Chapter Table of Contents

Chapter 5.2

Graph Fundamentals

Aim

To educate the students about the basics of Graphs and their applications

Instructional Objectives

After completing this chapter, you should be able to:

- Describe graph and its types
- Discuss the depth first search and breadth first search algorithm
- Distinguish between different types of graphs

Learning Outcomes

At the end of this chapter, you are expected to:

- Outline the steps to traverse depth first search and breadth first search
- Write a C programme for DFS and BFS
- Compare DFS and BFS
- Outline applications of graph

Introduction

Till now we have studied data structures like arrays, stack, queue, linked list, trees etc. In this chapter, we introduce an important mathematical and graphical structure called as Graphs. Graphs are used in subjects like Geography, Chemistry etc. This chapter deals with the study of graphs and how to solve problems using graph theory. This chapter also deals with different types of Graphs traversals.

5.2.1 Definition of Graph

A Graph G is a Graphical representation of a collection of objects having vertices and edges. Vertices or nodes are formed by the interconnected objects. The links that joins a pair of vertices is called as Edges.

A Graph G consists of a set V of vertices and a set E of edges. A Graph is defined as G = (V, E), where V is a finite and non-empty set of all the Vertices and E is a set of pairs of vertices called as edges. A graph is depicted below in the figure 5.2.1.

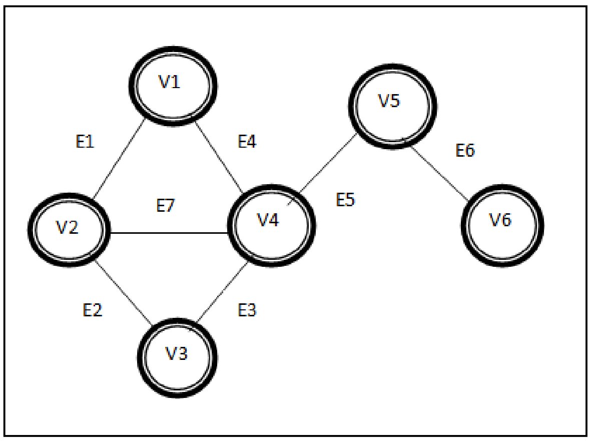


Figure 5.2.1: A Graph

Thus V (G) is the set of vertices of a graph and E (G) is a set of edges of that Graph.

Figure 5.2.1 shows an example of a simple graph. In this graph:

V (G) = {V1, V2, V3, V4, V5, V6}, forming 6 vertices and

E (G) = {E1, E2, E3, E4, E5, E6E, E7}, forming 7 edges.

Graphs are one of the objects of study in discrete mathematics and are very important in the field of computer science for many real world applications from building compilers to modelling communication networks.

A graph, G, consists of two sets V and E. V is a finite non-empty set of vertices. E is a set of pairs of vertices, these pairs are called edges.

V(G) and E(G) will represent the sets of vertices and edges of graph G.

We can also write G = (V,E) to represent a graph.

Terminologies used in a Graph

1. Adjacent Vertices:

Any vertex v1 is said to be an adjacent vertex of another vertex v2, if there exists and edge from vertex v1 to vertex v2.

Consider the following graph in figure 5.2.2. In this graph, adjacent vertices to vertex V1 are V2 and V4, whereas adjacent vertices for V4 are vertex V2, V1 and V5

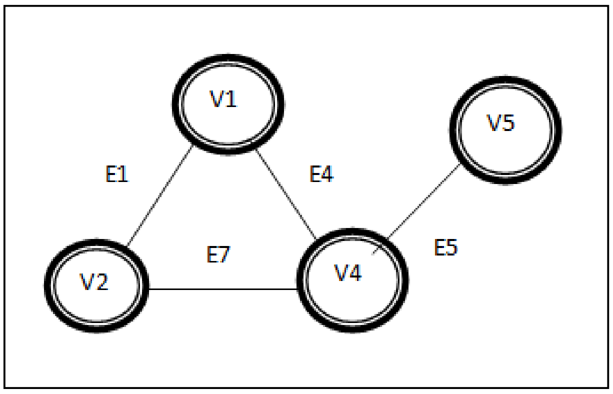


Figure 5.2.2: Adjacent Vertices

2. Point:

A point is any position in 1-Dimesioanl, 2-Dimensional or 3-Dimensional space. It is usually denoted by an alphabet or a dot.

3. Vertex:

A vertex is defined as the node where multiple lines meet. In the above figure, V1 is a vertex; V2 is a vertex and so on.

4. Edge:

An edge is a line which joins any two vertices in a graph. In the above sample graph, E1 is an Edge joining two vertices V1 and V2.

5. Path:

A path is a sequence of all the vertices adjacent to the next vertex starting from any vertex v. Consider above given figure 5.2.2. In this figure, V1, V2, V4, V5 is a path.

6. Cycle:

A Cycle is a path by itself having same first and last vertices. Thus it forms a cycle or a loop like structure. In the above figure, V1, V2, V4, V1 is and Cycle.

7. Connected Graph:

A graph is a connected graph if there is a path from any vertex to any other vertex. Consider the following figure 5.2.3 (a), this graph is a connected graph as there is a path from every vertex to other vertex.

Figure 5.2.3(b) shows an unconnected graph as there is no edge between vertex V4 and V5. Thus it forms two disconnected components of a graph.

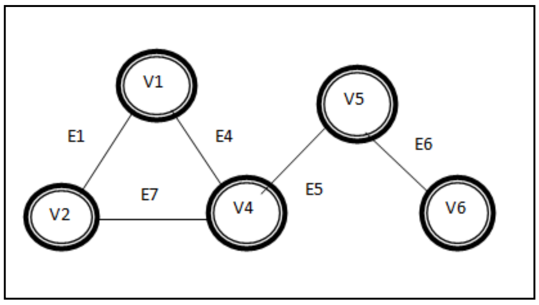


Figure 5.2.3 (a): A Connected Graph

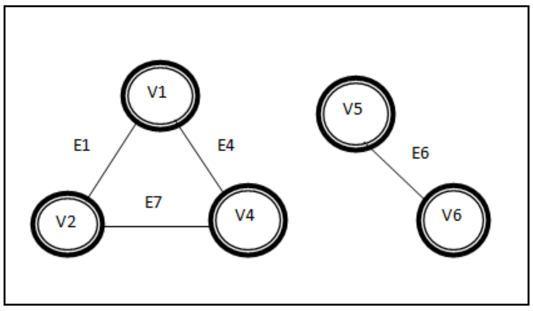
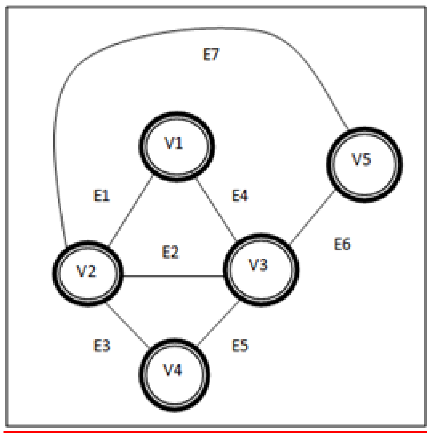


Figure 5.2.3 (b): An Unconnected Graph

8. Degree of Graph:

It means the number of edges incident on a vertex. The degree of any vertex v is denoted as degree (v). If degree (v) is 0, it means there are no edges incident on vertex v. Such a vertex is called as an isolated vertex. Below figure 5.2.4 shows a graph and its degrees of all the vertices.



Vertex	Degree
V1	**2**
V2	**4**
V3	**4**
V4	**2**
V5	**2**

Figure: 5.2.4: Degree of Vertices

9. Complete Graph:

A graph is said to be a Complete Graph if there is a path from every vertex to every other vertex. It is also called as Fully Connected Graph.

Figure 5.2.5 shows a complete Graph.

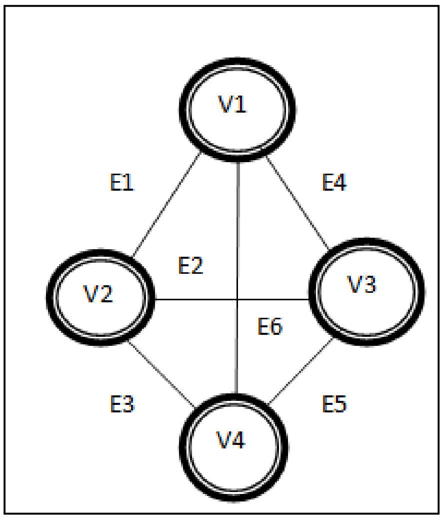


Figure 5.2.5: A Complete or Fully Connected Graph

Graph Data Structure

Graphs can be formally defines as an abstract data type with data objects and operations on it as follows:

Data objects: A graph G of vertices and edges. Vertices represent data or elements. The below given figure 5.2.6 shows a simple graph with 5 vertices.

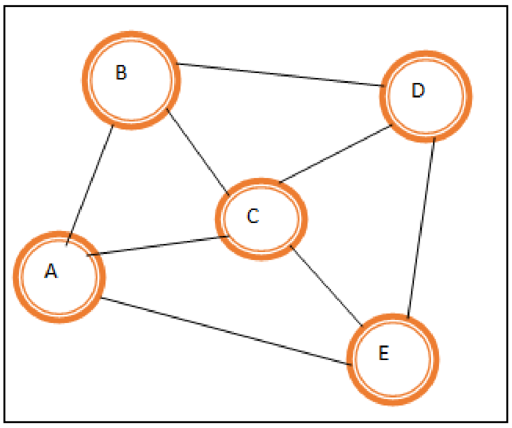


Figure 5.2.6: A Simple Graph

Operations

- **Check-Graph-Empty (G):** Check if graph G is empty - Boolean function. Above graph is having 5 vertices, thus this operation will give a false value as graph is not empty.

- **Insert-Vertex (G, V)**: Insert an isolated vertex V into a graph G. Ensure that vertex V does not exist in G before insertion. In the below graph in figure 5.2.7, we are adding a new vertex named F. Before adding this vertex to the graph, it acts as an isolated vertex.

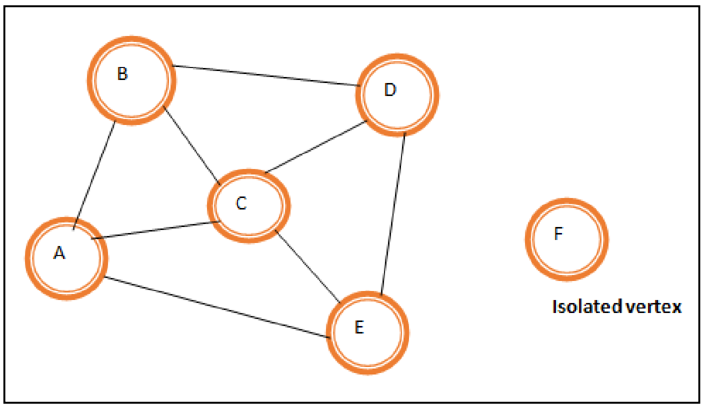


Figure 5.2.7: Isolated Vertex

- **Insert-Edge (G, u, v):** Insert an edge connecting vertices u, v into a graph G. Ensure that an edge does not exist in G before insertion. In the below given figure 5.2.8, a new edge joining vertices E and F is inserted.

