- Representing relationships between components in electronic circuits
- Transportation networks: Highway network, Flight network
- Computer networks: Local area network, Internet, Web
- Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases

9.4 Graph Representation

As in other ADTs, to manipulate graphs we need to represent them in some useful form. Basically, there are three ways of doing this:

- Adjacency Matrix
- Adjacency List
- Adjacency Set

Adjacency Matrix

Graph Declaration for Adjacency Matrix

First, let us look at the components of the graph data structure. To represent graphs, we need the number of vertices, the number of edges and also their interconnections. So, the graph can be declared as:

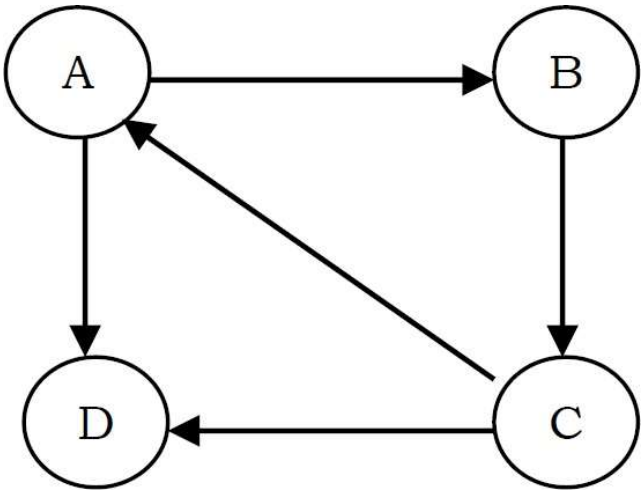
```
struct Graph {
 int V;
 int E;
 int **Adj; //Since we need two dimensional matrix
};
```

Description

In this method, we use a matrix with size $V \times V$. The values of matrix are boolean. Let us assume the matrix is *Adj*. The value *Adj*[*u*, *v*] is set to 1 if there is an edge from vertex u to vertex v and 0 otherwise.

In the matrix, each edge is represented by two bits for undirected graphs. That means, an edge from **u** to **v** is represented by 1 value in both *Adj*[**u**,**v**] and *Adj*[*u*,*v*]. To save time, we can process only half of this symmetric matrix. Also, we can assume that there is an “edge” from each vertex to itself. So, *Adj*[u, u] is set to 1 for all vertices.

If the graph is a directed graph then we need to mark only one entry in the adjacency matrix. As an example, consider the directed graph below.



The adjacency matrix for this graph can be given as:

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

Now, let us concentrate on the implementation. To read a graph, one way is to first read the vertex names and then read pairs of vertex names (edges). The code below reads an undirected graph.

```
//This code creates a graph with adj matrix representation
struct Graph *adjMatrixOfGraph() {
    int i, u, v;
    struct Graph *G = (struct Graph *) malloc(sizeof(struct Graph));
    if(!G) {
        printf("Memory Error");
        return;
    }
    scanf("Number of Vertices: %d, Number of Edges:%d", &G→V, &G→E);
    G→Adj = malloc(sizeof(G→V  * G→V));
    for(u = 0; u < G→V; u++)
            for(v = 0; v < G→V; v++)
                    G→Adj[v][v] = 0;
    for(i = 0;  i < G→E; i++) {
        //Read an edge
        scanf("Reading Edge: %d %d", &u, &v);
        //For undirected graphs set both the bits
        G→ Adj[u][v] = 1;
        G→ Adj[v][u] = 1;
    }
    return G;
}
```

The adjacency matrix representation is good if the graphs are dense. The matrix requires $O(V^2)$ bits of storage and $O(V^2)$ time for initialization. If the number of edges is proportional to V^2, then there is no problem because V^2 steps are required to read the edges. If the graph is sparse, the initialization of the matrix dominates the running time of the algorithm as it takes takes $O(V^2)$.

Adjacency List

Graph Declaration for Adjacency List

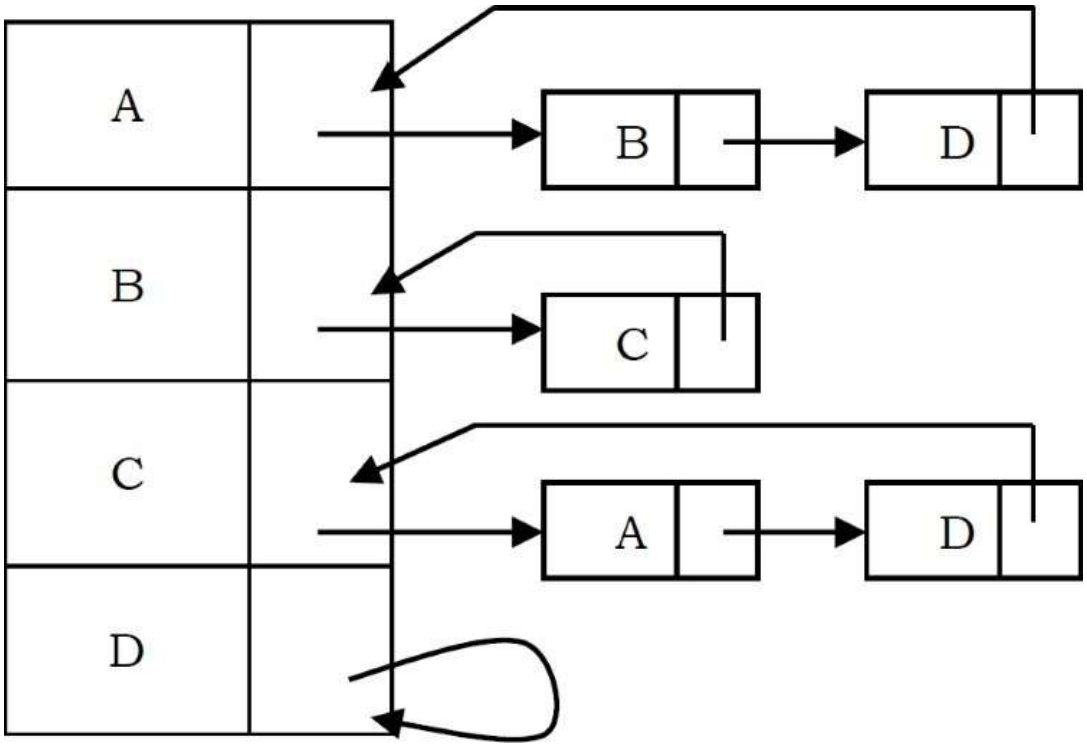
In this representation all the vertices connected to a vertex *v* are listed on an adjacency list for that vertex *v*. This can be easily implemented with linked lists. That means, for each vertex *v* we use a linked list and list nodes represents the connections between *v* and other vertices to which *v* has an edge.

