Graph

A graph G is a collection of two sets V & E where V is the collection of vertices $v_0, v_1, \ldots\ldots v_{n-1}$ also called nodes and E is the collection of edges $e_1, e_2, \ldots.e_n$ where an edge is an arc which connects two nodes. This can be represented as-

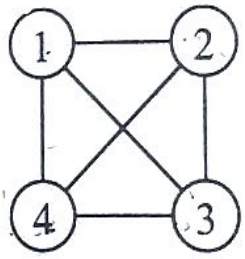
G = (V,E)
V(G) = $(v_0, v_1, \ldots\ldots v_n)$ or set of vertices
E(G) = $(e_1, e_2, \ldots\ldots e_n)$ or set of edges

A graph can be of two types-
1. Undirected graph
2. Directed graph

Undirected Graph-

A graph, which has unordered pair of vertices, is called undirected graph. Suppose there is an edge between v_0 & v_1 then it can be represented as (v_0,v_1) or (v_1,v_0) also

V(G) = {1,2,3,4}
E(G) = { (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) }

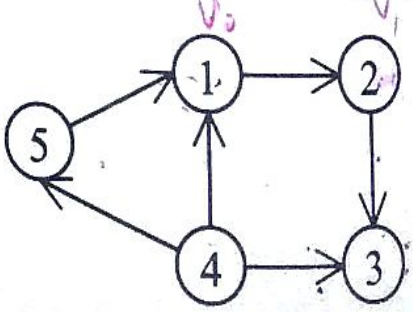


Here we can see this graph has 4 nodes and 6 edges.

Directed Graph-

A directed graph or digraph is a graph which has ordered pair of vertices $<v_1,v_2>$ where v_1 is the tail and v_2 is the head of the edge. In this type of graph each edge has direction, means <v1,v2> and <v2,v1> will represent different edges. Here the edge means that a direction will be associated with that edge. Directed graph is also known as digraph.

V(G) = {1,2,3,4,5}
E(G) = { <1,2>, <2,3>, <4,3>,<4,1>,<4,5>,<5,1> }



This graph has 5 nodes and 6 edges.

Weighted graph A graph is said to be weighted if it's edges have been assigned some non negative value as weight. A weighted graph is also known as network. Graph G9 is a weighted graph.

Adjacent nodes A node u is adjacent to another node or is a neighbour of another node

Representation of Graph-

We have mainly two components in graph, nodes & edges. Now we have to design the data structure to keep these components in mind. There are two ways for representing the graph in computer memory. First one is the sequential representation and second one is linked list representation.

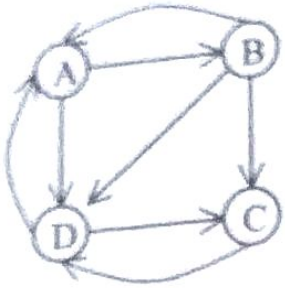
1. Adjacency Matrix-

Adjacency matrix is the matrix, which keeps the information of adjacent nodes. In other words, we can say that this matrix keeps the information that whether this node is adjacent to any other node or not. We know very well that we can represent a matrix in two dimensional array of n x n or array[n][n], where first subscript will be row and second subscript will be column of that matrix. Suppose there are 4 nodes in graph then row1 represents the node1, row2 represents the node2 and so on. Similarly column1 represents node1, column2 represents node2 and so on. The entry of this matrix will be as-

Adj[i][j] = 1 If there is an edge from node i to node j
= 0 If there is no edge from node i to node j

Hence all the entries of this matrix will be either 1 or 0.

Let us take a graph-



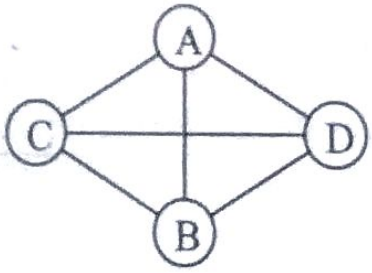
The corresponding adjacency matrix for this graph will be-

Adjacency Matrix A =

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

This adjacency matrix is maintained in the array arr[4][4]. Here the entry of matrix arr[0][1] = 1, which represents there is an edge in the graph from node A to node B. Similarly arr[2][0] = 0, which represents there is no edge from node C to node A.