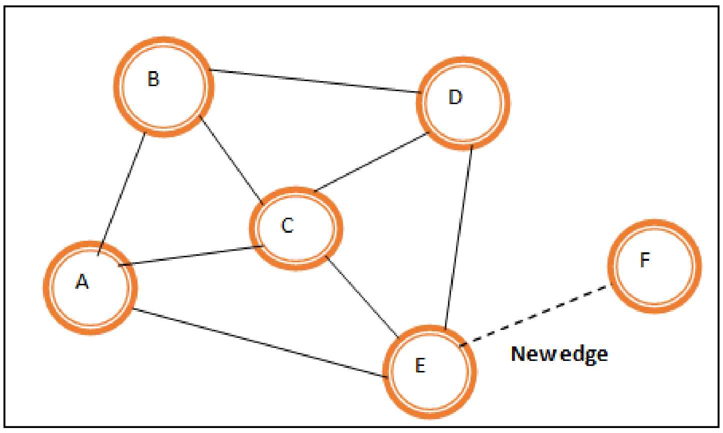


Figure 5.2.8: Adding New Edge

- **Delete-Vertex (G, V):** Delete vertex V and the entire edges incident on it from the graph G. Ensure that such a vertex exists in the graph G before deletion. In the graph given in below figure 5.2.9, a vertex D is deleted along with all the vertices incident on that vertex D.

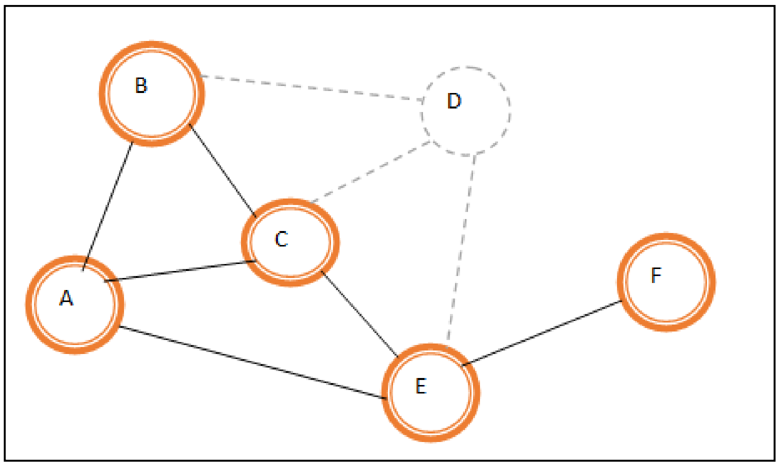


Figure 5.2.9: Vertex Deletion

- **Delete-Edge (G, u, v):** Delete an edge from the graph G connecting the vertices u, v. Ensure that such an edge exists before deletion. In the graph shown below (figure 5.2.10), an edge connecting vertices B and D is removed.

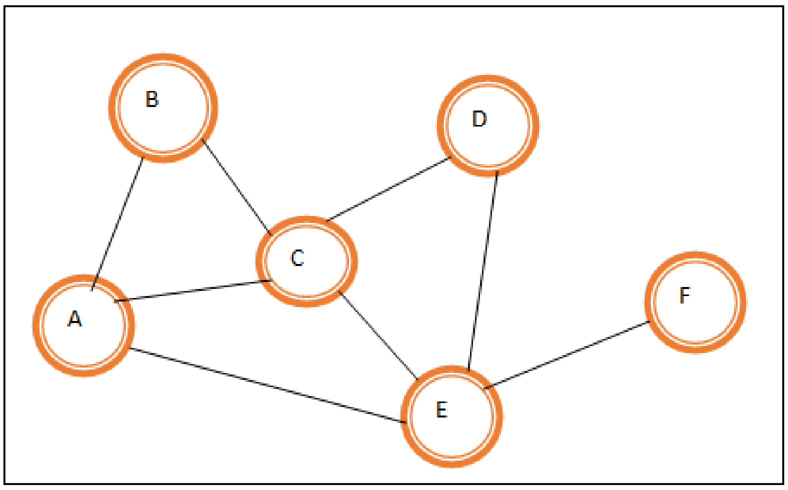


Figure 5.2.10: Deleting an Edge

- **Store-Data (G, V, Item):** Store Item into a vertex V of graph G. Once a vertex is created, its value can be added too. In below given graph (figure 5.2.11) a vertex is created and item G is stored in that vertex.

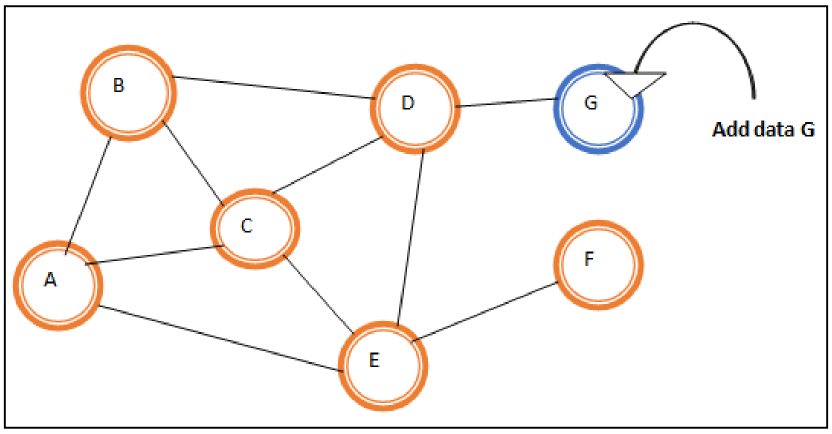


Figure 5.2.11: Adding Data into Vertex

- **Retrieve-Data (G, V, and Item):** Retrieve data of a vertex V in the graph G and return it in Item. In the below given example (figure 5.2.12) we have retrieved data from the vertex G.

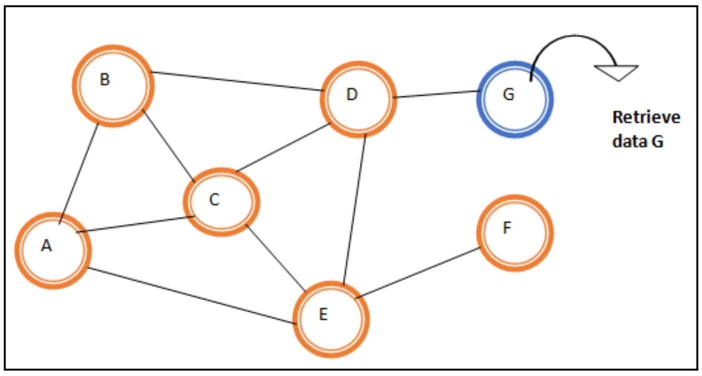


Figure 5.2.12: A Complete or Fully Connected Graph

- **BFT (G):** Perform Breath First Traversal of a graph. This traversal starts from the root node and explores the nodes level wise, thus exploring a node completely.
- **DFT (G):** Perform Depth First Traversal of a graph. This traversal starts from a root node and visit all the adjacent nodes completely in depth before backtracking.

Representation of Graphs

Since a graph is a mathematical structure, the representation of graphs is categorised into two types, namely (i) sequential representation and (ii) linked representation. Sequential representation uses array data structure whereas linked representation uses single linked list as its data structure.

The sequential or the matrix representations of graphs have the following methods:

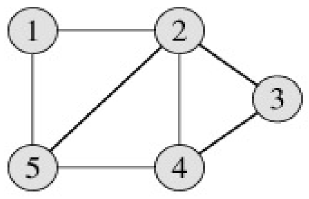
- Adjacency Matrix Representation
- Incidence Matrix Representation

a) Adjacency Matrix Representation

A graph with n nodes can be represented as n x n Adjacency Matrix A such that an element

Ai j =	1	if there exists an edge between nodes i and j
	0	Otherwise

For example 1:



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Figure 5.2.13 (a) Graph *Figure 5.2.13 (b) Adjacency Matrix*

Explanation: Above graph contains 5 vertices: 1, 2, 3, 4 and 5

Consider vertex 1: vertex 1 is connected to vertex 2 and vertex 5.

Thus $A_{1,2}$=1 and $A_{1,5}$=1

Similarly vertex 1 is not connected to vertex 3, 4 and 1 itself

Thus $A_{1,3}$=0, $A_{1,4}$=0 and $A_{1,1}$=0.

Consider vertex 5: vertex 5 is connected to vertex 1, vertex 2 and vertex 4.

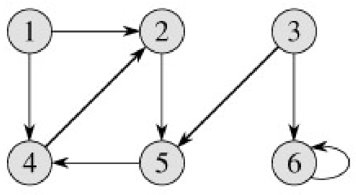
Thus $A_{5,1}$=1, $A_{5,2}$=1, and $A_{5,4}$=1

Similarly vertex 5 is not connected to vertex 3 and 5 itself

Thus $A_{5,3}$=0 and $A_{5,5}$=0

Same rule follows for other vertices also.

For example 2:



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Figure 5.2.14 (a) Digraph *Figure 5.2.14 (b) Adjacency Matrix*

Consider vertex 4: The given graph is a digraph, hence there is only an unidirectional path from 4 to 2.

Thus $A_{4,2}$=1

Similarly vertex 4 is not connected to any other vertex rest all entries in that row are 0.

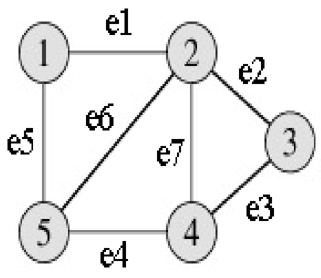
b) Incidence Matrix Representation

Let G be a graph with n vertices and e edges. Define an n x e matrix M = [mij] whose n rows

Corresponds to n vertices and e columns correspond to e edges, as

Aij = 1 ej incident upon vi

For example:



	e1	e2	e3	e4	e5	e6	e7
v1	1	0	0	0	1	0	0
v2	1	1	0	0	0	1	1
v3	0	1	1	0	0	0	0
v4	0	0	1	1	0	0	1
v5	0	0	0	1	1	1	0

Figure 5.2.15 (a) Undirected Graph *Figure 5.2.15 (b) Incidence Matrix*

Explanation: Above graph contains 5 vertices: 1, 2, 3, 4 and 5

Consider vertex 1: 2 edges are incident on vertex 1. They are e1 and e5.

Thus $A_{1,e1}$=1 and $A_{1,e5}$=1

Rest of the entries in that row will be 0.

Consider vertex 4: 3 edges are incident on vertex 4. They are e3, e4 and e7.

Thus $A_{4,e3}$=1, $A_{4,e4}$=1 and $A_{4,e7}$=1

Rest of the entries in that row will be 0.