The total number of linked lists is equal to the number of vertices in the graph. The graph ADT can be declared as:

```
struct Graph {
    int V;
    int E;
    int *Adj; //head pointers to linked list
};
```

Description

Considering the same example as that of the adjacency matrix, the adjacency list representation can be given as:



Since vertex A has an edge for B and D, we have added them in the adjacency list for A. The same is the case with other vertices as well.

```
//Nodes of the Linked List
struct ListNode {
    int vertexNumber;
    struct ListNode *next;
}

//This code creates a graph with adj list representation
struct Graph *adjListOfGraph() {
    int i, x, y;
    struct ListNode *temp;
    struct Graph *G = (struct Graph *) malloc(sizeof(struct Graph));
    if(!G) {
        printf("Memory Error");
        return;
    }
    scanf("Number of Vertices: %d, Number of Edges:%d", &G→V, &G→E);
    G→Adj = malloc(G→V * sizeof(struct ListNode));

    for(i = 0; i  < G→V; i++) {
        G→Adj[i] = (struct ListNode *) malloc(sizeof(struct ListNode));
        G→Adj[i]→vertexNumber = i;
        G→Adj[i]→next = G→ Adj[i];
    }
    for(i = 0;  i < E; i++) {
        //Read an edge
        scanf("Reading Edge: %d %d", &x, &y);
        temp = (struct ListNode *) malloc(struct ListNode);
        temp→vertexNumber = y;
        temp→next = G→Adj[x];
        G→ Adj[x]→next = temp;
        temp = (struct ListNode *) malloc(struct ListNode);
        temp→vertexNumber = y;
        temp→next = G→ Adj[y];
        G→Adj[y]→ next= temp;
    }

    retutn G;
}
```

For this representation, the order of edges in the input is *important*. This is because they determine the order of the vertices on the adjacency lists. The same graph can be represented in many different ways in an adjacency list. The order in which edges appear on the adjacency list affects the order in which edges are processed by algorithms.

Disadvantages of Adjacency Lists

Using adjacency list representation we cannot perform some operations efficiently. As an example, consider the case of deleting a node. . In adjacency list representation, it is not enugh if we simply delete a node from the list representation, if we delete a node from the adjacency list then that is enough. For each node on the adjacency list of that node specifies another vertex. We need to search other nodes linked list also for deleting it. This problem can be solved by linking the two list nodes that correspond to a particular edge and making the adjacency lists doubly linked. But all these extra links are risky to process.

Adjacency Set

It is very much similar to adjacency list but instead of using Linked lists, Disjoint Sets [Union-Find] are used. For more details refer to the *Disjoint Sets ADT* chapter.

Comparison of Graph Representations

Directed and undirected graphs are represented with the same structures. For directed graphs, everything is the same, except that each edge is represented just once. An edge from x to y is represented by a 1 value in $Agj[x][y]$ in the adjacency matrix, or by adding y on x's adjacency list. For weighted graphs, everything is the same, except fill the adjacency matrix with weights instead of boolean values.

Representation	Space	Checking edge between v and w?	Iterate over edges incident to v?
List of edges	E	E	E
Adj Matrix	V^2	1	V
Adj List	$E+V$	$Degree(v)$	$Degree(v)$
Adj Set	$E+V$	$log(Degree(v))$	$Degree(v)$

9.5 Graph Traversals

To solve problems on graphs, we need a mechanism for traversing the graphs. Graph traversal algorithms are also called *graph search* algorithms. Like trees traversal algorithms (Inorder, Preorder, Postorder and Level-Order traversals), graph search algorithms can be thought of as starting at some source vertex in a graph and "searching" the graph by going through the edges and marking the vertices. Now, we will discuss two such algorithms for traversing the graphs.

- Depth First Search [DFS]
- Breadth First Search [BFS]

Depth First Search [DFS]

DFS algorithm works in a manner similar to preorder traversal of the trees. Like preorder traversal, internally this algorithm also uses stack.

Let us consider the following example. Suppose a person is trapped inside a maze. To come out from that maze, the person visits each path and each intersection (in the worst case). Let us say the person uses two colors of paint to mark the intersections already passed. When discovering a new intersection, it is marked grey, and he continues to go deeper.

After reaching a "dead end" the person knows that there is no more unexplored path from the grey intersection, which now is completed, and he marks it with black. This "dead end" is either an intersection which has already been marked grey or black, or simply a path that does not lead to an intersection.

The intersections of the maze are the vertices and the paths between the intersections are the edges of the graph. The process of returning from the "dead end" is called *backtracking*. We are trying to go away from the starting vertex into the graph as deep as possible, until we have to backtrack to the preceding grey vertex. In DFS algorithm, we encounter the following types of edges.

Tree edge: encounter new vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees

For most algorithms boolean classification, unvisited/visited is enough (for three color implementation refer to problems section). That means, for some problems we need to use three colors, but for our discussion two colors are enough.

false ⟶ Vertex is unvisited

true ⟶ Vertex is visited

Initially all vertices are marked unvisited (false). The DFS algorithm starts at a vertex *u* in the graph. By starting at vertex *u* it considers the edges from *u* to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex *u*. If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That means the new vertex becomes the current vertex. Follow this process until we reach the dead-end. At this point start *backtracking*.

The process terminates when backtracking leads back to the start vertex. The algorithm based on this mechanism is given below: assume Visited[] is a global array.

```
int Visited[G→V];
void DFS(struct Graph *G, int u) {
    Visited[u] = 1;
    for( int v = 0; v < G→V; v++ ) {

        /* For example, if the adjacency matrix is used for representing the
           graph, then the condition to be used for finding unvisited adjacent
           vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */

        for each unvisited adjacent node v of u {
            DFS(G, v);
        }
    }
}
void DFSTraversal(struct Graph *G) {
    for (int i = 0; i< G→V;i++)
        Visited[i]=0;

    //This loop is required if the graph has more than one component
    for (int i = 0; i< G→V;i++)
        if(!Visited[i])
            DFS(G, i);
}
```

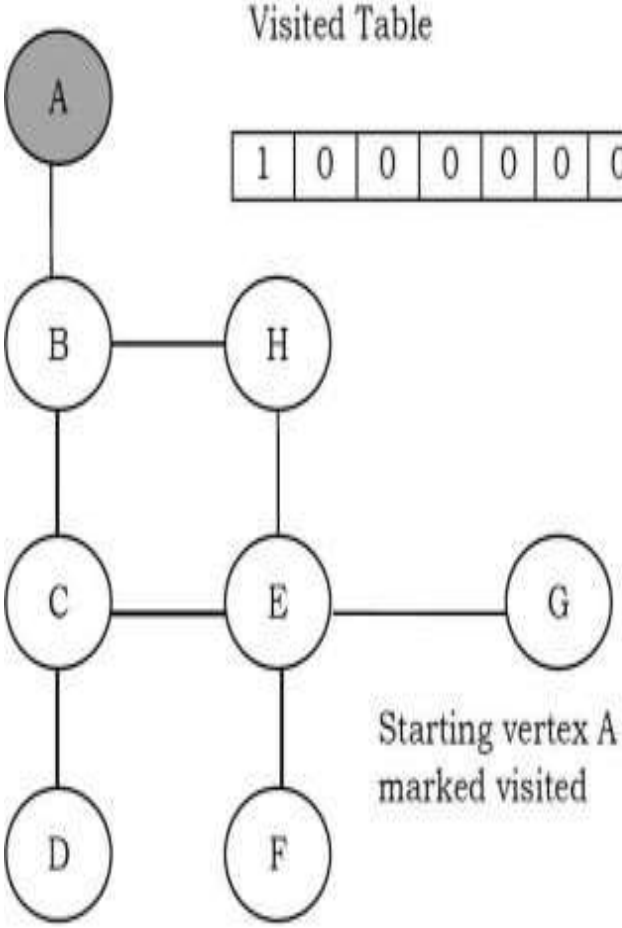
As an example, consider the following graph. We can see that sometimes an edge leads to an

already discovered vertex. These edges are called *back edges*, and the other edges are called *tree edges* because deleting the back edges from the graph generates a tree.

