

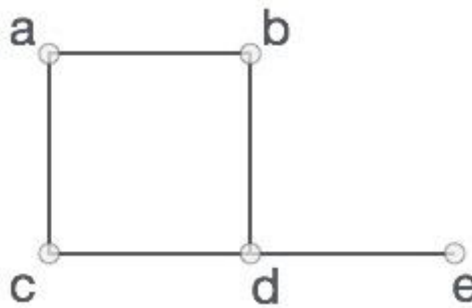
DATA STRUCTURES

(this is an additional material and not exhaustive study material)

Graphs: Definitions, Terminologies, Matrix and Adjacency List Representation Of Graphs, Elementary Graph operations, Traversal methods: Breadth First Search and Depth First Search.

A **graph** is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

Vertex – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, a to e are vertices.

Edge – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from a to b, d to e, and so on represents edges.

Adjacency – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, b is adjacent to a, c is not adjacent to b.

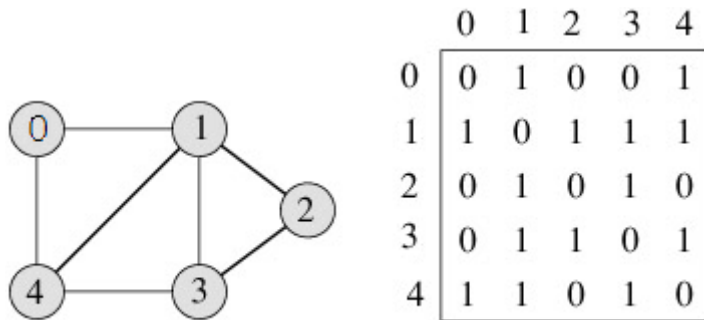
Path – Path represents a sequence of edges between the two vertices. For example, abde is a path.

Graph Representation

Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[i][j]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is

also used to represent weighted graphs. If $\text{adj}[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .



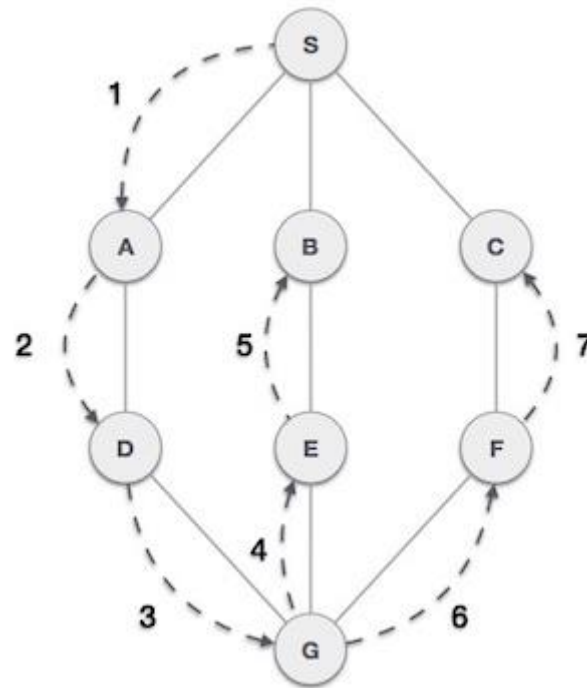
Applications of Graphs

- Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.
- Using GPS/Google Maps/Yahoo Maps, to find a route based on shortest route.
- 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.
- On eCommerce websites relationship graphs are used to show recommendations.