Let us take an undirected graph-



The corresponding adjacency matrix for this graph will be-

Adjacency Matrix A =

	A	B	C	D
A	0	1	1	1
B	1	0	1	1
C	1	1	0	1
D	1	1	1	0

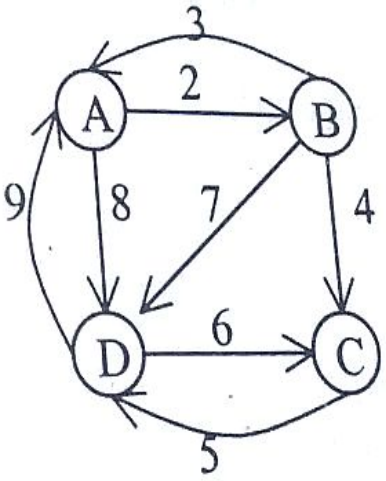
Note that the adjacent matrix for an undirected graph will be a symmetric matrix. This implies that for every i and j, A[i][j]=A[j][i] in an undirected graph.
In an undirected graph rowsum and columnsum for a node is same and represents the degree of that node and in directed graph rowsum represents the outdegree and columnsum represents the indegree of that node.

Suppose a graph has some weight on its edge then the elements of adjacency matrix will be defined as-

Arr[i][j] = Weight on edge If there is an edge from node i to node j.

Arr[i][j] = 0 Otherwise

Let us take the same graph-



The corresponding adjacency matrix will be as-

Weighted Adjacency Matrix W =

	A	B	C	D
A	0	2	0	8
B	3	0	4	7
C	0	0	0	5
D	9	0	6	0

Here all the elements of matrix represent the weight on that edge.

matrix */

Traversal In Graph-

Previously we have seen that traversal is nothing but visiting each node in some systematic approach. As we traverse a binary tree in preorder, postorder and inorder manner. Graph is represented by it's nodes and edges. So traversal of each node is the traversing in graph. There are two efficient techniques for traversing the graph. First is depth first search and second is breath first search. We maintain graph in an adjacency matrix or in adjacency list for nodes of graph and edge from one node to other. In breadth first search, we use queue for keeping nodes, which will be used for next processing, and in depth first search we use stack, which keeps the node for next processing.

Traversal in graph is different from traversal in tree or list because of the following reasons-

(a)There is no first node or root node in a graph, hence the traversal can start from any node.

(b) In tree or list when we start traversing from the first node, all the nodes are traversed but in graph only those nodes will be traversed which are reachable from the starting node. So if we want to traverse all the reachable nodes we again have to select another starting node for traversing the remaining nodes.

(c) In tree or list while traversing we never encounter a node more than once but while traversing graph ,there may be a possibility that we reach a node more than once. So to ensure that each node is visited only once we have to keep the status of each node whether it has been visited or not.

(d) In tree or list we have unique traversals. For example if we are traversing the tree in inorder there can be only one sequence in which nodes are visited. But in graph ,for the same technique of traversal there can be different sequences in which nodes can be visited .

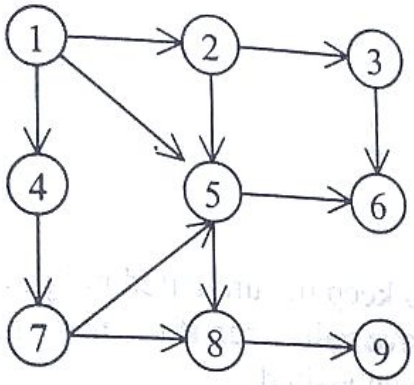
Breadth First Search-

This graph traversal technique uses queue for traversing all the nodes of the graph. In this, first we take any node as a starting node then we take all the nodes adjacent to that starting node. Similar approach we take for all other adjacent nodes, which are adjacent to the starting node and so on. We maintain the status of visited node in one array so that no node can be traversed again.