The final generated tree is called the DFS tree and the order in which the vertices are processed is called *DFS numbers* of the vertices. In the graph below, the gray color indicates that the vertex is visited (there is no other significance). We need to see when the Visited table is updated.

Visited Table

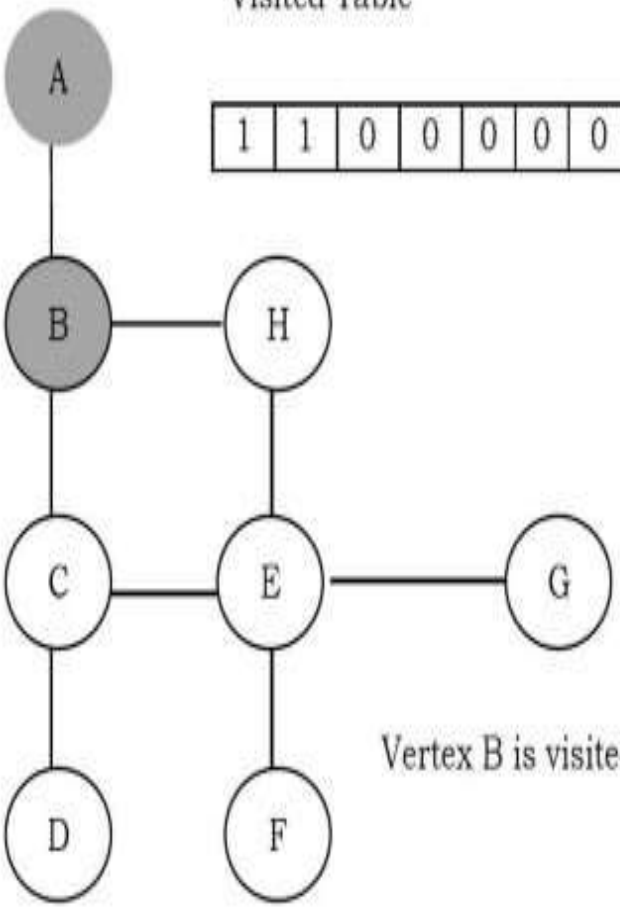
1	0	0	0	0	0	0	0



Starting vertex A is marked visited

Visited Table

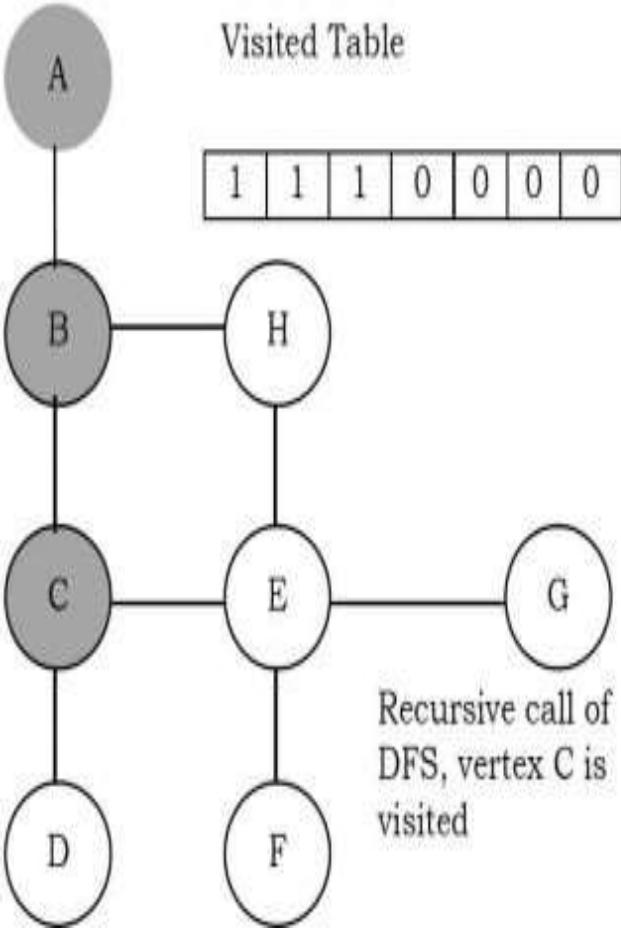
1	1	0	0	0	0	0	0



Vertex B is visited

Visited Table

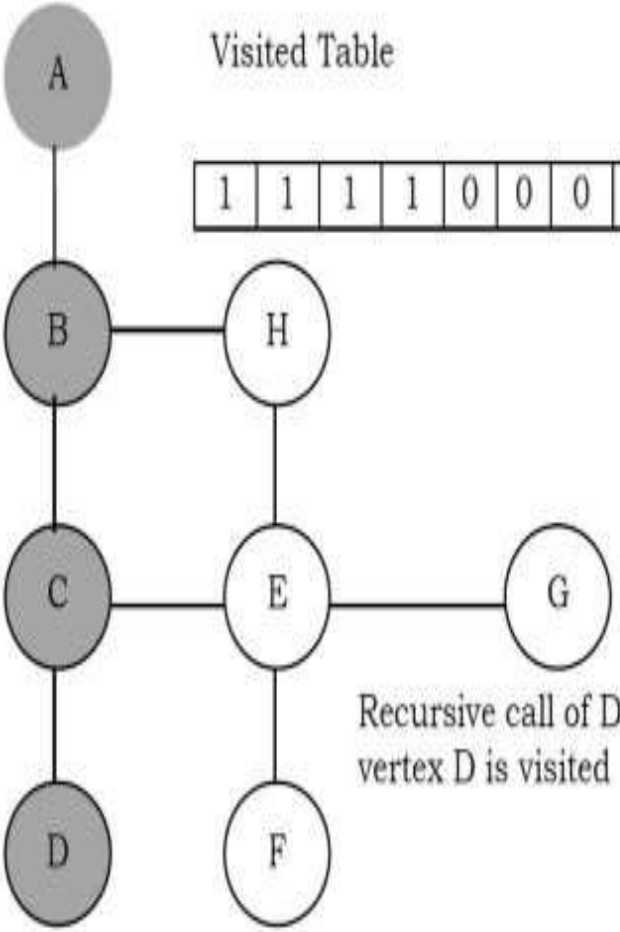
1	1	1	0	0	0	0	0



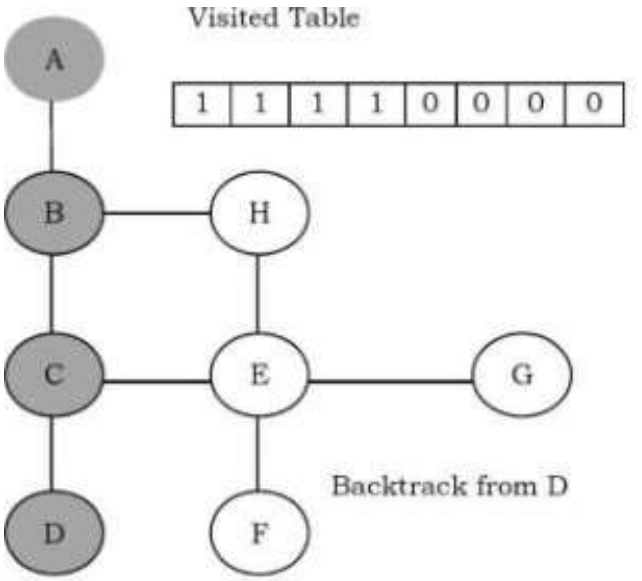
Recursive call of DFS, vertex C is visited