The incidence matrix contains only two elements, 0 and 1. Such a matrix is also called as binary matrix or a (0, 1)-matrix.

c) Linked Representation of Graphs

The linked representation of graphs is also referred to as adjacency list representation and is comparatively efficient with regard to adjacency matrix representation.

In Linked representation of Graphs, a graph is stores as a linked structure of nodes. It can be defined as a Graph G=(V, E), where all vertices are stored in a list and each vertex points to a singly linked list of nodes which are adjacent to that head node.

For example 1:

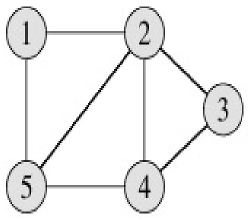


Figure 5.2.16 (a) Undirected Graph

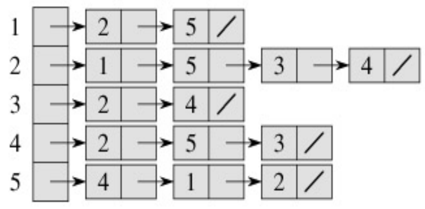


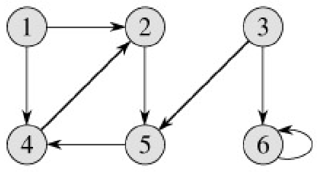
Figure 5.2.16 (b) Linked Representation of a Graph

In the above figure 5.2.16(a), an undirected graph is shown. In figure 5.2.16(b), its equivalent Linked list representation is shown.

Adjacent vertices to vertex 1 are vertex 2 and 5. Thus in the linked list representation, vertex 1 is shown linked to node 2 and node 2 is linked to node 5, thus forming a chain like structure.

Similarly, adjacent vertices to vertex 2 are vertex 1, 5, 3 and 4. Thus in the linked list representation, vertex 2is shown linked to node 1, node 1 is linked to node 5, node 5 is linked to node 3 and node 3 in turn is connected to node 4 thus forming a linked list of nodes.

For example 2:



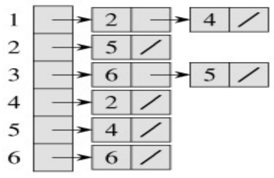


Figure 5.2.17 (a) Digraph *Figure 5.2.17 (b) Linked Representation of a Graph*

Did you know?

Social network graphs: to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

Self-assessment Questions

1) A Graph is a mathematical representation of a set of objects consisting of ________ and ____________.

a) Vertices and indices	b) Indices and edges
c) Vertices and edges	d) Edges and links

2) In a graph if e=(u,v) means _____________.
   a) *u* is adjacent to *v* but *v* is not adjacent to *u*.
   b) *e* begins at u and ends at *v*
   c) *u* is node and *v* is an edge.
   d) Both *u* and *v* are edges.

3) Graph can be represented as an adjacency matrix

a) True	b) False

4) A ________ is a particular position in a one-dimensional, two-dimensional, or three-dimensional space.

a) Point	b) Node
c) Edge	d) Vertex

5.2.2 Types of Graphs

Based on the degree of vertex, graphs are categorised into two types namely,

1. Undirected Graph
2. Directed Graph

In an *undirected graph,* the pair of vertices representing any edge is unordered. Thus, the pairs ($v1$, $v2$) and ($v2$, $v1$) represent the same edge.

In a *directed graph,* each edge is represented by a directed pair ($v1$, $v2$) where $v1$ is the *tail* and $v2$ is the *head* of the edge. Therefore <$v2$, $v1$> and <$v1$, $v2$> represent two different edges.

(i) Defined Graph

Degree of Vertex in a Defined Graph

In a directed graph, each vertex has an indegree and an outdegree.

Consider the following Directed graph from below figure:

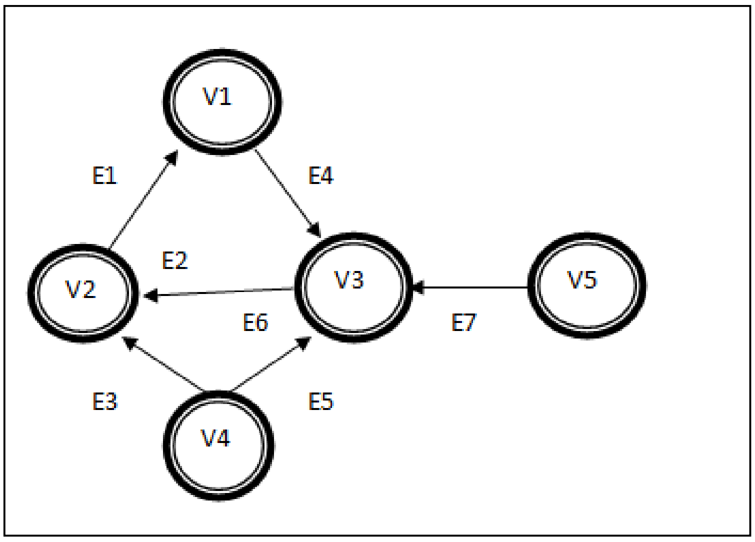


Figure 5.2.18: Directed Graph

Indegree of a Graph

Indegree of vertex V is the number of edges which are coming into the vertex V (incoming edges).

Notation – $\deg^+(V)$.

In the above example directed graph, there are 5 vertices: V1, V2, V3, V4 and V5.

Consider vertex V1: V2 is connected to V1 through edge E1. The edge comes from V2 towards Vertex V1. Thus the Indegree of vertex V1 is 1.

i.e., deg+(V1)=1

Consider vertex V3: V1 and V4 are connected to V3 through edges E4 and E5. Thus the Indegree of vertex V3 is 2.

i.e., deg+(V3)=2

Outdegree of a Graph

Outdegree of vertex V is the number of edges which are going out from the vertex V (outgoing edges).

Notation – $\deg^{-}(V)$.

In the above example directed graph, there are 5 vertices: V1, V2, V3, V4 and V5.

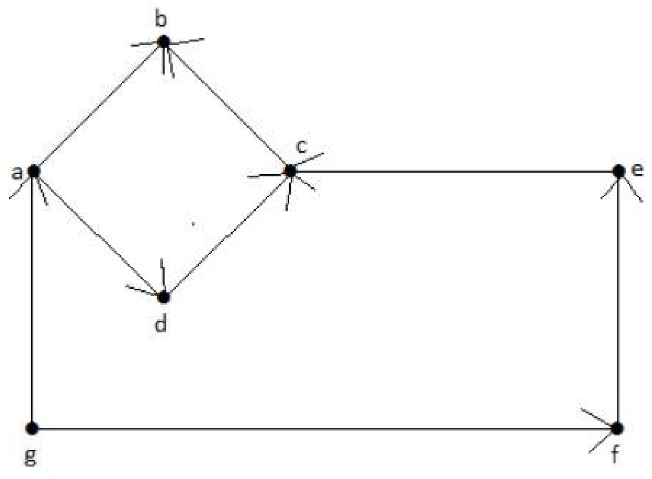
Consider vertex V1: V1 is connected to V3 through edge E4. The edge goes from V1 towards Vertex V3. Thus the Outdegree of vertex V1 is 1.

i.e., deg-(V1)=1

For example 1:

Consider the following directed graph.

Vertex ‘a’ has two edges, ‘ad’ and ‘ab’, which are going outwards. Hence its outdegree is 2. Similarly, there is an edge ‘ga’, coming towards vertex ‘a’. Hence the indegree of ‘a’ is 1.



The indegree and outdegree of other vertices are shown in the following table.

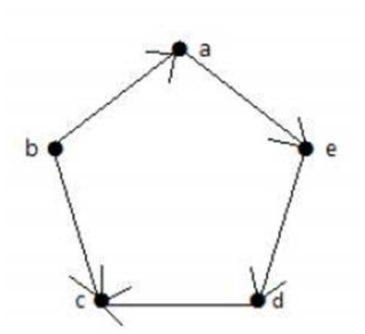
Table 5.2.1 : Indegree and Outdegree of Vertices

Vertex	Indegree	Outdegree
a	1	2
b	2	0
c	2	1
d	1	1
e	1	1
f	1	1
g	0	2

For example 2:

Consider the following directed graph.

Vertex ‘a’ has an edge ‘ae’ going outwards from vertex ‘a’. Hence its outdegree is 1. Similarly, the graph has an edge ‘ba’ coming towards vertex ‘a’. Hence the indegree of ‘a’ is 1.



The indegree and outdegree of other vertices are shown in the following table.

Table 5.2.1: Indegree and Outdegree of Vertices

Vertex	Indegree	Outdegree
a	1	1
b	0	2
c	2	0
d	1	1
e	1	1

(ii) Undefined Graph

An undefined graph is a graph in which the nodes are connected by *undefinedarcs*. An undefined arc is an edge that has no arrow. Both ends of an undefined arc are equivalent--there is no head or tail. Therefore, we represent an edge in an undefined graph as a set rather than an ordered pair:

Definition (Undefined Graph) An defined *graph* is an ordered pair $G = (\mathcal{V}, \mathcal{E})$ with the following properties:

1. The first component, $\mathcal{V}$, is a finite, non-empty set. The elements of $\mathcal{V}$ are called the *vertices* of G.

2. The second component, $\mathcal{E}$, is a finite set of sets. Each element of $\mathcal{E}$ is a set that is comprised of exactly two (distinct) vertices. The elements of $\mathcal{E}$ are called the *edges* of G.

For example, consider the undefined graph $G_4 = (\mathcal{V}_4; \mathcal{E}_4)$ comprised of four vertices and four edges:

$$\begin{aligned}\mathcal{V}_4 &= \{a, b, c, d\} \\ \mathcal{E}_4 &= \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}\end{aligned}$$

The graph G_4 can be represented *graphically* as shown in Figure 5.2.19. The vertices are represented by appropriately labelled circles, and the edges are represented by lines that connect associated vertices.

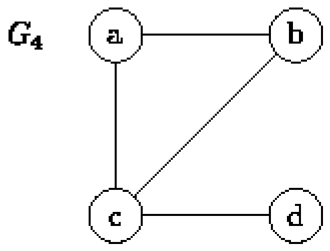


Figure 5.2.19: An Undefined Graph

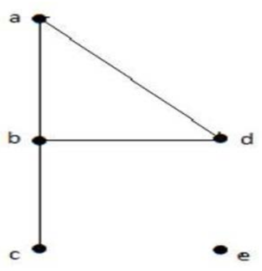
Notice that because an edge in an undefined graph is a set, $\{a, b\} \equiv \{b, a\}$, and since $\mathcal{E}_4$ is also a set, it cannot contain more than one instance of a given edge. Another consequence of Definition is that there cannot be an edge from a node to itself in an undirected graph because an edge is a set of size two and a set cannot contain duplicates.

Degree of Vertex in a Directed Graph

In a defined graph, each vertex has an indegree and an outdegree.