Suppose V_0 is our starting node and V_1, V_2, V_3 are nodes adjacent to it. V_{11}, V_{12}, V_{13} are nodes adjacent to V_1, V_{21}, V_{22} are nodes adjacent to V_2 and V_{31} is adjacent to V_3. So we will traverse V_0 first and then all nodes adjacent to V_0 i.e V_1, V_2, V_3. Then we will traverse all nodes adjacent to V_1 i.e V_{11}, V_{12}, V_{13} and then we will traverse nodes adjacent to V_2 i.e V_{21}, V_{22} and then nodes adjacent to V_3 i.e V_{31}. Traversal will be in following order-

V_0 V_1 V_2 V_3 V_{11} V_{12} V_{13} V_{21} V_{22} V_{31}

Let us take a graph and apply breadth first traversal to it.



Take the node 1 as starting node and start the traversal of graph.

First we will traverse the node 1. Then we will traverse all nodes adjacent to node1 i.e 2,5,4. Here we can traverse these three nodes in any order. So we can see that traversal is not unique. Suppose we traverse these nodes in order 2,4,5.

So now the traversal is-

1 2 4 5

Now first we traverse all the nodes adjacent to 2,then all the nodes adjacent to 4 then all the nodes adjacent to 5. So first we will traverse 3 ,then 7 and then 6,8.

So now the traversal is -

1 2 4 5 3 7 6 8

Now we will traverse one by one all the nodes adjacent to nodes 3,7,6,8. We can see that node adjacent to node 3 is node 6 , but it has already been traversed so we will just ignore it and proceed further. Now node adjacent to node 7 is node 8 which has already been traversed so ignore it also .Node 6 has no adjacent nodes. Node 8 has node 9 adjacent to it which has not been traversed, so traverse node 9.

So now the traversal is-

1 2 4 5 3 7 6 8 9

This was the traversal when we take node 1 as the starting node. Suppose we take node 2 as the starting node. Then applying above technique, we will get the following traversal-

2 3 5 6 8 9

We can see that all the nodes are not traversed when starting node is 2. The nodes which are in the traversal are those nodes which are reachable from 2. See the different traversals when we take different starting nodes.

Start Node	Traversal
1	2 4 5 3 7 6 8 9
2	2 3 5 6 8 9
3	3 6
4	4 7 5 8 6 9
5	5 6 8 9
6	6
7	7 5 8 6 9
8	8 9
9	9

Breadth First Search through queue-

Take an array queue which will be used to keep the unvisited neighbours of the node.
Take a boolean array visited which will have value true if the node has been visited and will have value false if the node has not been visited.
Initially queue is empty and front= -1 and rear= -1
Initially visited[i]=false where i =1 to n, n is total number of nodes.

Procedure-

1. Insert starting node into the queue.
2. Delete front element from the queue and insert all its unvisited neighbours into the queue at the end ,and traverse them. Also make the value of visited array true for these nodes.
3. Repeat step 2 until the queue is empty.

Let us take node 1 as the starting node for traversal.

Step 1-

Insert starting node 1 into queue.
Traversed nodes = 1
visited[1]=true
front =0 rear=0 queue=1
Traversal = 1

Step 2-

Delete front element node 1 from queue and insert all its unvisited neighbours 2, 4, 5 into the queue .
Traversed nodes - 2, 4, 5
visited[2]=true, visited[4]=true, visited[5]=true
front =1 rear=3 queue = 2, 4, 5
Traversal = 1, 2, 4, 5

Step 3-
Delete front element node 2 from queue, traverse it's unvisited neighbour node 3 ,and insert it into the queue.
Traversed nodes - 3
visited[3] = true
front =2 rear = 4 queue = 4,5,3
Traversal = 1, 2, 4, 5, 3

Step 4-
Delete front element node 4 from queue, traverse it's unvisited neighbour node 7 and insert it into the queue.
Traversed nodes - 7
visited[7] = true
front = 3 rear = 5 queue = 5, 3, 7
Traversal = 1, 2, 4, 5, 3, 7