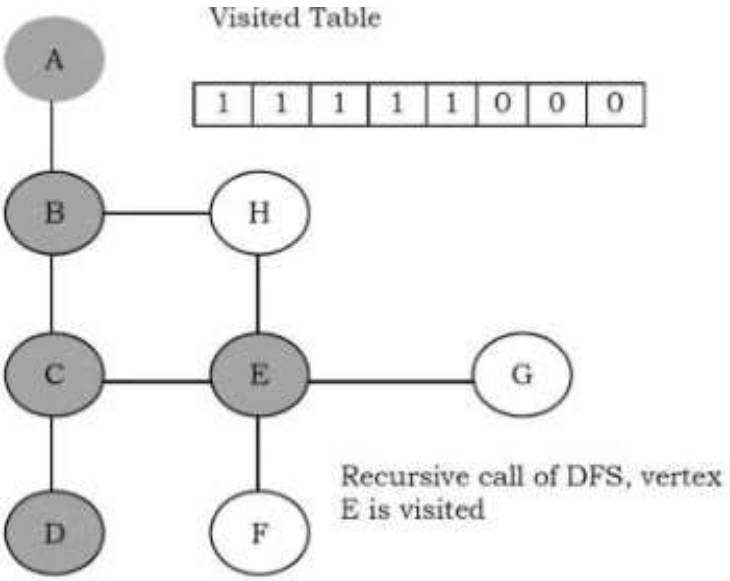
Visited Table

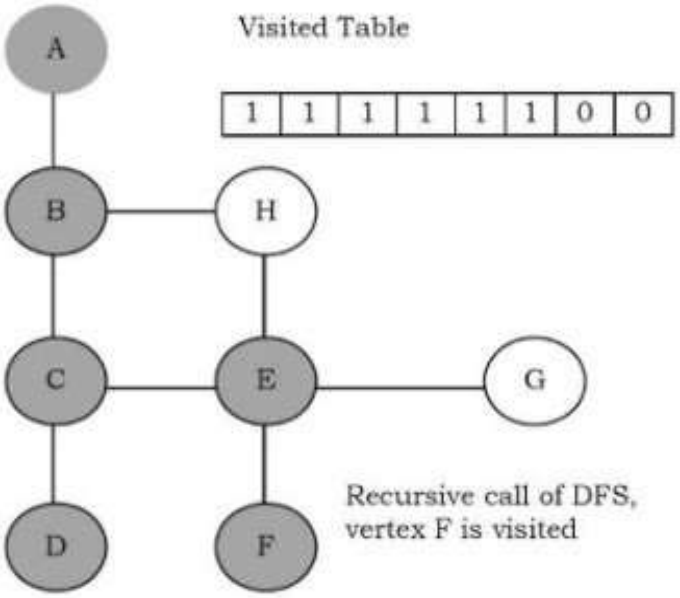
1	1	1	1	0	0	0	0

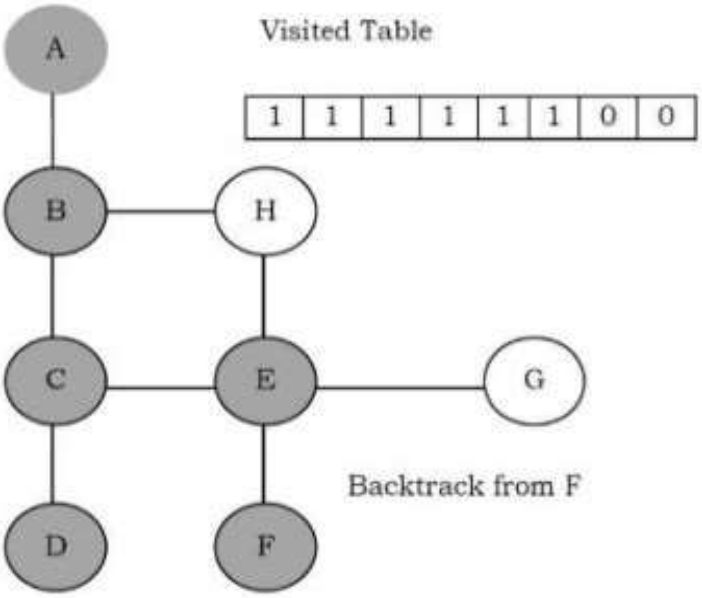


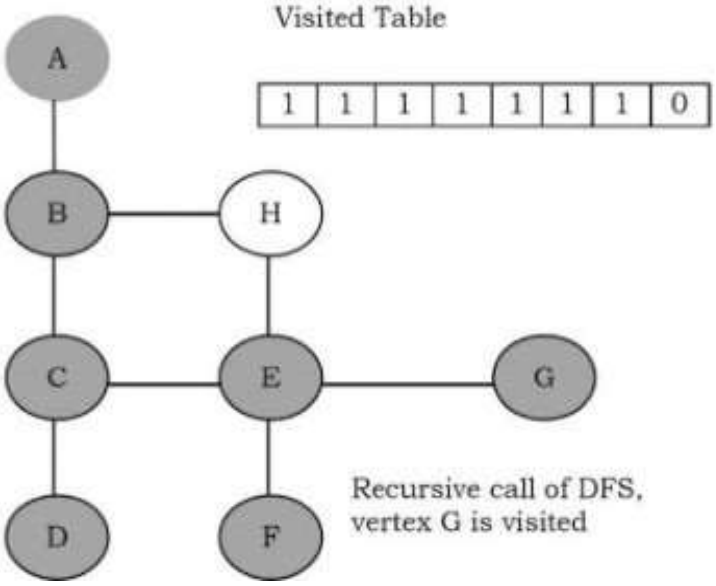
Recursive call of DFS, vertex D is visited