For example 1:

Consider the following graph –



In the above Undirected Graph,

deg(a) = 2, since there are 2 edges meeting at vertex '*a*'.

deg(b) = 3, since there are 3 edges meeting at vertex '*b*'.

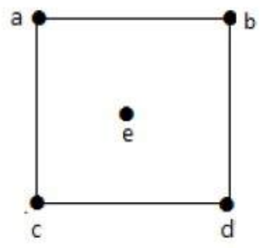
deg(c) = 1, since there is 1 edge formed at vertex '*c*'

deg(d) = 2, since there are 2 edges meeting at vertex '*d*'.

deg(e) = 0, since there are 0 edges formed at vertex '*e*'.

For example 2:

Consider the following graph –



In the above graph,

deg(a) = 2, deg(b) = 2, deg(c) = 2, deg(d) = 2, and deg(e) = 0.

Tree

A tree can be defined as a nonlinear data structure similar to graphs, in which all the elements are arranged in a sorted manner. A tree can be used to represent some hierarchical relations among various data elements. Trees do not contain any Cycle.

Tree has a **root** node from where the tree structure begins. Starting from the root node the tree will have many subtrees formed by its child nodes. A **node** is a data element present in a tree.

The figure 5.2.20 given below shows a simple tree:

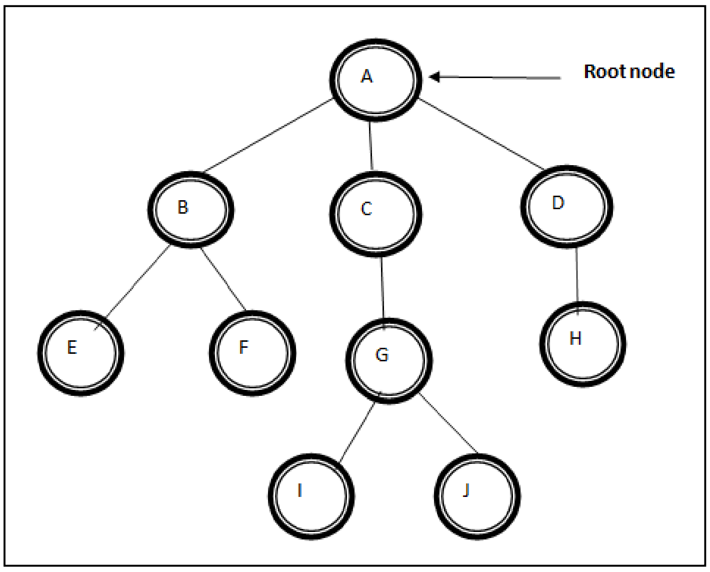


Figure 5.2.20: A Tree

Degree of any node:

- Degree of a node is defined as a number of subtrees of that node.
- ***For example,*** Degree of nod A is 3, whereas Degree of node H is 0 as there are no subtrees of H.

Degree of a tree:

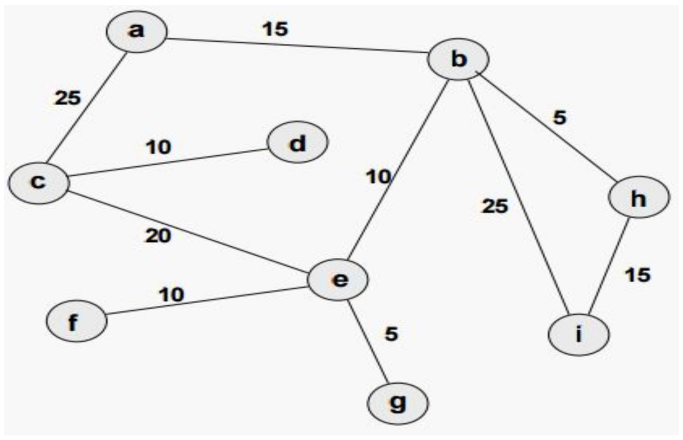
- Degree of a tree is the maximum degree of any nodes in the given tree.
- ***For example,*** in the above given tree, Degree of that tree is 3 as node A is having the degree 3, which is the maximum value in that tree.

Did you know?

Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.

The weight of an edge is often referred to as the "cost" of the edge.

In applications, the weight may be a measure of the length of a route, the capacity of a line, the energy required to move between locations along a route, etc.



Given a weighted graph, and a designated node S, we would like to find a path of least total weight from S to each of the other vertices in the graph.

The total weight of a path is the sum of the weights of its edges.

Self-assessment Questions

5) An undirected graph has no directed edges
   - a) True
   - b) False

6) ____________ of vertex V is the number of edges which are coming into the vertex
   - a) Degree
   - b) Indegree
   - c) Outdegree
   - d) Path

7) A connected acyclic graph is called a ____________
   - a) Connected Graph
   - b) Tree
   - c) Hexagon
   - d) Pentagon

8) _______________ of vertex V is the number of edges which are going out from the vertex
   - a) Degree
   - b) Indegree
   - c) Outdegree
   - d) Path

5.2.3 Graph Traversal

A Graph traversal is a method by which we visit all the nodes in any given graph. Graph traversals are required in many application areas, like searching an element in graph, finding the shortest path to any node etc. There are many methods for traversing through a graph. In this chapter we will study the following 2 methods for graph traversals. They are:

1. Depth First Search (DFS)
2. Breadth First Search (BFS)

(i) Depth First Search (DFS) Traversal

In this Depth First Search method, we need to start from any root node in a graph and explore all the nodes along each branch before backtracking. It means we need to explore all the unvisited graph nodes of any root node. We can use a Stack data structure to keep track of the visited nodes of a graph.

Algorithm:

1. Start
2. Consider a Graph G=(V, E)
3. Initially mark all the nodes of graph G as unseen
4. Push Root node onto the Stack from where it begins
5. Repeat until the Stack S is empty
6. Pop the node from the Stack S
7. If this popped node is having any unseen nodes, traverse the unseen child nodes, mark them as visited and push it on stack
8. If the node is not having any unseen child nodes, Pop the node from the stack S
9. End

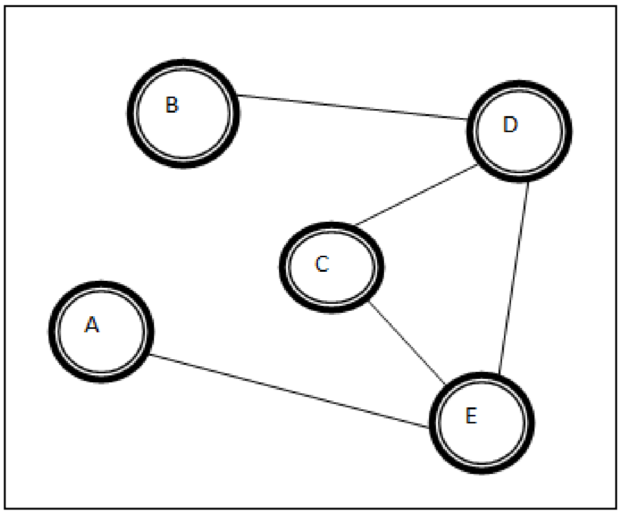
Pseudo Code:

Consider a Graph G= (V, E) where V is set of vertices and E is a set of Edges.

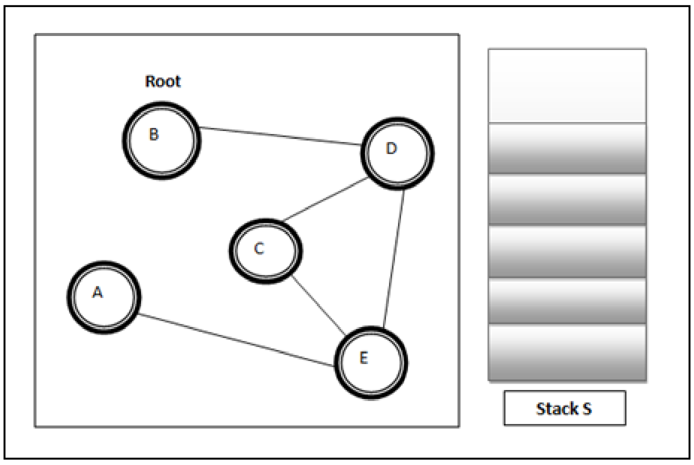
```
DFS(G, Root)
{
  root: is a vertex from where the traversal begins
  Visited[v] is a status flag denoting that any vertex v is visited
  Consider Stack s used to store visited vertices
  For each vertex v in the graph
  Set Visited[v] = false;     //to mark all nodes unvisited initially
  Push root vertex on Stack S;            //Start from Root vertex
  While Stack is not empty
  {
    Pop element from S and put in v
    If (not visited[v] =true)
    {
       For every unvisited node x of v
       Push vertex x on Stack S
    }
  }
}
```

Example:

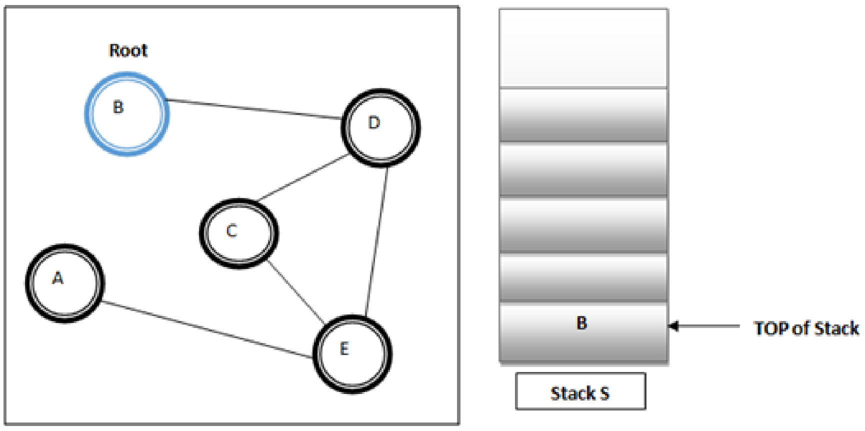
Consider the following Graph. We will apply the above algorithm on this graph to implement Depth first Search Traversal.



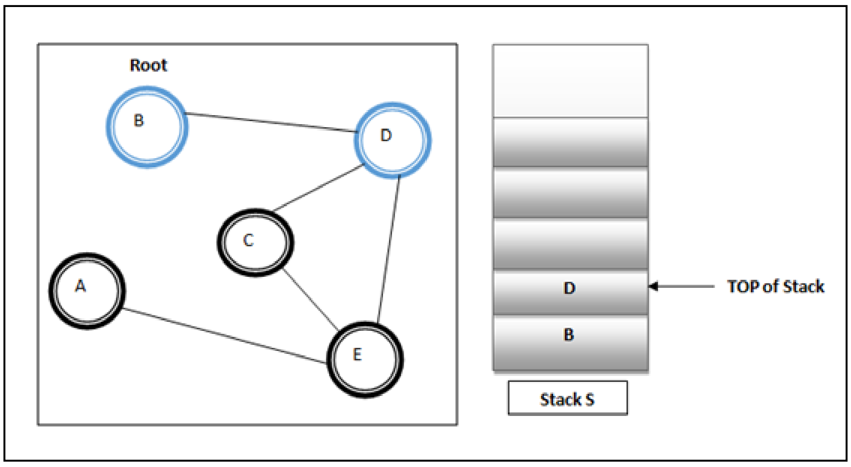
Step 1: Initially we start from root node B. Stack S is empty.