Step 5-
Delete front element node 5 from the queue, traverse it's unvisited neighbours nodes 6, 8 and insert them into the queue.
Traversed nodes – 6, 8
visited[6] = true, visited[8] = true
front = 4 rear = 7 queue = 3, 7, 6, 8
Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 6-
Delete front element node 3 from queue. It has no unvisited neighbours.
front = 5 rear = 7 queue = 7, 6, 8
Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 7-
Delete front elemnt node 7 from queue. It has no unvisited neighbours .
front = 6 rear = 7 queue = 6, 8
Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 8-
Delete front element node 6 from queue. It has no unvisited neighbours.
front = 7 rear = 7 queue = 8
Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 9-
Delete front element node 8 from queue, traverse it's unvisited neighbour node 9 and insert it into the queue.
Traversed nodes - 9
visited[9] = true
front = 8 rear = 8 queue = 9
Traversal = 1, 2, 4, 5, 3, 7, 6, 8, 9

Step 10-
Delete front element node 9 from queue. It has no unvisited neighbours
front = 8 rear = 7 queue = EMPTY
Traversal=1,2,4,5,3,7,6,8,9

Since front > rear, hence now queue is empty so we will stop our process
As we have seen this process can traverse only those nodes which are reachable from the starting node. If we want to traverse all the nodes then we will take the unvisited node as starting node and again start this process from starting node. This process will continue until all the nodes are traversed.

The function for Breadth first search is as-

```
bfs(int v)
{
        int i,front,rear;
        int que[20];
        front=rear= -1;
        printf("%d ",v);
        visited[v]=true;
        rear++;
        front++;
        que[rear]=v;
        while(front<=rear)
        {
                v=que[front];     /* delete from queue */
                front++;
                for(i=1;i<=n;i++)
                {
                        /* Check for adjacent unvisited nodes */
                        if( adj[v][i]==1 && visited[i]==false)
                        {
                                printf("%d ",i);
                                visited[i]=true;
                                rear++;
                                que[rear]=i;
                        }
                }
        }/*End of while*/
}/*End of bfs( )*/
```

Depth First Search-

In Depth first search technique also we take one node as a starting node. Then go to the path which is from starting node and visit all the nodes which are in that path. When we reach at the last node then we traverse another path starting from that node. If there is no path in the graph from the last node then it returns to the previous node in the path and

traverse another path and so on. This technique uses stack.

Let us take the same graph and starting node as node 1.
First, we will traverse node 1. Then we will traverse any node adjacent to node 1. Here we traverse node 2, then traverse node 3, which is adjacent to node 2, then traverse adjacent node 6. Now there is no node adjacent to node 6, means we have reached the end of the path or a dead end from where we can't go forward. So we will move backward. Till now the traversal is -
1 2 3 6
Now we reach node 3, see if there is any node adjacent to it, and not traversed yet. There is no such node so we will reach node 2 and we see that node 5 is adjacent to it and not traversed yet . So we will traverse node 5 and move along the path which starts at node 5. So we will traverse 8 and then 9 .Now there is no node adjacent to node 9 so we have reached the end of the path and now we will move backward. Till now the traversal is-
1 2 3 6 5 8 9
Now we reach node 8 and we see that there is no node adjacent to it and not traversed yet, so we reach node 5, here also we observe the same thing, so we reach node 6 here also there is no untraversed adjacent node, then we reach 3 and then 2. On reaching node 1 we see that node 4 is adjacent to it and has not been traversed. So we traverse it and move along a path which starts at node 4. So we traverse node 7. Now there is no untraversed node adjacent to node 7 so we have reached a dead end. We can't move forward but now we can't move backward also so we will stop our process. The traversal is as-
1 2 3 6 5 8 9 4 7

See the different traversals when we take different starting nodes.