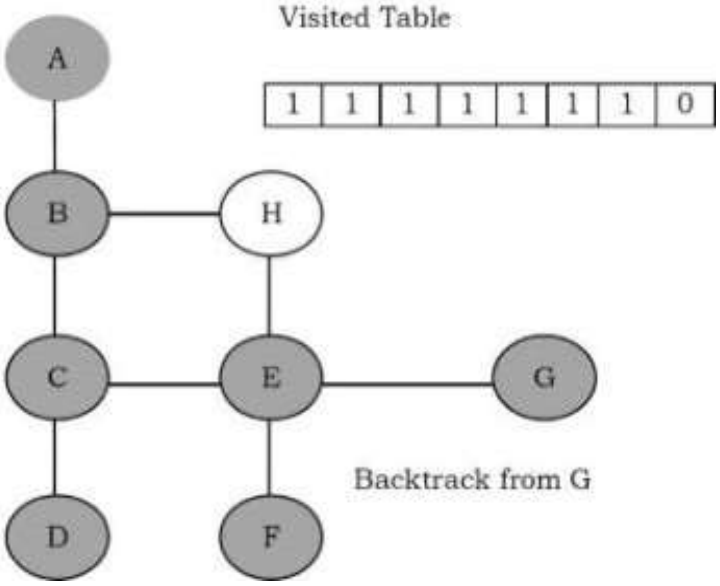

Visited Table
1 1 1 1 0 0 0 0
A
B
H
C
E
G
D
F
Backtrack from D

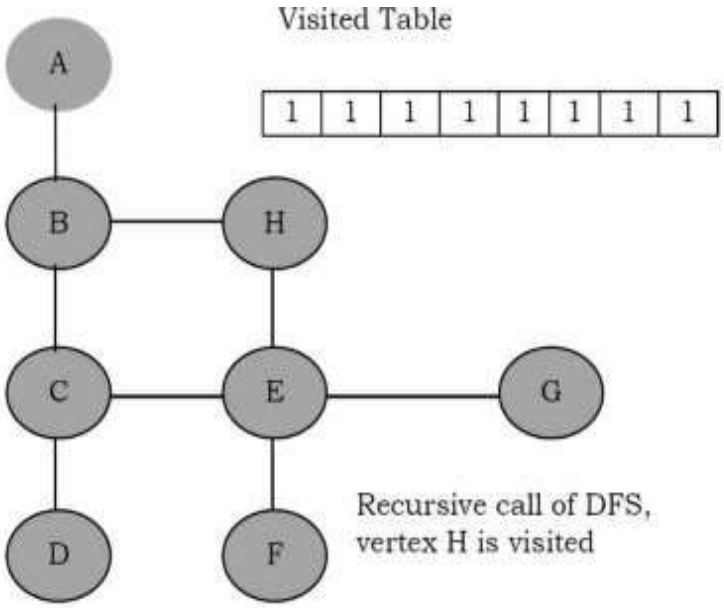

Visited Table
1 1 1 1 1 0 0 0
A
B
H
C
E
G
D
F
Recursive call of DFS, vertex E is visited

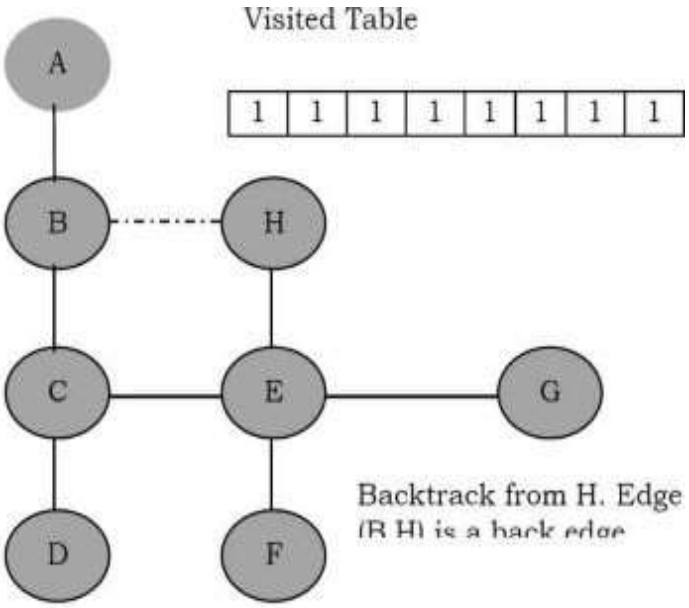

Visited Table
1 1 1 1 1 1 0 0
A
B
H
C
E
G
D
F
Recursive call of DFS, vertex F is visited

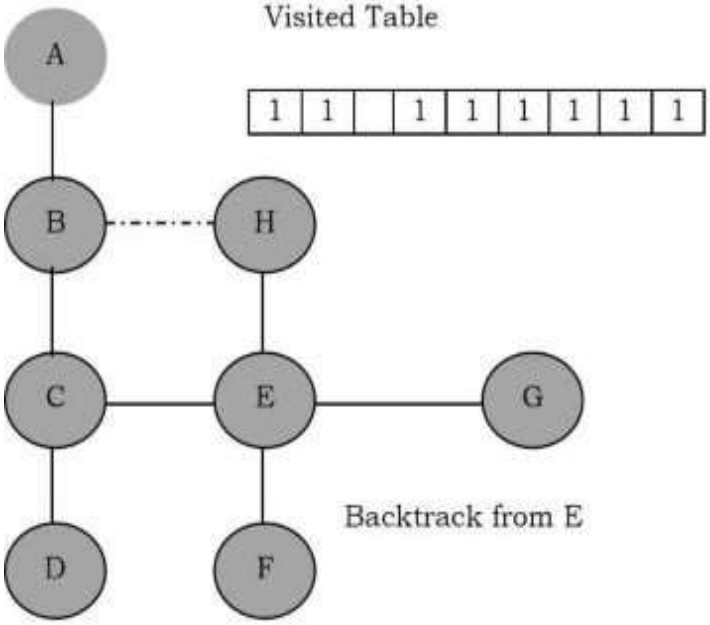

Visited Table
1 1 1 1 1 1 0 0
A
B
H
C
E
G
D
F
Backtrack from F

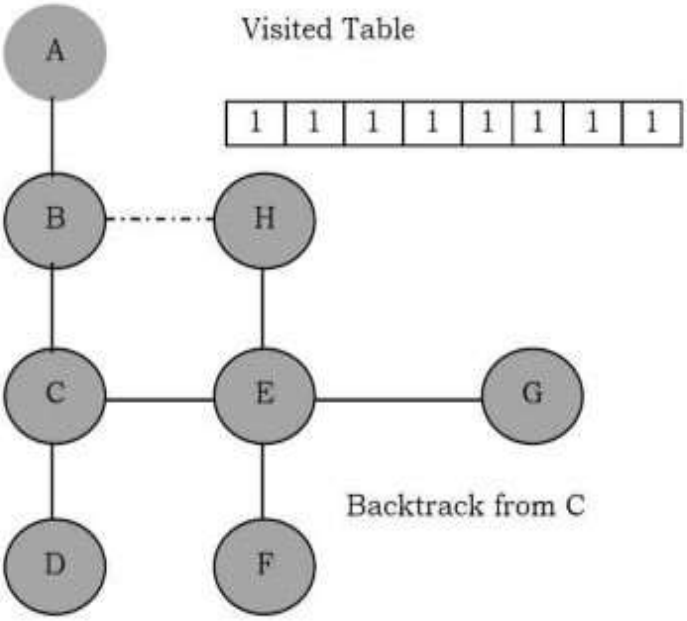

Visited Table
1 1 1 1 1 1 1 0
A
B
H
C
E
G
D
F
Recursive call of DFS, vertex G is visited