Two ways to traverse a Graph – DFS and BFS

DFS

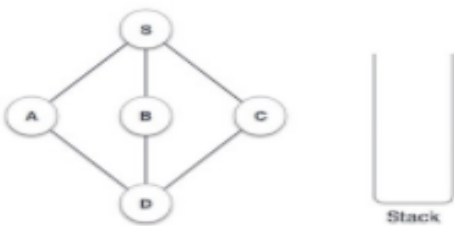
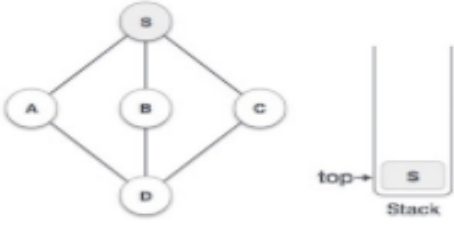
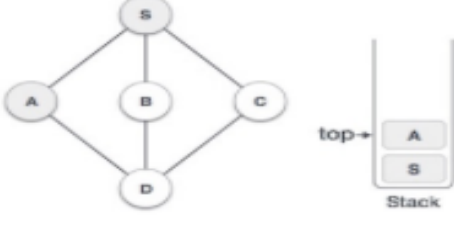
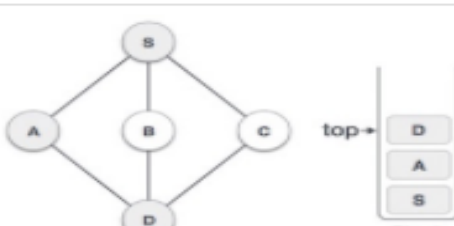
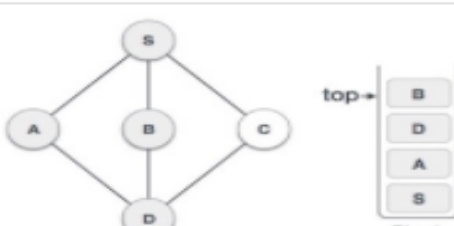
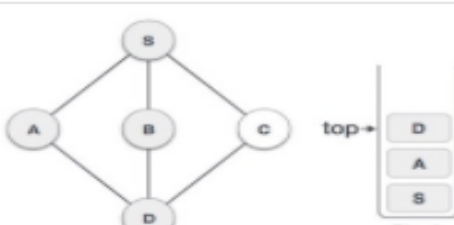
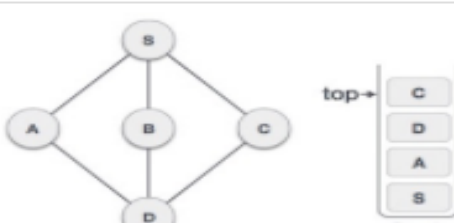
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to G. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

DATA STRUCTURES NOTES FOR II SEM B.Tech – 'B' SECTION

Step	Traversal	Description
1.		Initialize the stack.
2.		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A . Both S and D are adjacent to A but we are concerned for unvisited nodes only.
4.		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose B , mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7.		Only unvisited adjacent node is from D is C now. So we visit C , mark it as visited and put it onto the stack.

As C does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Depth First Search (DFS) Algorithm

```
n ← number of nodes
Initialize visited[ ] to false (0)
for(i=0;i<n;i++)
    visited[i] = 0;

void DFS(vertex i) [DFS starting from i]
{
    visited[i]=1;
    for each w adjacent to i
        if(!visited[w])
            DFS(w);
}
```

Depth First Search (DFS) Program in C [Adjacency Matrix]

```
#include<stdio.h>

void DFS(int);
int G[10][10],visited[10],n;    //n is no of vertices and graph is sorted in ar

void main()
{
    int i,j;
    printf("Enter number of vertices:");

    scanf("%d",&n);

    //read the adjacency matrix
    printf("\nEnter adjacency matrix of the graph:");

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&G[i][j]);

    //visited is initialized to zero
    for(i=0;i<n;i++)
        visited[i]=0;

    DFS(0);
}

void DFS(int i)
{
    int j;
    printf("\n%d",i);
    visited[i]=1;

    for(j=0;j<n;j++)
        if(!visited[j]&&G[i][j]==1)
            DFS(j);
}
```