Start Node	Traversal
1	1 2 3 6 5 8 9 4 7
2	2 3 6 5 8 9
3	3 6
4	4 7 5 6 8 9
5	5 6 8 9
6	6
7	7 5 6 8 9
8	8 9
9	9

Depth First Search through stack-

Take an array stack which will be used to keep the unvisited neighbours of the node. Take a boolean array visited which will have value true if the node has been visited and will have value false if the node has not been visited.

Initially stack is empty and top = -1
Initially visited[i]=false where i =1 to n, n is total number of nodes

Procedure-
1. Push starting node into the stack.
2. Pop an element from the stack, if it has not been traversed then traverse it, if it has already been traversed then just ignore it. After traversing make the value of visited array true for this node.
3. Now push all the unvisited adjacent nodes of the popped element on stack .Push the element even if it is already on the stack.
4. Repeat steps 3 and 4 until stack is empty.

Suppose the starting node for traversal is 1.

Step1-
Push node 1 into stack
top = 0 stack = 1

Step 2-
Pop node 1 from stack ,traverse it.
Traversed node - 1
visited[1] = true
Now push all the unvisited adjacent nodes 5,4,2 of the popped element on stack .
top = 2 stack = 5, 4, 2
Traversal=1

Step 3-
Pop the element node 2 from the stack, traverse it and push all its unvisited adjacent nodes 5,3 on the stack
Traversed node - 2
visited[2] = true
top = 3 stack = 5, 4, 5, 3
Traversal = 1, 2

Step 4-
Pop the element node 3 from the stack, traverse it and push its unvisited adjacent node 6 on the stack
Traversed node - 3
visited[3] = true
top = 3 stack = 5, 4, 5, 6
Traversal = 1, 2, 3

Step 5-
Pop the element node 6 from stack, traverse it. Node 6 has no adjacent nodes,so nothing is pushed
Traversed node – 6
visited [6] = true
top = 2 stack = 5, 4, 5
Traversal = 1, 2, 3, 6

Step 6-
Pop the element node 5 from stack, traverse it and push its unvisited adjacent node 8 on the stack,node 6 is its adjacent node but it has been visited so it is not pushed.
Traversed node - 5
visited[5] = true
top = 2 stack = 5, 4, 8
Traversal = 1, 2, 3, 6, 5

Step 7-
Pop the element node 8 from stack, traverse it and push its unvisited adjacent node 9 on the stack
Traversed node - 8
visited[8] = true
top = 2 stack = 5, 4, 9
Traversal = 1, 2, 3, 6, 5, 8

Step 8-
Pop the element 9 from the stack and traverse it ,9 has no adjacent nodes
visited[9]=true
top=1 stack=5, 4
Traversal=1,2,3,6,5,8,9

Step 9 -
Pop the element node 4 from stack, traverse it and push its unvisited adjacent node 7 on the stack
Traversed node - 4
visited[4] = true
top = 1 stack = 5, 7
Traversal = 1, 2, 3, 6, 5, 8, 9, 4

Step 10-
Pop the element node 7 from stack, traverse it ,it has no unvisited adjacent nodes.
Traversed node - 7
visited[7]=true
top = 0 stack = 5
Traversal = 1, 2, 3, 6, 5, 8, 9, 4, 7

Step 11-
Pop the element node 5 from stack, since visited[5] = true, so just ignore it.
top = -1 Stack = EMPTY

Since the stack is empty so we will stop our process.

The function for Depth First Search is as-

```
dfs(int v)
{
        int i,stack[MAX],top=-1,pop_v,j,t;
        int  ch;

        top++;
        stack[top]=v;s
        while (top>=0)
        {
                pop_v=stack[top];
                top--;                  /*pop from stack*/
                if( visited[pop_v]==false)
                {
                        printf("%d ",pop_v);
                        visited[pop_v]=true;
                }
                else
                        continue;

                for(i=n;i>=1;i--)
                {
                        if( adj[pop_v][i]==1 && visited[i]==false)
                        {
                                top++;   /* push all unvisited neighbours of pop_v *
                                stack[top]=i;
                        }/*End of if*/
                }/*End of for*/
        }/*End of while*/
*End of dfs( )*/
```