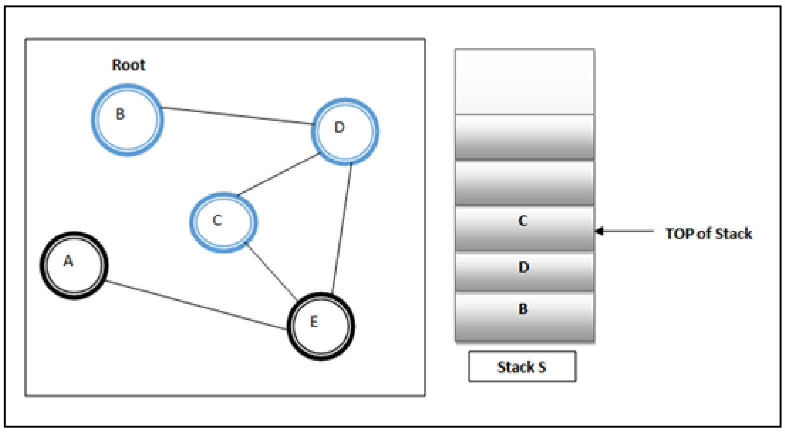
Step 2: Mark the root node B as visited and Push B on the Stack S.



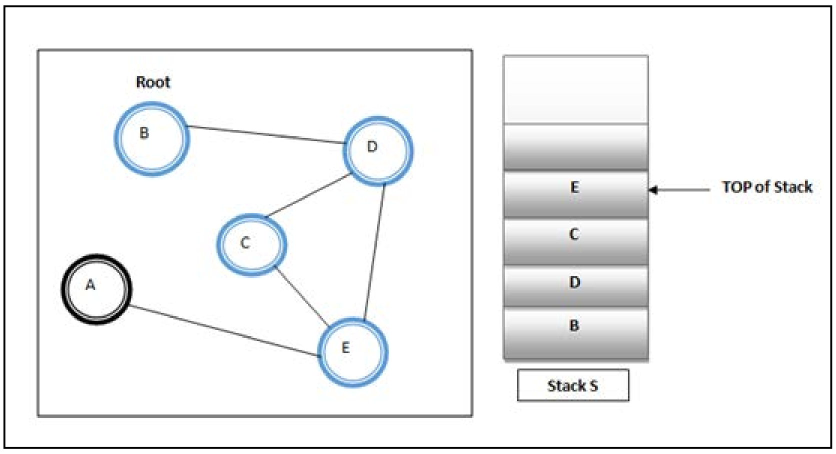
Step 3: Check for the nodes adjacent to node B. Only Node D is adjacent to node B. So visit that node D and Push it onto the stack S.



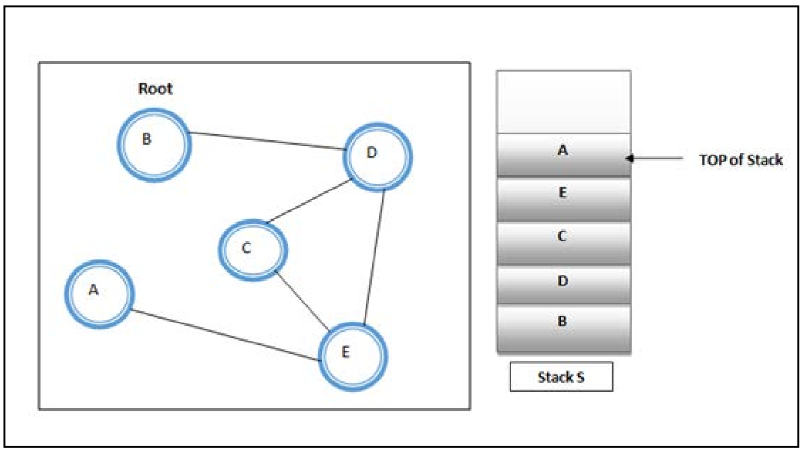
Step 4: Check for the nodes adjacent to node D. There are 3 Nodes adjacent to node D. They are B, C and E. But node B is already visited. So we need to consider either of the nodes C and E. Let us consider node C. Mark first child node C as visited and push C on stack S.



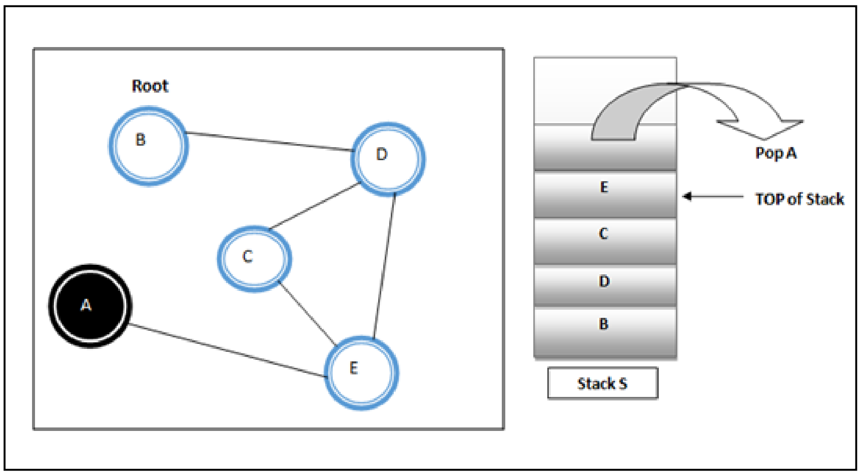
Step 5: Check for the nodes adjacent to node C. There are 2 Nodes adjacent to node C. They are D and E. But node D is already visited. So select node E as the next node. Mark node E as visited and push E on stack S.



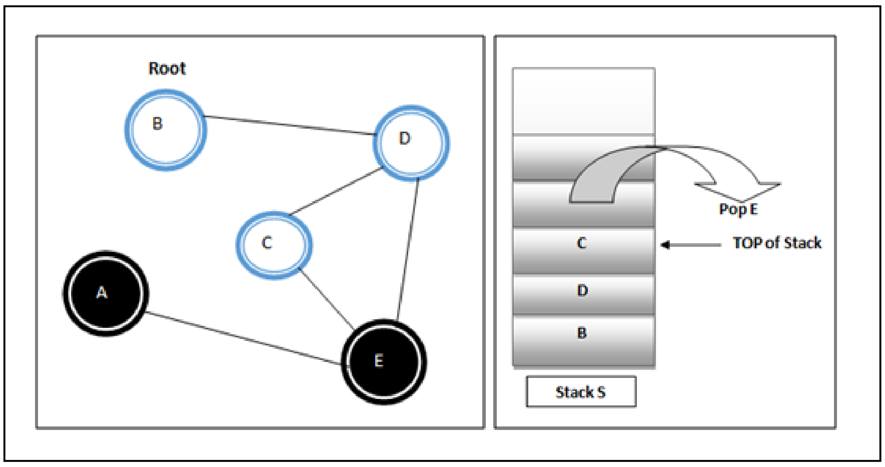
Step 6: Check for the nodes adjacent to node E. There are 3 Nodes adjacent to node E. They are D, C and A. But nodes D and C are already visited. So select node A as the next node. Mark node A as visited and push A on stack S.



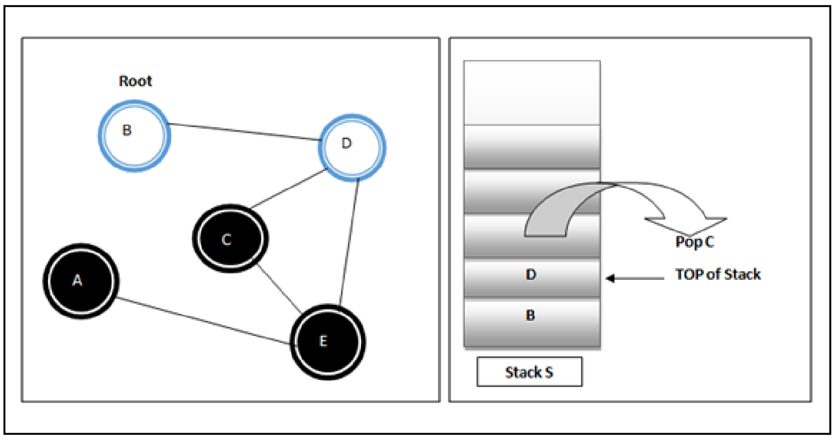
Step 7: Check for the nodes adjacent to node A. But there is only 1 node adjacent to A i.e. E and E is already visited. So as per the algorithm, if there are no unseen nodes then pop that node from stack. So pop out node A from stack. Hence A has no vertices left to be visited now. So we need to backtrack to node E.



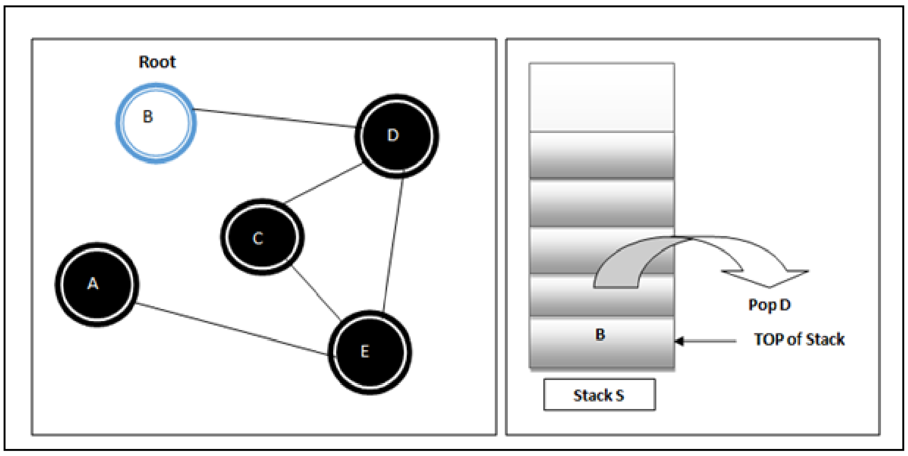
Step 8: Check for the nodes adjacent to node E. Nodes A, C and D are adjacent. But A, C and D are already marked as visited. So as per the algorithm, if there are no unseen nodes then pop that node from stack. So pop out node E from stack. Hence E has no vertices left to be visited now. So we need to backtrack to node C.



