a website could be represented by a directed graph.
3) Weighted graphs are used to represent structures in which pair wise connections have some numerical values. For example if a graph represents a road network, the weights could represent the length of each road.
4) Graph is used to model and analyze traffic networks.
5) Graphs are used for analysis of electrical circuit, finding shortest path, Statistical analysis etc.

Theory Assignment

1) What is Graph? Write its different applications.
2) Explain the following terms:

1) Directed Graph	11) Cyclic graph
2) Undirected Graph	12) Source node
3) Weighted Graph	13) Sink node
4) Adjacent vertices	14) Pendant node
5) Path	15) Loop
6) Closed path	16) Multiple edge
7) Simple path	17) Multi graph
8) Cycle	18) Indegree
9) Shortest path	19) Outdegree
10) Length of path	

3) What is Adjacency Matrix? Represent any directed and undirected graph by using adjacency matrix.
4) Explain, how we can represent graph by using adjacency list (by using linked list) with one example.
5) Explain following graph operations:
   I) Insert Vertex II) Display Vertex III) Search Vertex
   IV) Insert Edge V) Find adjacent nodes VI) Display Graph
6) What are the problems or difficulties in graph traversal?
7) Explain BFS traversal of graph.
8) Explain DFS traversal of graph.
9) Explain Dijkstra's algorithm to find shortest path between two vertices.
10) Find shortest path (From Node A to F) using Dijkstra's algorithm for following graph-



Practical Assignment

1) Write a program to represent directed graph using adjacency matrix.

2) Write a program to represent undirected graph using adjacency matrix.

3) Write a program to represent weighted directed graph using adjacency matrix.

4) Write a program to implement graph using linked list with all basic operations.

5) Write a program to implement Breadth first search (BFS) traversal of graph.

6) Write a program to implement Depth first search (DFS) traversal of graph.