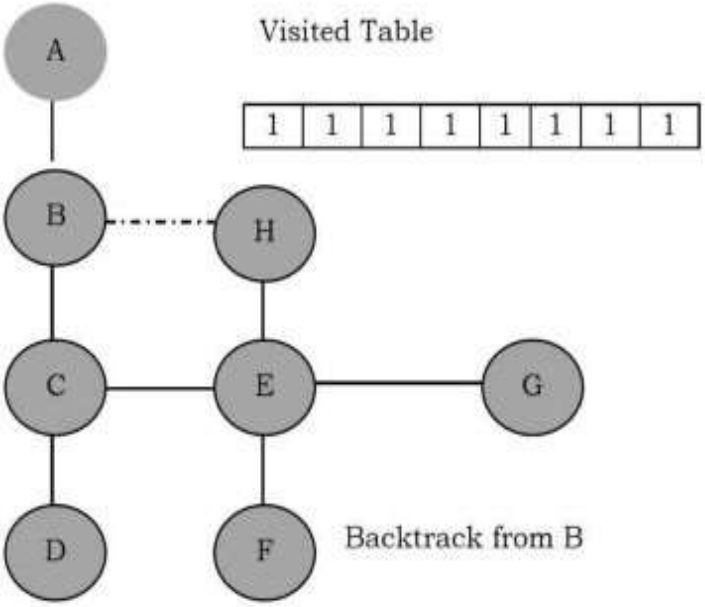

Visited Table
1 1 1 1 1 1 1 0
A
B
H
C
E
G
D
F
Backtrack from G

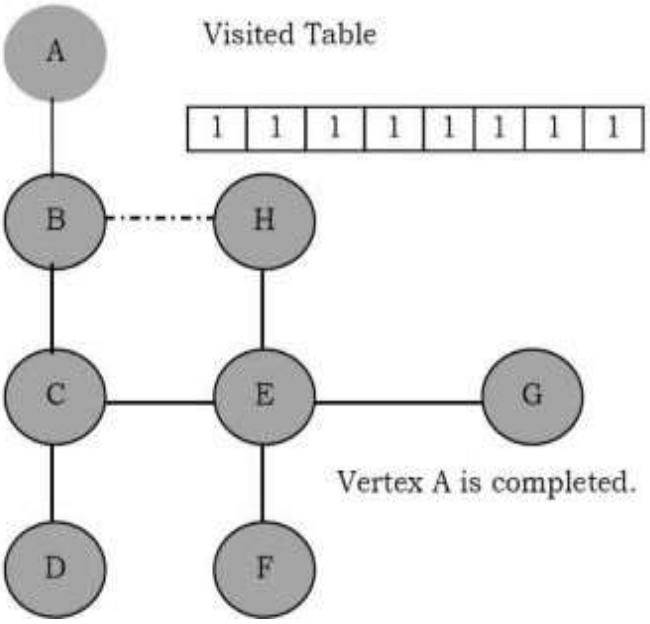

Visited Table
1 1 1 1 1 1 1 1
A
B
H
C
E
G
D
F
Recursive call of DFS,
vertex H is visited


Visited Table
1 1 1 1 1 1 1 1
A
B
H
C
E
G
D
F
Backtrack from H. Edge


Visited Table
1 1 1 1 1 1 1 1
A
B
H
C
E
G
D
F
Backtrack from E


Visited Table
1 1 1 1 1 1 1 1
A
B
H
C
E
G
D
F
Backtrack from C


Visited Table
1 1 1 1 1 1 1 1
A
B
H
C
E
G
D
F
Backtrack from B


Visited Table
1 1 1 1 1 1 1 1
A
B
H
C
E
G
D
F
Vertex A is completed.

From the above diagrams, it can be seen that the DFS traversal creates a tree (without back edges) and we call such tree a *DFS tree*. The above algorithm works even if the given graph has connected components.

The time complexity of DFS is O(V + E), if we use adjacency lists for representing the graphs. This is because we are starting at a vertex and processing the adjacent nodes only if they are not visited. Similarly, if an adjacency matrix is used for a graph representation, then all edges adjacent to a vertex can't be found efficiently, and this gives O(V^2) complexity.

Applications of DFS

- Topological sorting
- Finding connected components
- Finding articulation points (cut vertices) of the graph
- Finding strongly connected components
- Solving puzzles such as mazes

For algorithms refer to *Problems Section*.

Breadth First Search [BFS]

The BFS algorithm works similar to *level – order* traversal of the trees. Like *level – order* traversal, BFS also uses queues. In fact, *level – order* traversal got inspired from BFS. BFS works level by level. Initially, BFS starts at a given vertex, which is at level 0. In the first stage it visits all vertices at level 1 (that means, vertices whose distance is 1 from the start vertex of the graph). In the second stage, it visits all vertices at the second level. These new vertices are the ones which are adjacent to level 1 vertices.

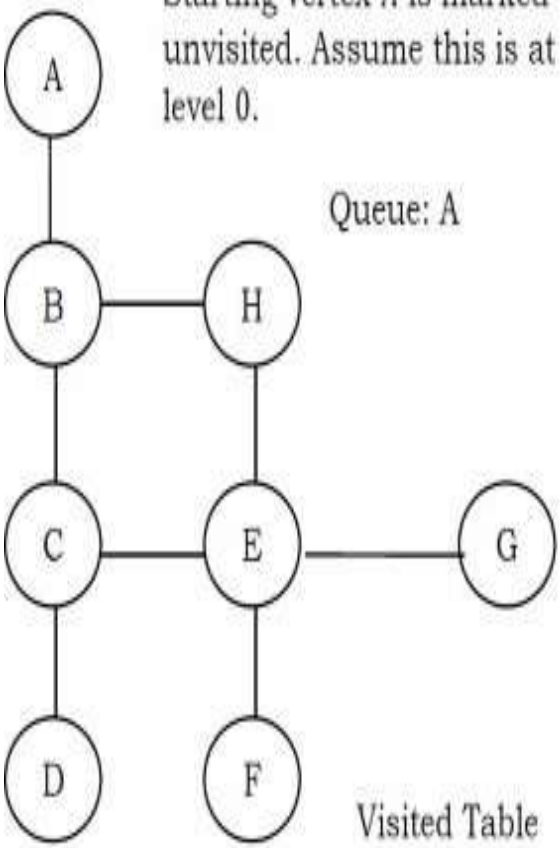
BFS continues this process until all the levels of the graph are completed. Generally *queue* data structure is used for storing the vertices of a level.

As similar to DFS, assume that initially all vertices are marked *unvisited* (*false*). Vertices that have been processed and removed from the queue are marked *visited* (*true*). We use a queue to represent the visited set as it will keep the vertices in the order of when they were first visited. The implementation for the above discussion can be given as:

```
void BFS(struct Graph *G, int u) {
    int v;
    struct Queue *Q = CreateQueue();

    EnQueue(Q, u);

   while(!IsEmptyQueue(Q)) {
        u = DeQueue(Q);

        Process u; //For example, print

        Visited[s]=1;

        /* For example, if the adjacency matrix is used for representing the  graph,
        then the condition be used for finding unvisited adjacent vertex of u  is:
        if( !Visited[v] && G→Adj[u][v] ) */
        for each unvisited adjacent node v of u {
            EnQueue(Q, v);
        }
    }
}

void BFSTraversal(struct Graph *G) {
    for (int i = 0; i< G→V;i++)
        Visited[i]=0;

    //This loop is required if the graph has more than one component
   for (int i = 0; i< G→V;i++)
        if(!Visited[i])
            BFS(G. i):
}
```

As an example, let us consider the same graph as that of the DFS example. The BFS traversal can be shown as:

Starting vertex *A* is marked unvisited. Assume this is at level 0.

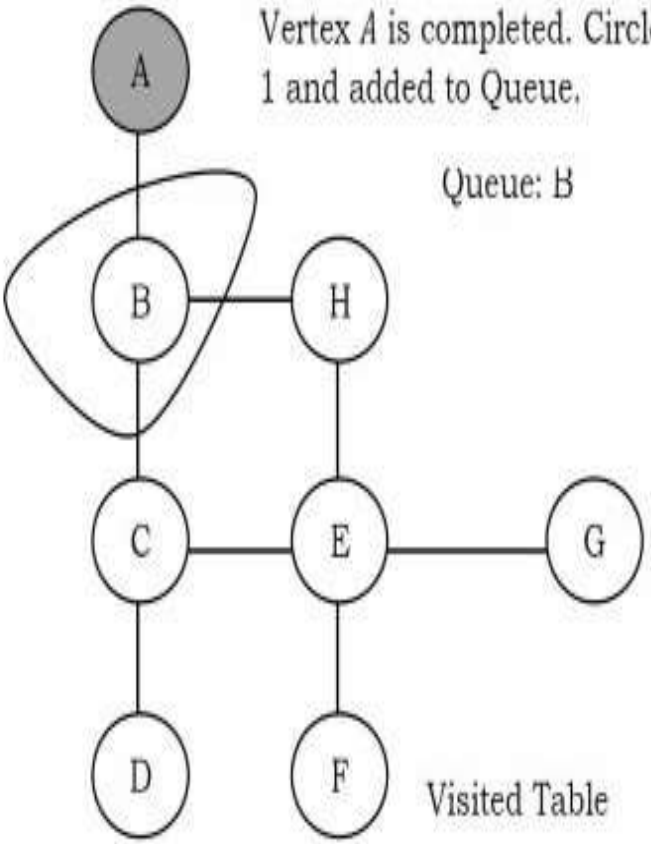


Queue: A

Visited Table

0		0		0	0	0	0	0

Vertex *A* is completed. Circled part is level 1 and added to Queue.



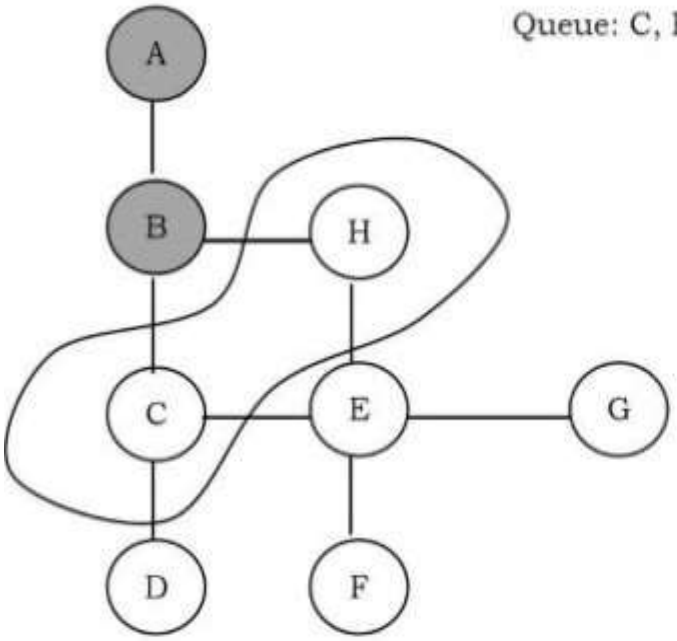
Queue: B

Visited Table

1	0	0	0	0	0	0

B is completed. Selected part is level 2 (add to Queue).

Queue: C, H

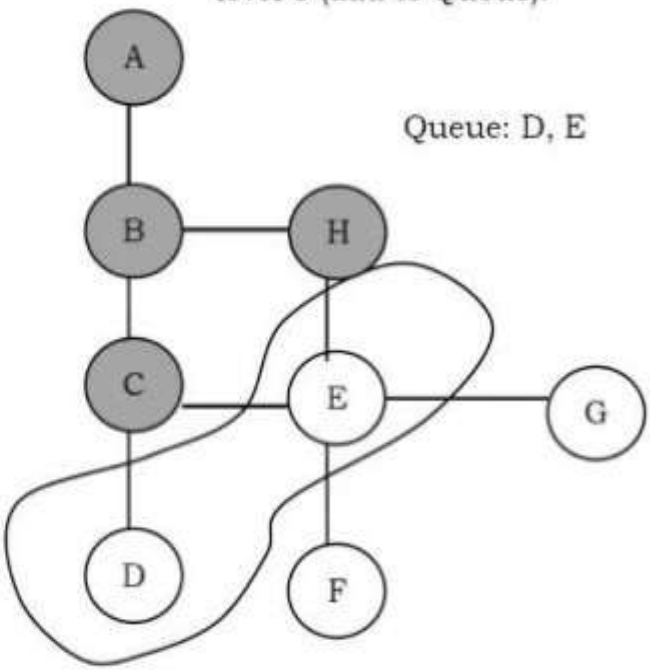


Visited Table

1	1	0	0	0	0	0

Vertices C and H are completed. Circled part is level 3 (add to Queue).

Queue: D, E

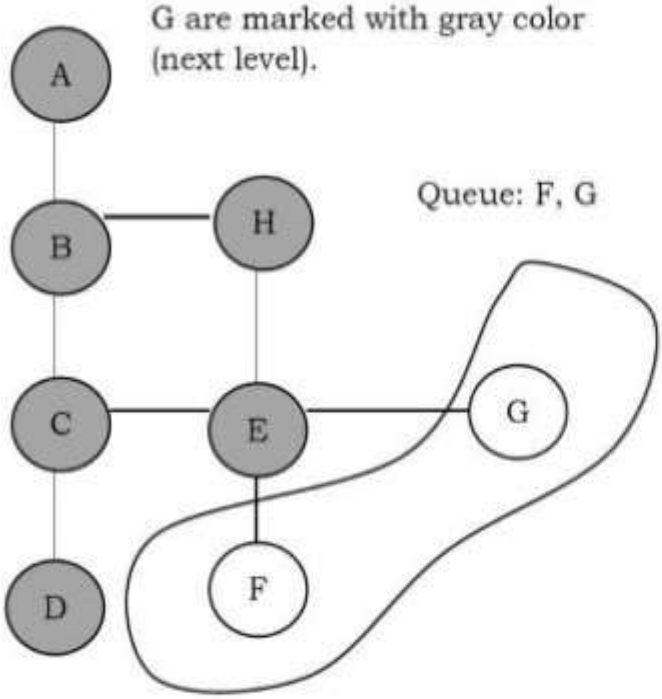


Visited Table

1	1	0	0	0	0	1

D and E are completed. F and G are marked with gray color (next level).

Queue: F, G

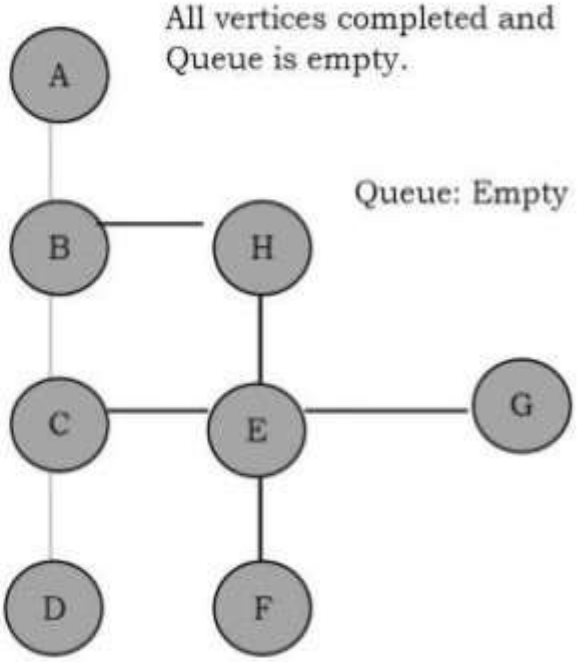


Visited Table

1	1	1	1	0	0	1

All vertices completed and Queue is empty.

Queue: Empty



Visited Table

1	1	1	1	1	1	1

Time complexity of BFS is O(V + E), if we use adjacency lists for representing the graphs, and O(V^2) for adjacency matrix representation.

Applications of BFS

- Finding all connected components in a graph
- Finding all nodes within one connected component
- Finding the shortest path between two nodes
- Testing a graph for bipartiteness

Comparing DFS and BFS

Comparing BFS and DFS, the big advantage of DFS is that it has much lower memory requirements than BFS because it's not required to store all of the child pointers at each level. Depending on the data and what we are looking for, either DFS or BFS can be advantageous. For example, in a family tree if we are looking for someone who's still alive and if we assume that person would be at the bottom of the tree, then DFS is a better choice. BFS would take a very long time to reach that last level.

The DFS algorithm finds the goal faster. Now, if we were looking for a family member who died a very long time ago, then that person would be closer to the top of the tree. In this case, BFS finds faster than DFS. So, the advantages of either vary depending on the data and what we are looking for.

DFS is related to preorder traversal of a tree. Like *preorder* traversal, DFS visits each node before its children. The BFS algorithm works similar to *level – order* traversal of the trees.

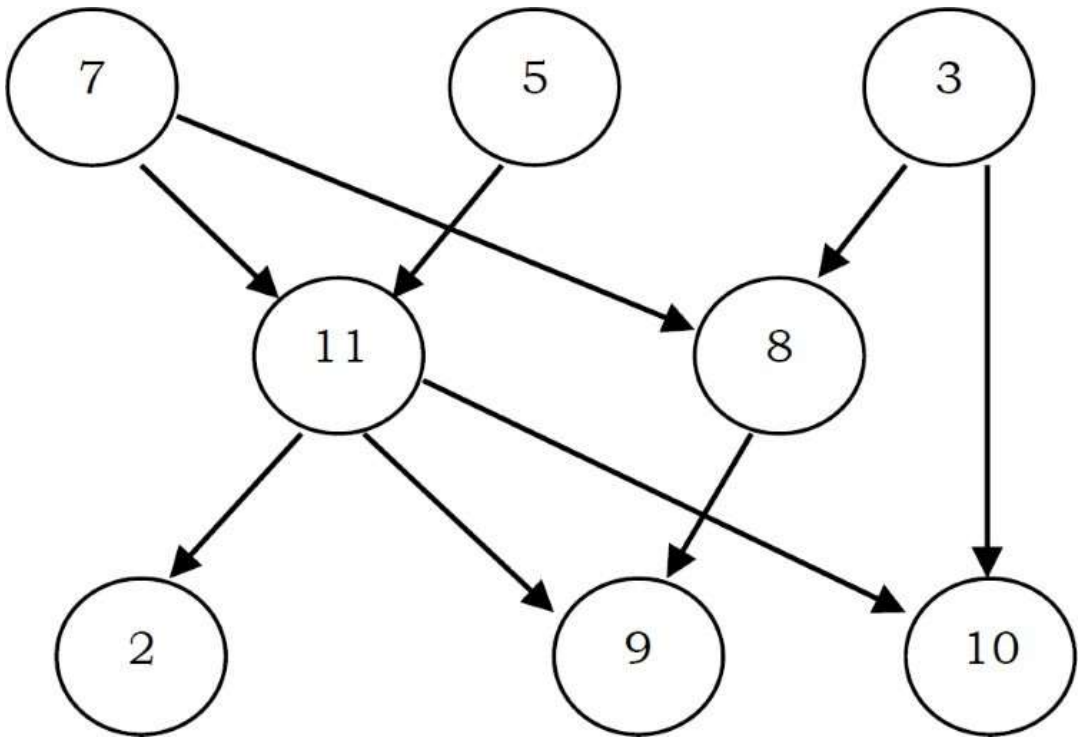
If someone asks whether DFS is better or BFS is better, the answer depends on the type of the problem that we are trying to solve. BFS visits each level one at a time, and if we know the solution we are searching for is at a low depth, then BFS is good. DFS is a better choice if the solution is at maximum depth. The below table shows the differences between DFS and BFS in terms of their applications.

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	Yes	Yes
Shortest paths		Yes
Minimal use of memory space	Yes	

9.6 Topological Sort

Topological sort is an ordering of vertices in a directed acyclic graph [DAG] in which each node comes before all nodes to which it has outgoing edges. As an example, consider the course prerequisite structure at universities. A directed *edge* (*v*,*w*) indicates that course *v* must be completed before course *w*. Topological ordering for this example is the sequence which does not violate the prerequisite requirement. Every DAG may have one or more topological orderings. Topological sort is not possible if the graph has a cycle, since for two vertices *v* and w on the cycle, *v* precedes w and w precedes *v*.

Topological sort has an interesting property. All pairs of consecutive vertices in the sorted order are connected by edges; then these edges form a directed Hamiltonian path [refer to *Problems Section*] in the DAG. If a Hamiltonian path exists, the topological sort order is unique. If a topological sort does not form a Hamiltonian path, DAG can have two or more topological orderings. In the graph below: 7, 5, 3, 11, 8, 2, 9, 10 and 3, 5, 7, 8, 11, 2, 9, 10 are both topological orderings.



Initially, *indegree* is computed for all vertices, starting with the vertices which are having indegree 0. That means consider the vertices which do not have any prerequisite. To keep track of vertices with indegree zero we can use a queue.

All vertices of indegree 0 are placed on queue. While the queue is not empty, a vertex *v* is removed, and all edges adjacent to *v* have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0. The topological ordering is the order in which the vertices DeQueue.

The time complexity of this algorithm is $O(|E| + |V|)$ if adjacency lists are used.