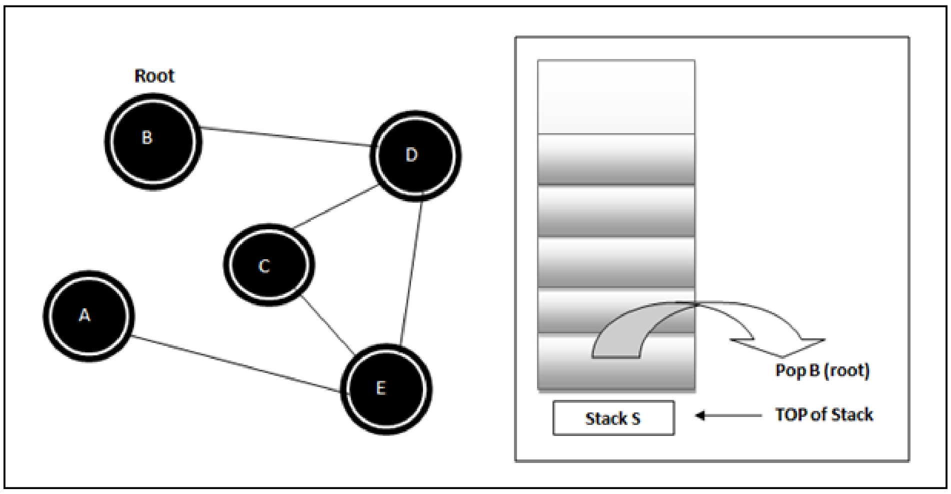
Step 9: Check for the nodes adjacent to node C. Nodes D and E are adjacent. But D and E are already marked as visited. So pop out node C from stack. So we need to backtrack to node D.



Step 10: Check for the nodes adjacent to node D. Nodes B, C and E are adjacent. But C, B and E are already marked as visited. So pop out node D also from stack. So we need to backtrack to node B.



Step 11: Check for the nodes adjacent to node B. Node D is adjacent. But D is already marked as visited. So pop out node B(root) also from stack. Hence the stack is empty now. It means the algorithm is successfully executed.



Result of DFS Traversal: B. D, C, E, A

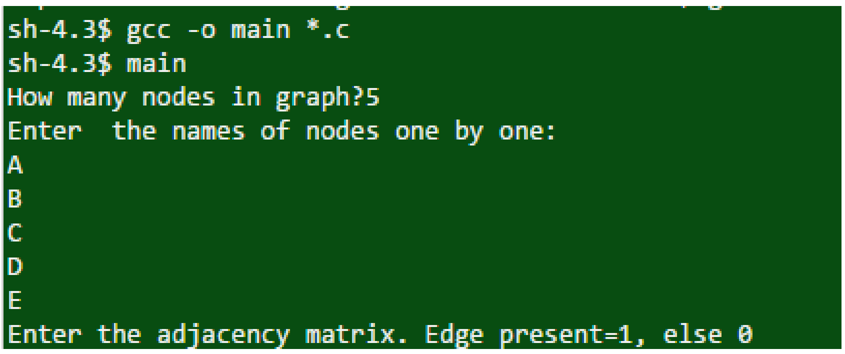
Program:

```
/*C program for Depth first search graph traversal using Stack */
#include<stdio.h>
char nodeid[20]; //to store nodes names
char stack[50];
int temp=0;
int tos=-1, nodes;  //top of stack initialized to -1
char arr[20];
char dfs(int );
```

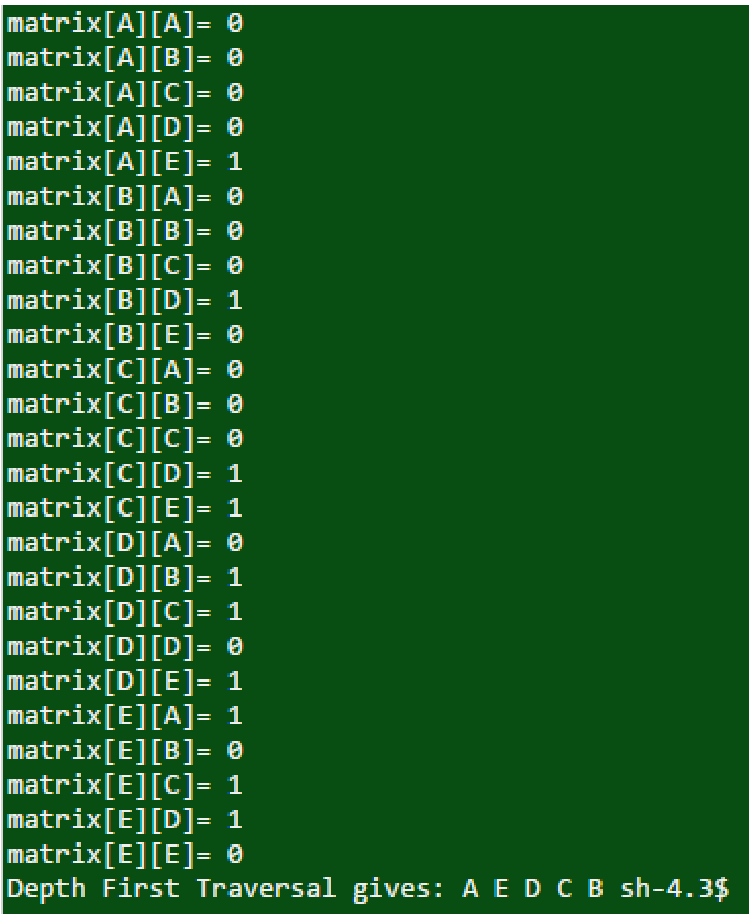
```
int matrix[30][30];

void push(char val) //push vertex on stack
{
     tos=tos+1;
     stack[tos]=val;
}
char pop()      //pop vertex from stack
{
     return stack[tos];
}
void outputDFS()
{
     printf("Depth First Traversal gives: ");
     for(int i=0; i<nodes; i++)
        printf("%c ",arr[i]);
}
int unVisited(char val)
{
      for(int i=0; i<temp; i++)
        if(val==arr[i])
            return 0;
      for(int i=0; i<=tos; i++)
          if(val==stack[tos])
            return 0;
        return 1;
}
char dfs(int i)
{
     int k;
     char m;
         if(tos==-1)
         {
              push(nodeid[i]);
         }
     m=pop();
     tos=tos-1;
     arr[temp]=m;
     temp++;
     for(int j=0; j<nodes; j++)
     {
          if(matrix[i][j]==1)
          {
               if(unVisited(nodeid[j]))
               {
                     push(nodeid[j]);
               }
          }
     }
    return stack[tos];
}
```

```
int main()
{
    char v;
    int l=0;
    printf("How many nodes in graph?");
    scanf("%d",&nodes);
    printf("Enter  the names of nodes one by one: \n");
    for(int i=0; i<nodes; i++)
        {
            scanf("%s",&nodeid[i]);
        }
    char root=nodeid[0]; //consider first node as root node
    printf("Enter the adjacency matrix. Edge present=1, else 0\n");
    for(int i=0;i<nodes; i++)
    {
         for(int j=0; j<nodes; j++)
         {
              printf("matrix[%c][%c]= ", nodeid[i], nodeid[j]);
              scanf("%d", &v);
              matrix[i][j]=v;
         }
    }
for(int i=0;i<nodes;i++)
    {
         l=0;
         while(root!=nodeid[l])
         l++;
         root=dfs(l);
    }
    outputDFS();
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
How many nodes in graph?5
Enter  the names of nodes one by one:
A
B
C
D
E
Enter the adjacency matrix. Edge present=1, else 0
```



```
matrix[A][A]= 0
matrix[A][B]= 0
matrix[A][C]= 0
matrix[A][D]= 0
matrix[A][E]= 1
matrix[B][A]= 0
matrix[B][B]= 0
matrix[B][C]= 0
matrix[B][D]= 1
matrix[B][E]= 0
matrix[C][A]= 0
matrix[C][B]= 0
matrix[C][C]= 0
matrix[C][D]= 1
matrix[C][E]= 1
matrix[D][A]= 0
matrix[D][B]= 1
matrix[D][C]= 1
matrix[D][D]= 0
matrix[D][E]= 1
matrix[E][A]= 1
matrix[E][B]= 0
matrix[E][C]= 1
matrix[E][D]= 1
matrix[E][E]= 0
Depth First Traversal gives: A E D C B sh-4.3$
```

(ii) Breadth First Search (BFS) Traversal

In this method of traversing, we select a node as a start node from where the traversal begins. It is visited and marked, and all the unvisited nodes adjacent to the next node are visited and marked in an order. Similarly, all the unvisited nodes of adjacent nodes are visited and marked until full graph nodes are covered.

Algorithm:

1. Start
2. Consider a Graph G=(V, E)
3. Initially mark all the nodes of graph G as unseen
4. Add the Root node into the Queue Q from where it begins
5. Repeat until the Queue Q is empty
6. Remove the node from the Queue Q

7. If this popped node is having any unseen nodes, visit and mark all the unvisited nodes.

8. End

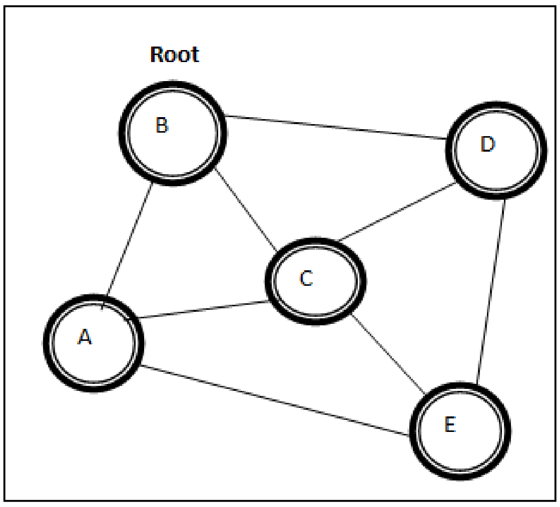
Pseudo Code:

Consider a Graph G= (V, E) where V is set of vertices and E is a set of Edges.