BFS algorithm

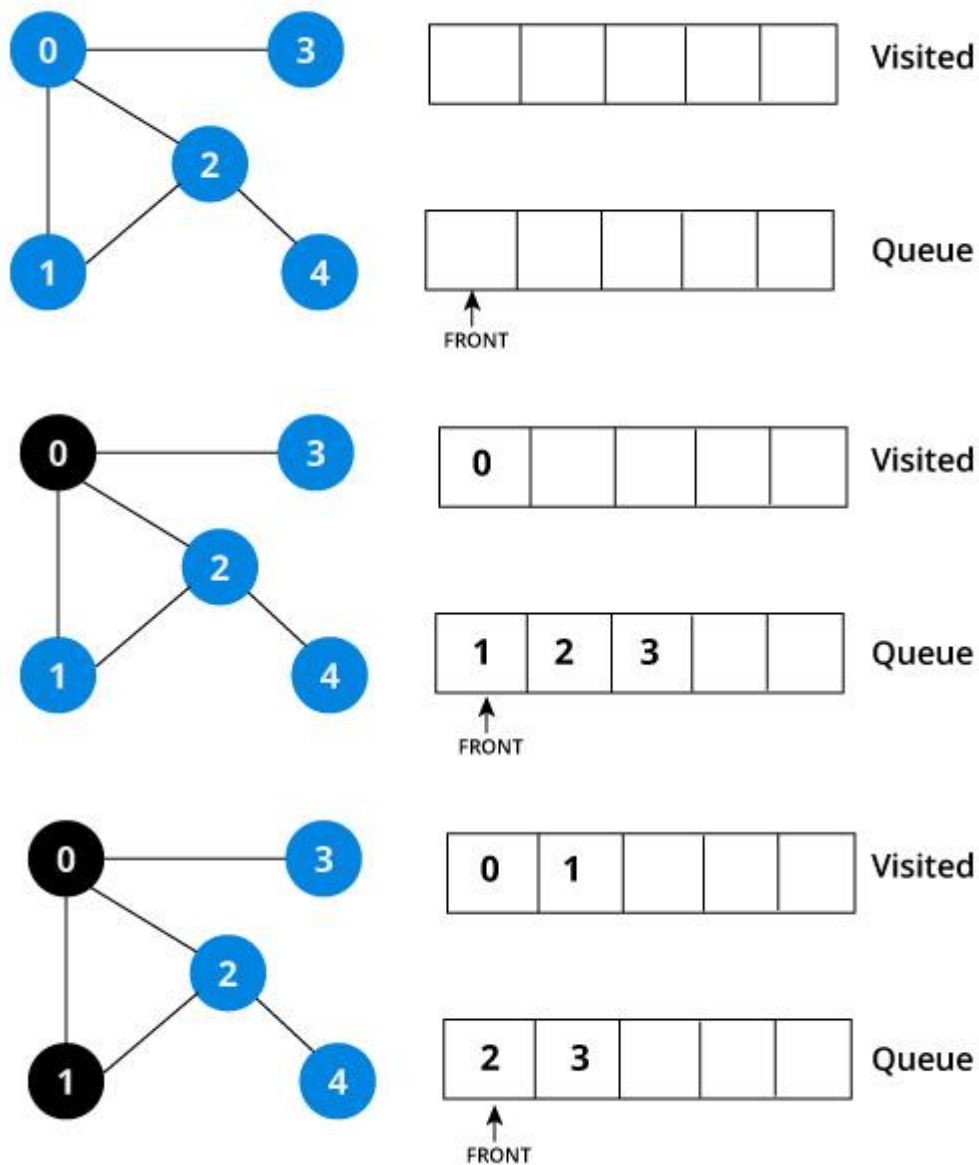
A standard BFS implementation puts each vertex of the graph into one of two categories:

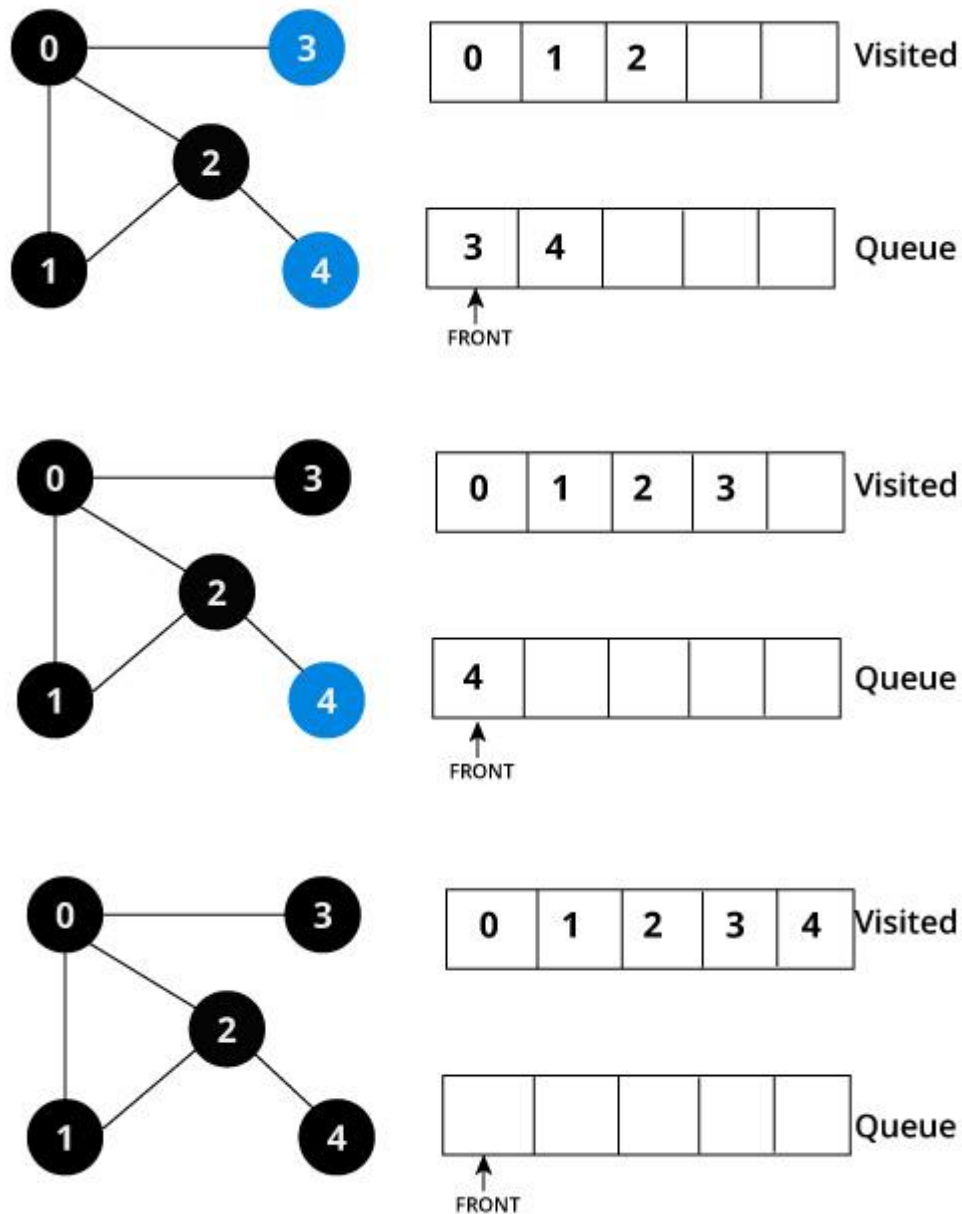
1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.





BFS Algorithm

```

create a queue Q
mark v as visited and put v into Q
while Q is non-empty
    remove the head u of Q
    mark and enqueue all (unvisited) neighbours of u
  
```

Program

```

int a[20][20], q[20], visited[20], n, i, j, f = 0, r = -1;
void bfs(int v) {
    for(i = 1; i <= n; i++)
        if(a[v][i] && !visited[i])
            q[++r] = i;
    if(f <= r) {
        visited[q[f]] = 1;
        bfs(q[f++]);
    }
}

void main() {
    int v;
    printf("\n Enter the number of vertices:");
    scanf("%d", &n);
    printf("\n Enter graph data in matrix form:\n");
    for(i=1; i<=n; i++)
    {
        for(j=1;j<=n;j++)
            scanf("%d", &a[i][j]);
    }
    for(i=1; i <= n; i++)
    {
        q[i] = 0;
        visited[i] = 0;
    }
    printf("\n Enter the starting vertex:");
    scanf("%d", &v);
    bfs(v);
    printf("\n The node which are reachable are:\n");

    for(i=1; i <= n; i++) {
        if(visited[i])
            printf("%d\t", i);
        else {

```



```

        printf("\n Bfs is not possible. Not all nodes are reachable");
        break;
    }
}
}

```

Difference between BFS and DFS

S. No.	Breadth First Search (BFS)	Depth First Search (DFS)
1.	BFS visit nodes level by level in Graph.	DFS visit nodes of graph depth wise . It visits nodes until reach a leaf or a node which doesn't have non-visited nodes.
2.	A node is fully explored before any other can begin.	Exploration of a node is suspended as soon as another unexplored is found.
3.	Uses Queue data structure to store Un-explored nodes.	Uses Stack data structure to store Un-explored nodes.
4.	BFS is slower and require more memory.	DFS is faster and require less memory.
5.	Some Applications: <ul style="list-style-type: none"> Finding all connected components in a graph. Finding the shortest path between two nodes. Finding all nodes within one connected component. Testing a graph for bipartiteness. 	Some Applications: <ul style="list-style-type: none"> Topological Sorting. Finding connected components. Solving puzzles such as maze. Finding strongly connected components. Finding articulation points (cut vertices) of the graph.