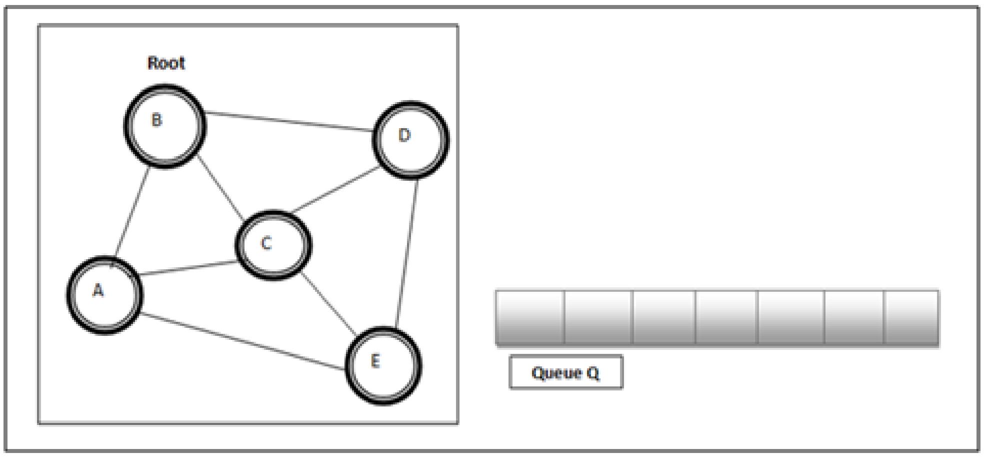
```
BFS (G, Root)
{
   Root: is a vertex from where the traversal begins
   Visited[v] is a status flag denoting that any vertex v is visited
   Consider Queue Q used to store visited vertices
   For each vertex v in the graph
   Set Visited[v] = false;    //to mark all nodes unvisited initially
   Add Root vertex to the Queue Q;        //Start from Root vertex
   While the Queue Q is not empty
   {
      Remove element from Q and put in v
      If (not visited[v] =true)
      {
         Visit and mark all the unvisited node x of v
      }
   }
}
```

Example:

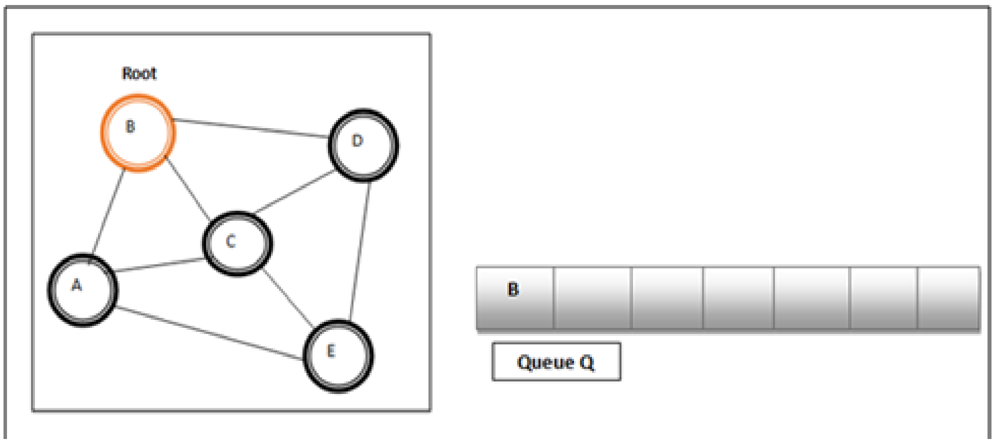
Consider the following Graph. We will apply the above algorithm on this graph to implement Breadth first Search Traversal.



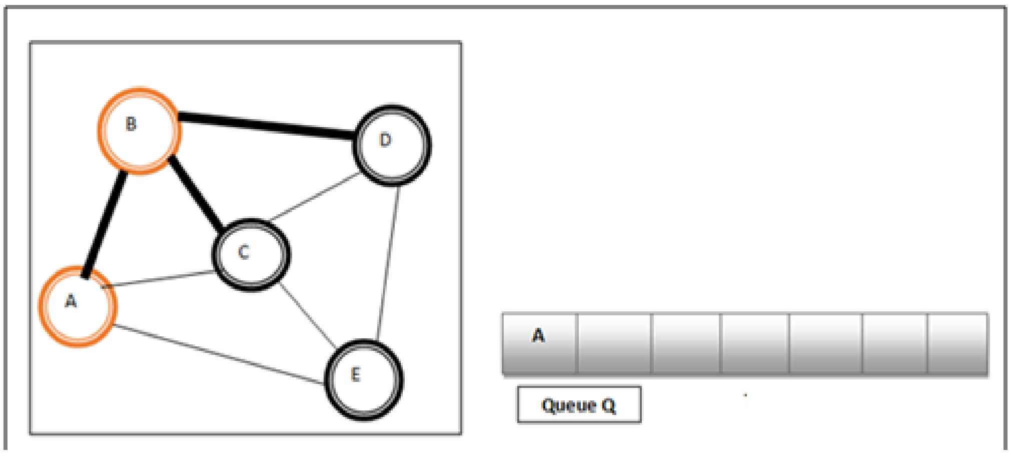
Step 1: We will start from node B. Hence B is a root node. We have a queue Q for keeping track of vertices.



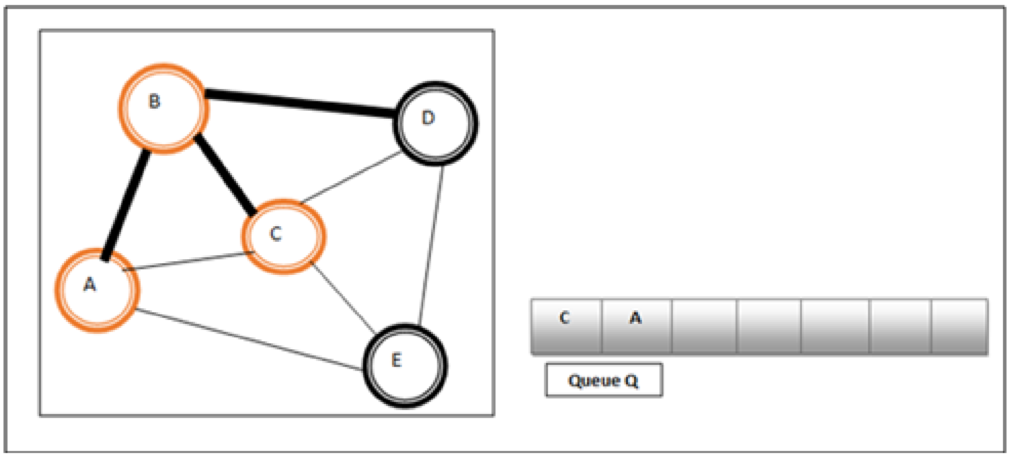
Step 2: We will start from node B. Hence B is a root node. We have a queue Q for keeping track of vertices. Add B to the Queue, Mark B as visited.



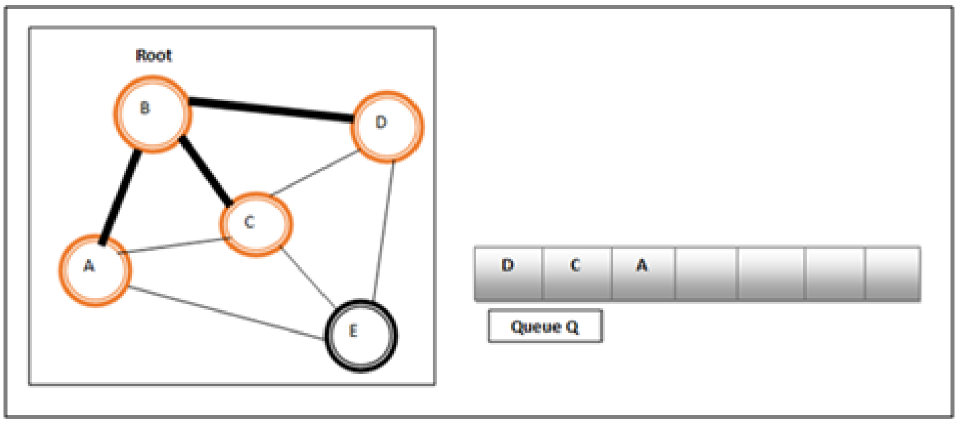
Step 3: Nodes adjacent to B are A, C and D. All 3 are unvisited. But we will start from A. As B is visited, Remove B from Queue. Mark A as visited and add it to the Queue Q.



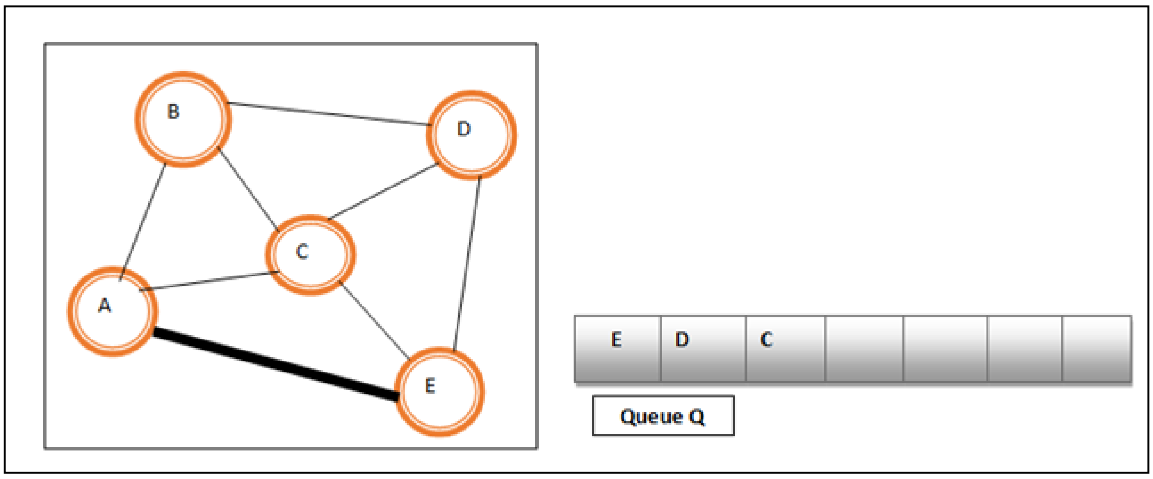
Step 4: Next Node adjacent to B is C. Mark C as visited and add it to the Queue Q.



Step 5: Next Node adjacent to B is D. Mark D as visited and add it to the Queue Q.

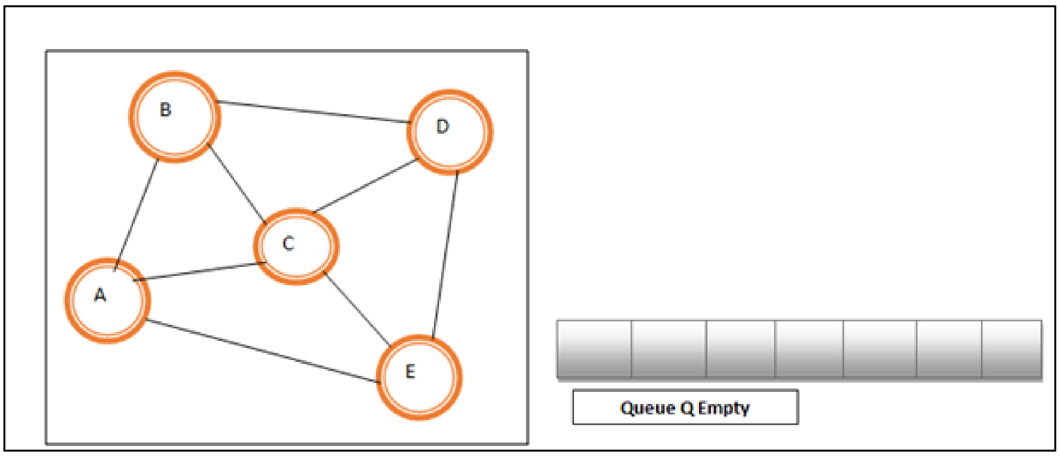


Step 6: Now the node B is fully explored. So, the next node to be visited is A. Nodes adjacent to node A are B, C and E. But B and C are already visited. Mark E as visited and add it to the Queue Q. But remove A from queue Q as it is fully explored.



Step 7: Now the nodes adjacent to E are A, C and D. But all are marked and visited.

If we see the graph, vertices E, C and D are fully visited so remove them from the queue. Hence the Queue is empty. Thus the algorithm is successfully implemented.



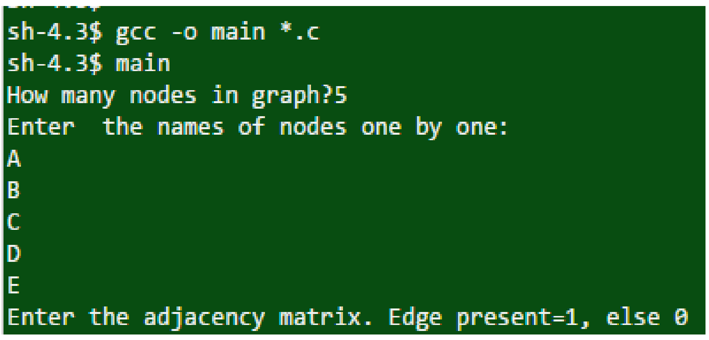
Result of BFS: B, A, C, D, E

Program:

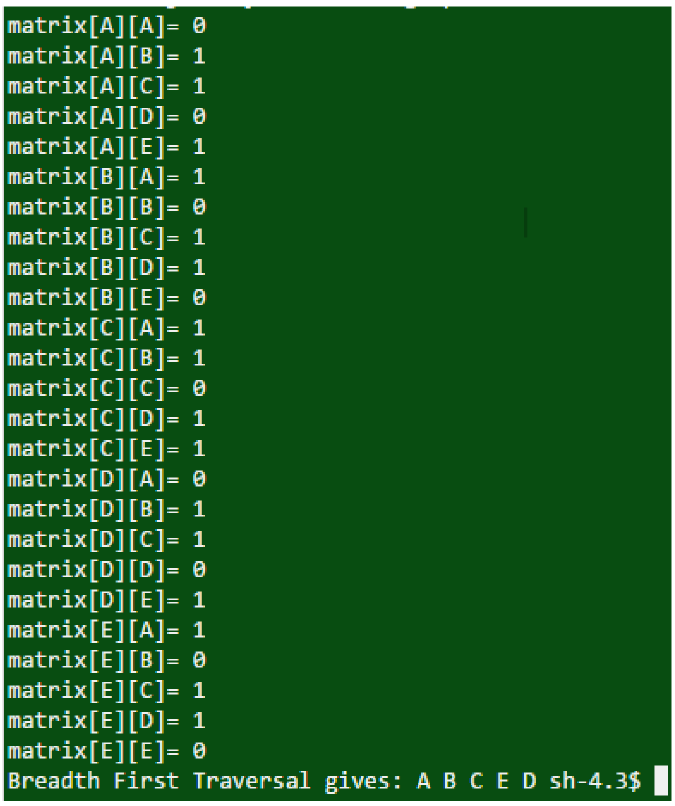
```
/*C program for Breadth first search graph traversal using Queue */
#include<stdio.h>
char nodeid[20]; //to store nodes names
char queue[50];
int temp=0;
int front=0, rear=0, nodes;  //front and rear =0
char arr[20];
```

```
int bfs(int);
int matrix[30][30];

void qadd(char value)   //function to add vertex to a queue
{
    queue[front]=value;
    front++;
}
char qremove()  //function to remove visited vertex from queue
{
    rear=rear+1;
    return queue[rear-1];
}

void outputBFS()
{
     printf("Breadth First Traversal gives: ");
     for(int i=0; i<nodes; i++)
        printf("%c ",arr[i]);
}
int unVisited(char value)
{
    for(int i=0; i<front; i++)
    {
         if(value==queue[i])
         return 0;
    }
    return 1;
}

int bfs(int i)
{
     char r;
     if(front==0)
     {
         qadd(nodeid[i]);
     }
     for(int j=0; j<nodes; j++)
     {
         if(matrix[i][j]==1)
         {
              if(unVisited (nodeid[j]))
              {
                   qadd(nodeid[j]);
              }
         }
     }
     r=qremove();
     arr[temp]=r;
     temp++;
     return 0;
}

int main()
{
    char v;
    printf("How many nodes in graph?");
    scanf("%d",&nodes);
    printf("Enter  the names of nodes one by one: \n");
    for(int i=0; i<nodes; i++)
        {
            scanf("%s",&nodeid[i]);
        }
printf("Enter the adjacency matrix. Edge present=1, else 0\n");
    for(int i=0;i<nodes; i++)
    {
         for(int j=0; j<nodes; j++)
         {
              printf("matrix[%c][%c]= ", nodeid[i], nodeid[j]);
              scanf("%d", &v);
              matrix[i][j]=v;
         }
    }
   for(int i=0;i<nodes;i++)
        bfs(i);
    outputBFS();
}
```

Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
How many nodes in graph?5
Enter  the names of nodes one by one:
A
B
C
D
E
Enter the adjacency matrix. Edge present=1, else 0
```



```
matrix[A][A]= 0
matrix[A][B]= 1
matrix[A][C]= 1
matrix[A][D]= 0
matrix[A][E]= 1
matrix[B][A]= 1
matrix[B][B]= 0
matrix[B][C]= 1
matrix[B][D]= 1
matrix[B][E]= 0
matrix[C][A]= 1
matrix[C][B]= 1
matrix[C][C]= 0
matrix[C][D]= 1
matrix[C][E]= 1
matrix[D][A]= 0
matrix[D][B]= 1
matrix[D][C]= 1
matrix[D][D]= 0
matrix[D][E]= 1
matrix[E][A]= 1
matrix[E][B]= 0
matrix[E][C]= 1
matrix[E][D]= 1
matrix[E][E]= 0
Breadth First Traversal gives: A B C E D sh-4.3$
```

Comparison: DFS versus BFS

Table: 5.2.1: Comparison of DFS with BFS

Depth First Search Traversal	Breadth First Search Traversal
This traversal starts from a root node and visit all the adjacent nodes completely in depth before backtracking.	This traversal starts from the root node and explores the nodes level wise, thus exploring a node completely.
It may not necessarily give a shortest path in a graph.	It always gives the shortest path within a graph, thus gives an optimal solution by giving shortest path.
If a loop exists in a graph, the algorithm may go into infinite loop. Hence care should be taken while marking visited vertices.	Marking visited nodes can improve the efficiency of the algorithm, but even without doing this, the search is guaranteed to terminate.
Applications: 1. Connectivity testing 2. Spanning trees	Applications: 1. Finding Shortest path 2. Spanning tree

Self-assessment Questions

9) Sequential representation of binary tree uses___________.

a) Array with pointers
b) Single linear array
c) Two dimensional arrays
d) Three dimensional arrays

10) In the _____________traversal, we process all of a vertex's descendants before we move to an adjacent vertex.

a) Depth First
b) Breadth First
c) Path First
d) Root First

11) The data structure required for Breadth First Traversal on a graph is__________.

a) Tree
b) Stack
c) Array
d) Queue

12) The aim of BFS algorithm is to traverse the graph that are__________

a) As close as possible to the root node
b) With high depth
c) With large breadth
d) With large number or nodes

Summary

- Graphs are non-linear data structures. Graph is an important mathematical representation of a physical problem.
- Graphs and directed graphs are important to computer science for many real world applications from building compilers to modeling physical communication networks.
- A graph is an abstract notion of a set of nodes (vertices or points) and connection relations (edges or arcs) between them.
- The representation of graphs can be categorized as (i) sequential representation and (ii) linked representation.
- The sequential representation makes use of an array data structure whereas the linked representation of a graph makes use of a singly linked list as its fundamental data structure.
- The depth first search (DFS) and breadth first search (BFS) and are the two algorithms used for traversing and searching a node in a graph.

Terminal Questions

1. Explain graphs as a data structure.

2. Explain two different ways of sequential representation of a graph with an example.

3. Explain the linked representation of an undirected and directed graph.

4. Which are the two standard ways of traversing a graph? Explain them with an example of each.

5. Consider the following specification of a graph G,

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

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

   a) Draw an undirected graph.

   b) Draw its adjacency matrix.

Answer Keys

Self-assessment Questions	
Question No.	Answer
1	c
2	d
3	a
4	a
5	a
6	b
7	b
8	c
9	b
10	a
11	d
12	a

Activity

Activity Type: Offline **Duration:** 30 Minutes

Description:

Ask the students to solve given problem:

Consider the graph G with vertices V ={1, 2, 3, 4} and edges
E={(1,2),(2,3),(3,4),(4,1),(2,1),(2,4)}.

- For every vertex u, find its indegree in (u) and its out degree out (u).
- What is the value of the following sum for this graph?

Case study

Study of different applications of Graphs

Since they are powerful abstractions, graphs can be very important in modelling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. Transportation networks. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

2. Utility graphs. The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analysing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

3. Document link graphs. The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyse relevance of web pages, the best sources of information, and good link sites.

4. Protein-protein interactions graphs. Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.

5. Network packet traffic graphs. Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analysing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

6. Scene graphs. In graphics and computer games scene graphs represent the logical or special relationships between objects in a scene. Such graphs are very important in the computer games industry.

7. Finite element meshes. In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements form a graph that is called a finite element mesh.

8. Robot planning. Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

9. Neural networks. Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.

10. Graphs in quantum field theory. Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyse path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).

11. Semantic networks. Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.

12. Graphs in epidemiology. Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analysing such graphs has become an important component in understanding and controlling the spread of diseases.

13. Graphs in compilers. Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

Questions:

1. List down different applications of graphs from above study.

2. Explain in brief how graphs can be used in computer networks.

3. Can you thinks of any additional application of graph to solve real world problem.

Bibliography

e-Reference

- courses.cs.vt.edu, (2016). Graph Traversals .Retrieved on 19 April 2016, from, http://courses.cs.vt.edu/~cs3114/Fall09/wmcquain/Notes/T20.GraphTraversals.pdf

External Resources

- Kruse, R. (2006). *Data Structures and program designing using 'C'* (2nd ed.). Pearson Education.
- Srivastava, S. K., & Srivastava, D. (2004). *Data Structures Through C in Depth* (2nd ed.). BPB Publications.
- Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C* (2nd ed.). Pearson Education.

Video Links

Topic	Link
Introduction to Graphs	https://www.youtube.com/watch?v=vfCo5A4HGKc
Graph Types and Representations	https://www.youtube.com/watch?v=VeEneWqC5a4
Graph Traversals	https://www.youtube.com/watch?v=H4_vRy4xQpc&list=PLT2H5PXNSXgM_Mqzk7bChFvB6xyuWilIa

Notes: