UNIT 3 GRAPH ALGORITHMS

Structure **Page Nos.**

3.0 INTRODUCTION

The vast majority of computer algorithm operate on data. Organsing these data in a certain way (i.e. data structure) has a significant role is design and analysis of algorithm. Graph is one such fundamental data structure. Array, linked list, stack, queue, tree, sets are other important data structures. A graph is generally used to represent connectivity information i.e. connectivity between cities for example. Graphs have been used and considered very interesting data structures with a large number of applications for example the shortest path problem. While several representations of a graph are possible, we discuss in the unit the two most common representations of a graph: adjacency matrix and adjacency list. Many graph algorithms requires visiting nodes and vertices of a graph. This kind of operation is also called traversal. You must have read various traversal methods for tree such as preorder, postorder and inorder In this unit we present two graph traversal algorithms which are called as Depth first search and Breadth first search algorithm.

3.1 OBJECTIVES

After going through this unit you will be able to

- define a graph,
- differentiate between an undirected and a directed graph,
- represent a graph though a adjacency matrix and an adjacency list and;
- traverse a graph using DFS and BFS.

3.2 BASIC DEFINITION AND TERMINOLOGIES

A graph G = (V. E) is a set of vertices V, with edges connecting some of the vertices (edge set E). An edge between vertex u and v is denoted as (u, v). There are two types of a graph: (1) undirected a graph and directed graph (digraph). In a undirected graph the edges have no direction whereas in a digraph all edges have direction.

You can notice that edges have no direction. Let us have an example of an undirected graph (figure 1) and a directed graph (figure 2)

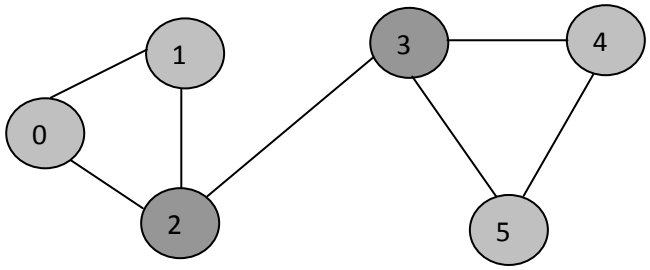


Figure.1 Undirected graph

V = {0, 1, 2, 3, 4, 5}

E = {(0, 1) , (0, 2),

(1,2), or (2, 2) both are same

(2, 3),

(3, 4), (3, 5)

(4, 5)

}

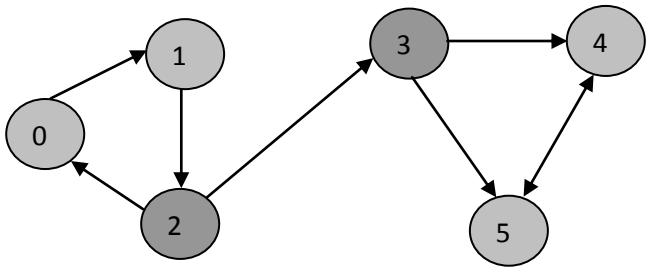


Figure 2: Diagraph

V = {0, 1, 2, 3, 4, 5, }

E = { (0, 1),

(1, 2)

(2, 0), (2, 3),

(3, 4), (3, 5)

(4, 5) and (5, 4) are not the same. These are two different edges.

(5, 4)

You can notice in Figure 2 that edges have direction

You should also consider the following graph preparations.

The geometry of drawing has no particular meaning: edges of a graph can be drawn "straight" or "curved".

A vertex v is adjacent to vertex u, if there is an edge (u, v). In an undirected graph, existence of edge (u, v) means both u and v are adjacent to each other. In a digraph, existence of edge (u, v) does not mean u is adjacent to v.

PATH

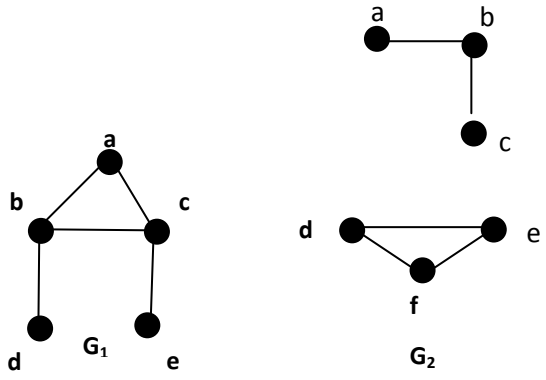
An edge may not have a weight. A path in a graph is sequence of vertices V_1 V_2….V_n such that consecutive vertices V_i V_{i+1} have an edge between them, i.e., V_{i+1} is adjacent to V_i

A path in a graph is simple if all vertices are distinct i.e. no repetition of a path of any vertices (and therefore edges) in the sequence, except possibly the first and the last one. Length of a path is the number of edges in the path. A cycle is a path of length at least 1 such that the first and the last vertices are equal. A cycle is a simple path with the same vertex as the first and the last vertex in the sequence if the path is simple. For undirected graph, we require a cycle to have distinct edges. Length of a cycle is the number of edges in the cycle.

There are many problems in computer science such as of route with minimum time and diagnostic: minimum shortcut path routing, traveling sales problem etc. can be designed using paths obtained by marking traversal along the edges of a graph.

CONNECTED GRAPHS

Connectivity: A graph is connected if there is a path from every vertex to every other vertex. In an undirected graph, if there is a path between every pair of distinct vertices of the graph, then the undirected graph is connected. The following example illustrates this:



(a) Connected **(b) Unconnected**

Figure 3: The connected and unconnected undirected graph

In the above example G, there is a path between every pair of distinct vertices of the graph, therefore G_1 is connected. However the graph G_2 is not connected.

In directed graph, two vertices are strongly connected if there is a (directed) path from one to the other.

Undirected: Two vertices are connected if there is a path that includes them.

Directed: Two vertices are strongly-connected if there is a (directed) path from any vertex to any other.

3.3 GRAPH REPRESENTATION

In this section, we will study the two more important data structure for graph representation: Adjacency matrix and Adjacency list.

3.3.1 ADJACENCY MATRIX

The adjacency matrix of a graph G = {V, E} with n vertices is a *n* x *n* boolean/matrix. In this matrix the entry in the *i*th row and *j*th column is 1 if there is an edge from the *i*th vertex to the *j*th vertex in the graph G. If there is no such edge, then the entry will be zero. It is to be noted that
(i) the adjacency matrix of a undirected graph is always symmetric, i.e., M [*i*,*j*] = M [*j*, *i*]

(ii) The adjacency matrix for a directed graph need not be symmetric.

(iii) The memory requirement of an adjacency matrix is n^2 bits

For example for the graph in the following figure (a) is adjacency matrix is given in (b)

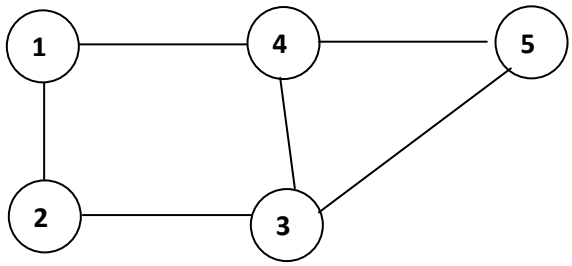


Figure. 4 (a)

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	1	1	0	1	1
	0	0	1	0	1
	0	0	1	1	0

Figure.4 (b) Adjacency Matrix

Let us answer the following questions:

(i) Suppose if we want to know how much time will take in finding number of edges a graph with *n* vertices?

Since the space needed to represent a graph is n^2 bits where *n* is a number of vertices. All algorithm will require at least 0 (n^2) time because $n^2 - n$ entries of the matrix have to be examined. Diagonal entries are zero.

(ii) Suppose the most of the entries in the adjacency matrix are zeros, i.e., when a graph is a sparse... How much time is needed to the find *m* number of edges in a graph? It will take much less time if say 0 ($e + n$), where *e* is the number of edges is a graph and $e << n^2/2$. But this can be achieved if a graph is represented through an adjacency list where only the edges will be represented.

3.3.2 ADJACENCY LIST

The adjacency list of a graph or a diagraph is a set of linked lists, one linked list for each vertex. The nodes in the linked list *i* contain all the vertices that are adjacent to vertex *i* of the list (i.e. all the vertices connected to it by an edge). The following figure.5 represents adjacency list of the graph in figure 4 (a).

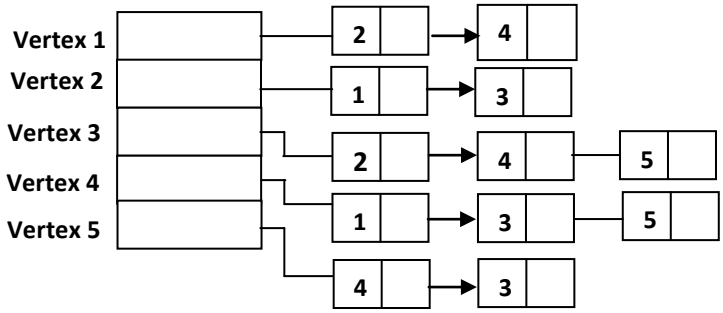


Figure. 5 Adjacency List

Putting it in another way of an adjacency list represents only columns of the adjacency matrix for a given vertex that contains entries as 1's. It is to be observed that adjacency list compared to adjacency matrix consumes less memory space if a graph is sparse. A graph with few edges is called sparse graph. If the graph is dense, the situation is reverse. A dense graph, is a graph will relatively few missing edges. In case of an undirected graph with *n* vertices and e edge adjacency list requires *n* head and 2 *e* list nodes (i.e. each edges is represented twice).

What is the storage requirement (in terms of bits) for a adjacency list of any graph!

(i) For storing *n* (*n* vertices) head nodes – we require – $\log_2 n$ bits –

(ii) For storing list nodes for each head *n* nodes – we require $\log n + \log e$

Therefore total storage requirement in item of bits for adjacency matrix is $^2log_2 n$ ($^2log_2{}^n + log_2{}^e$)

Question. What is time complexity in determining number of edges in an undirected graph.

It may be done in just 0 ($n + e$) because in degree of any vertex (i.e. number of edges incident to that vertex) in an undirected graph may be determined by just counting the number of nodes in its adjacency list.

Use of adjacency matrix or adjacency list for representing your graph – depends upon the type of a problem; type of algorithm to be used for solving a problem and types of a input graph (dense or sparse)

3.4 GRAPH TRAVERSAL ALGORITHMS

3.4.1 DEPTH-FIRST SEARCH

You are aware of tree traversal mechanism. Give a tree, you can traverse it using preorder, inorder and postorder. Similarly given an undirected graph you can traverse it or visit its nodes using breadth first-search and depth-first search.

Searching in breadth-first search or depth first search means exploring a given graph. Through searching a graph one can find out whether a graph is connected or not? There are many more applications of graph searching algorithms. In this section we will illustrate Depth First Search algorithm followed by Breadth first Search algorithm in the next section.

The logic behind this algorithm is to go as far as possible from the given starting node searching for the target. In case, we get a node that has no adjacent/successor node, we get back (recursively) and continue with the last vertex that is still not visited.

Broadly it is divided into 3 steps:

- Take a vertex that is not visited yet and mark it visited
- Go to its first adjacent non-visited (successor) vertex and mark it visited
- If all the adjacent vertices (successors) of the considered vertex are already visited or it doesn't have any more adjacent vertex (successor) – go back to its parent vertex

Before starting with an algorithm, let us discuss the terminology and structure used in the algorithm. The following algorithm works for undirected graph and directed graph both.

The following color scheme is to maintain the status of vertex i.e mark a vertex is visited or unvisited or target vertex:

white- for an undiscovered/unvisited vertex

gray - for a discovered/visited vertex

black - for a finished/target vertex

The structure given below is used in the algorithm.

p[u]- Predecessor or parent node.

Two (2) timestamps referred as

t[u] – First time discovering/visiting a vertex, store a counter or number of times

f[u]= finish off / target vertex

Let us write the algorithm DFS for any given graph G. In graph G, V is the vertex set and E is the set of edges written as G(V,E). Adjacency list for the given graph G is stored in Adj array as described in the previous section.

color[] - An array color will have status of vertex as white or gray or black as defined earlier in this section.

```
DFS(G)
  {
     for each v in V,                //for loop  V+1 times
     {
          color[v]=white; // V times
           p[v]=NULL;                // V times
     }
     time=0;                         // constant time O(1)
    for each u in V,                 //for loop V+1 times
       if (color[u]==white)          // V times
                DFSVISIT(u) // call to DFSVISIT(v) , at most V times O(V)

  }

DFSVISIT(u)
  {
     color[u]=gray;                  // constant time
     t[u] = ++time;
     for each v in Adj(u)            // for loop
        if (color[v] == white)
           {
              p[v] = u;
              DFSVISIT(v);      // call to DFSVISIT(v)
           }
     color[u] = black;               // constant time
     f[u]=++time;                    // constant time
}
```

Complexity analysis

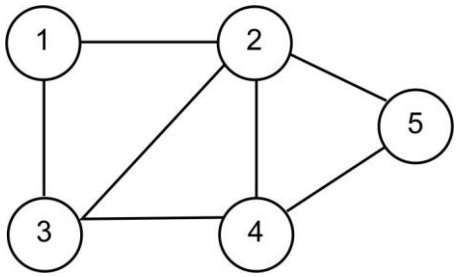
In the above algorithm, there is only one DFSVISIT(u) call for each vertex u in the vertex set V. Initialization complexity in DFS(G) for loop is O(V). In second for loop of DFS(G) , complexity is O(V) if we leave the call of DFSVISIT(u).

Now, Let us find the complexity of function DFSVISIT(u)

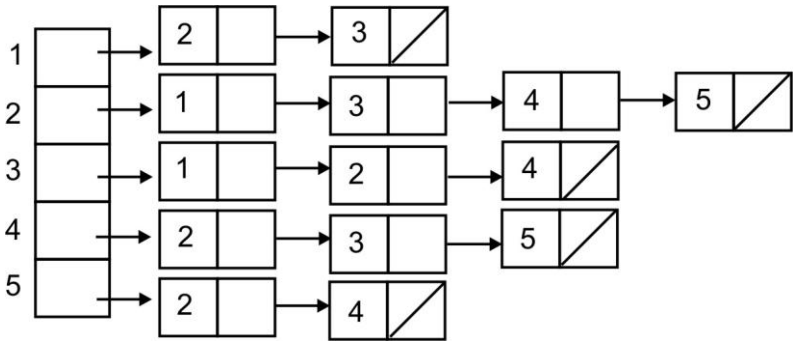
The complexity of for loop will be O(deg(u)+1) if we do not consider the recursive call to DFSVISIT(v). For recursive call to DFSVISIT(v), (complexity will be O(E) as Recursive call to DFSVISIT(v) will be at most the sum of degree of adjacency for all vertex v in the vertex set V. It can be written as $\sum_{v \in V} |Adj(v)| = O(E)$

Hence, overall complexity for DFS algorithm is O(V + E)

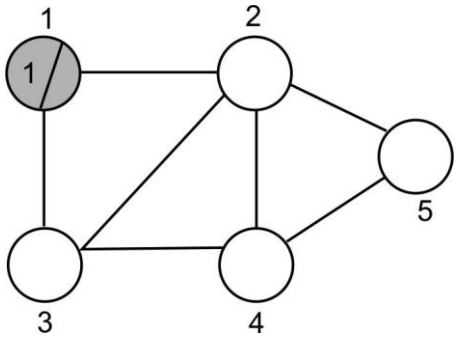
The strategy of the DFS is to search "deeper" in the graph whenever possible.
Exploration of vertex is in the fashion that first it goes deeper then widened.
Let us take up an example to see how exploration of vertex takes place by Depth First Search algorithm.

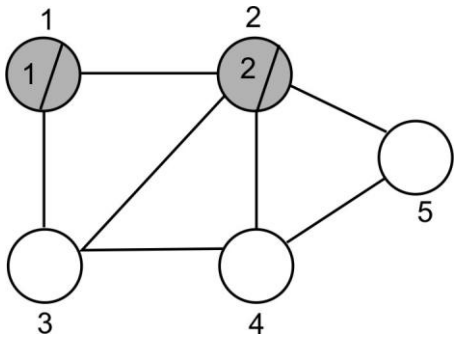


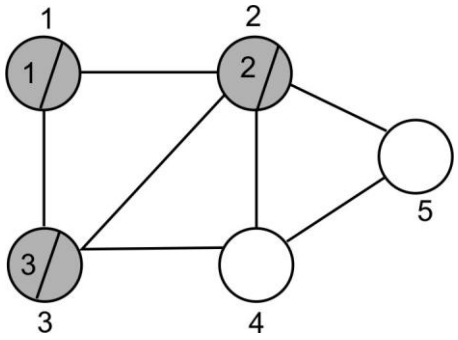
Adjacency list of the above graph is as below:

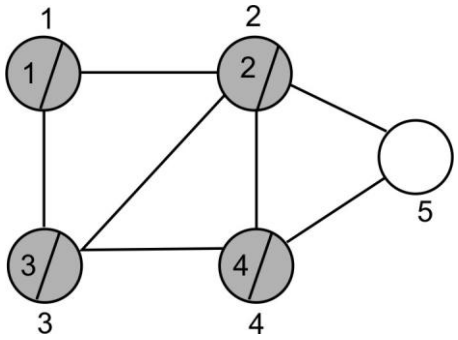


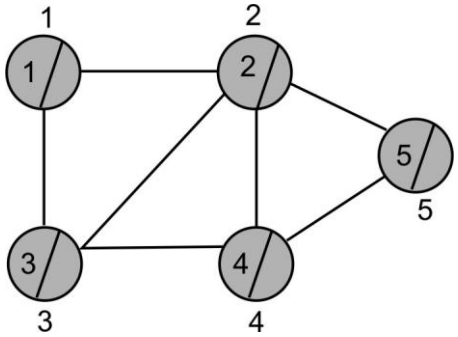
Let us explore the vertices of the graph using DFS algorithm.

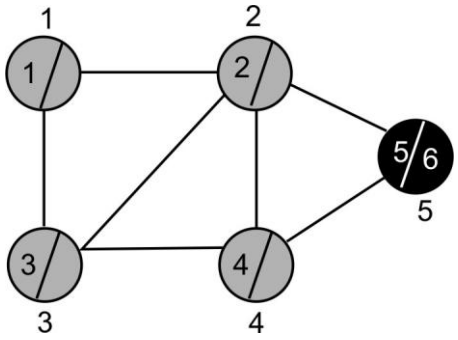


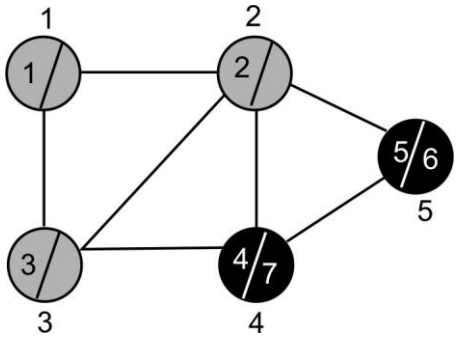


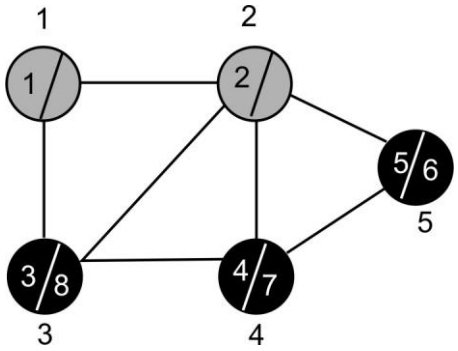

1
2
1
2
5
3
3
4

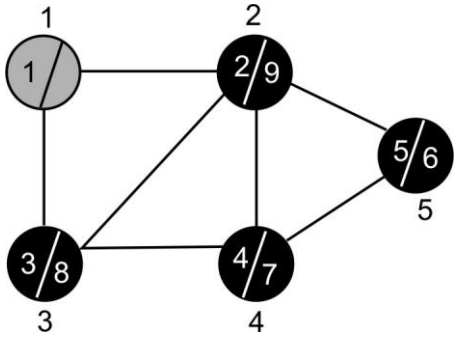

1
2
1
2
5
3
4
3
4

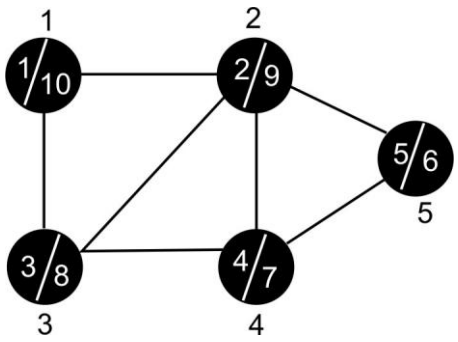

1
2
1
2
5
5
3
4
3
4


1
2
1
2
5/6
5
3
4
3
4

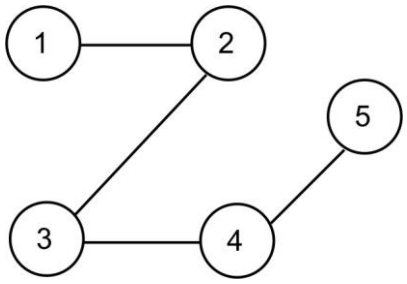








Now each vertex of the given graph is visited/explored by DFS algorithm and DFS tree is as follows:



Data structure used for implementing DFS algorithm is stack. In the diagram along with each vertex start and finish time is written in the format a/b here a represent start time and b represent finish time. This will result in to tree or forest. The order of vertices explored by DFS algorithm according to adjacency list considered for given graph is 1,2,3,4,5.

3.4.2 BREADTH-FIRST SEARCH

In this section, we will discuss breadth first search algorithm for graph. This is very well known searching algorithm. A traversal depends both on the starting vertex, and on the order of traversing the adjacent vertices of each node. The analogy behind breadth first search is that it explores the graph wider then deeper. The method starts with a vertex v then visit all its adjacent nodes v1,v2,v3…then move to the next node which is adjacent to v1, v2, v3 …. This also referred as level by level search.

Basic steps towards exploring a graph using breadth-first search:

- Mark all vertices as "unvisited".
- Start with start vertex v
- Find an unvisited vertex that are adjacent to v , mark them visited
- Next consider all recently visited vertices and visit unvisited vertices adjacent to them
- Continue this process till all vertices in the graph are explored /visited

Now, let us see the structure used in this algorithm and color scheme for status of vertex.

Color scheme is same as used in DFS algorithm i.e to maintain the status of vertex i.e mark a vertex is visited or unvisited or target vertex:

white- for an undiscovered/unvisited vertex

gray - for a discovered/visited vertex

black - for a finished/target vertex

The structure given below is used in the algorithm. G will be the graph as G(V,E) with set of vertex V and set of edges E.

p[v]-The parent or predecessor of vertex

d[v]-the number of edges on the path from *s* to *v*.

Data structure used for breadth-first search is queue, Q (FIFO), to store gray vertices.

color[v]- This array will keep the status of vertex as white, grey or black

The following algorithm for BFS takes input graph G(V,E) where V is set of vertex and E is the set of edges. Graph is represented by adjacency list i.e Adj[]. Start vertex is s in V.

```
Line    BFS(G,s)
No.     {
1.              for each v in V  - {s}                  // for loop
                {
2.                      color[v]=white;
3.                      d[v]= INFINITY;
4.                       p[v]=NULL;
                }
5.              color[s] = gray;
6.              d[s]=0;
7.              p[s]=NULL;
8.              Q=∅;            // Initialize queue is empty
9.              Enqueue(Q,s); /* Insert start vertex s in Queue Q */
10.             while Q is nonempty             // while loop
               {
11.                     u = Dequeue[Q];  /* Remove an element from Queue Q*/

12.                     for each v in Adj[u]            // for loop
                        {
13.                       if (color[v]  == white) /*if v is unvisted*/
                         {
14.                           color[v] = gray;           /* v is visted */
15.                           d[v]  = d[u] + 1;     /*Set distance of v to no. of edges
        from s to u*/
16.                              p[v]  = u;             /*Set parent of v*/
17.                           Enqueue(Q,v);      /*Insert  v in Queue Q*/
                         }
                        }
18.                     color[u]  = black;          /*finally visted or explored vertex
u*/
                }
        }
```

Complexity Analysis

In this algorithm first for loop executes at most O(V) times.

While loop executes at most O(V) times as every vertex v in V is enqueued only once in the Queue Q. Every vertex is enqueued once and dequeued once so queuing will take at most O(V) time.

Inside while loop, there is for loop which will execute at most O(E) times as it will be at most the sum of degree of adjacency for all vertex v in the vertex set V.

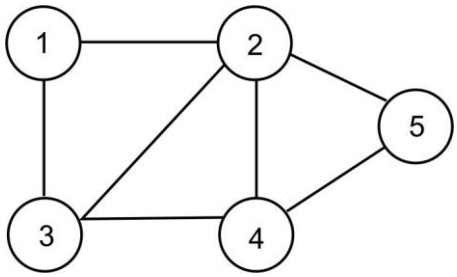
Which can be written as $\sum_{v \varepsilon V} |Adj(v)| = O(E)$

Let us summarize the number of times a statement will execute in the algorithm for BFS.

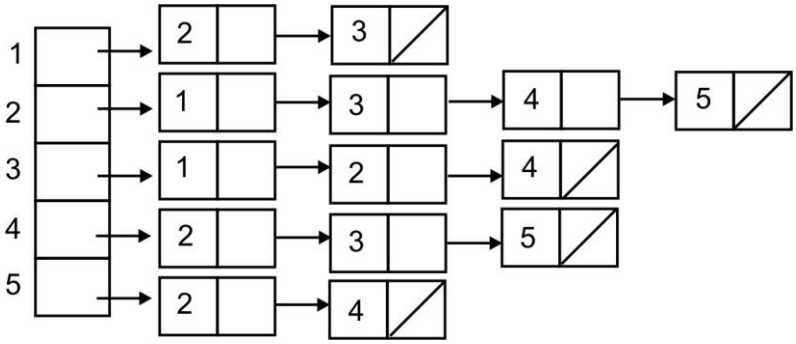
Line no.	No. of times statement will execute	Cost
1	V	O(V)
2	V-1	
3	V-1	
4	V-1	O(1)
5	1	
6	1	
7	1	
8	1	
9	1	
10	V+1	O(V)
11	V	
12	V+E+1	O(V+E)
13	V+E	
14	V	
15	V	
16	V	
17	V	
18	V	

Thus overall complexity of BFS will be V + V + E, i.e. O(V+E)

Let us take up an example to see how exploration of vertex takes place by Breadth First Search algorithm.



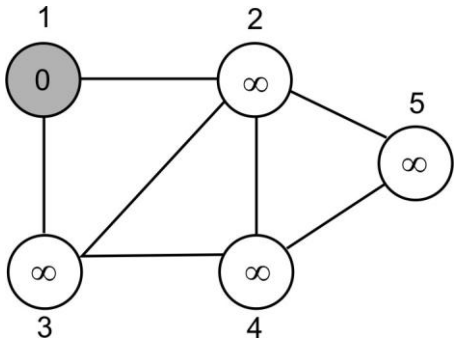
The adjacency list of the above graph is as below:

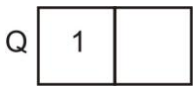


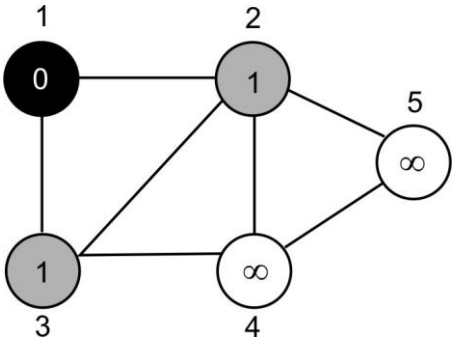
Let us explore vertices of the graph by BFS algorithm.

Consider initial vertex as vertex 1.

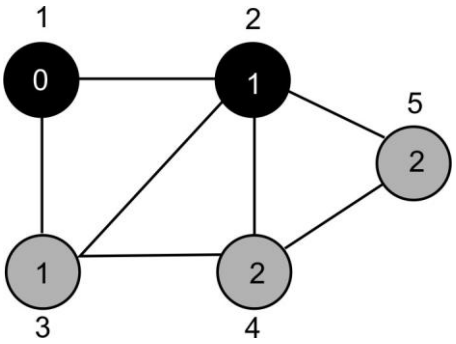
Initial status of the Queue is Q = Ø



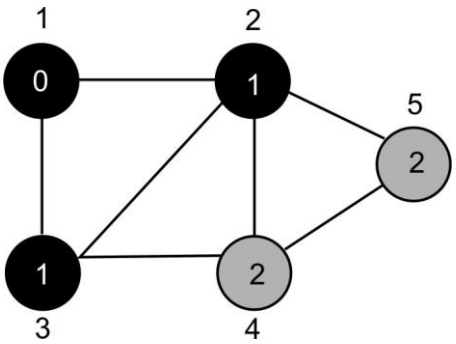



1
2
0
1
5
∞
1
∞
3
4

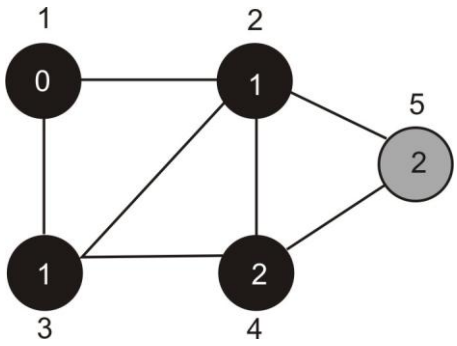

Q
2
3

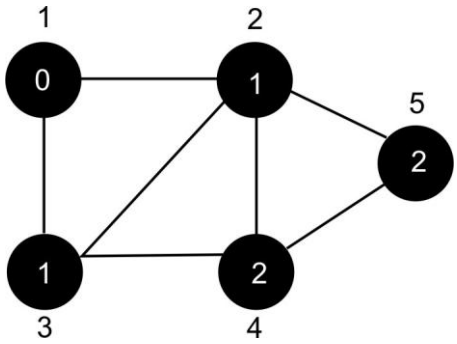

1
2
0
1
5
2
1
2
3
4


Q
3
4
5


1
2
0
1
5
2
1
2
3
4


Q
4
5





Q = Ø

After exploring the vertex of given graph by BFS algorithm, BFS traversal sequence is

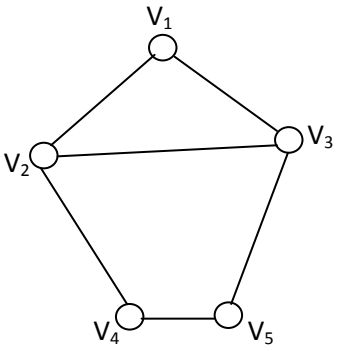
1, 2, 3, 4, 5

In this algorithm sequence of vertex visited or explored may vary. The final sequence of vertex visited is dependent on adjacency list. But the array d[] will have same number irrespective of order of vertices in adjacency list. In the above diagram distance is shown along the vertex. According to adjacency list drawn in the diagram, exploration sequence of vertex by BFS algorithm is 1,2,3,4,5.

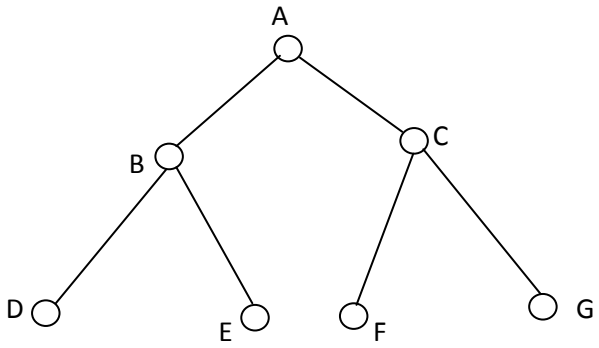
☞ Check Your Progress 1

1. What is the complexity of graph search algorithms if graph is represented by adjacency matrix and adjacency list?
2. Enlist few applications where DFS and BFS can be used?

3. Consider a graph with 5 vertices and 6 edges. Write its adjacency matrix and adjacency list.



4. For the following graph write DFS and BFS traversal sequence.



3.5 SUMMARY

A graph G(V,E) where V is the finite set of vertices i.e { v1,v2,v3….} and E is the finite set of edges {(u,v),(w,x)….}. Graph is known as directed graph if the each edge in the graph has ordered pair of vertices i.e (u,v) means an edge from u to v. In Undirected graph each edge is unordered pair of vertices i.e (u,v) and (v,u) refers to the same edge. A graph can be represented by adjacency matrix and adjacency list. In adjacency, list memory requirement is more as compared to adjacency list representation. Graph searching problem has wide range of applications. Breadth First search and Depth first search are very well known searching algorithms. In breadth first search, exploration of vertex is wider first then deeper. In depth first search it is deeper first and then it is widened. By exploration of vertex in any search algorithm, implies visiting or traversing each vertex in the graph. Data structure used for Breadth first search is queue and depth first search is stack. By using these search algorithms, connected components of graph can be found. Breadth first search method, gives shortest path between two vertices u and v. Depth first search is used in topological sorting. There are many more applications where these searching algorithms are used.

3.6 SOLUTIONS/ANSWERS

Check Your Progress 1

1. For BFS algorithm complexity will be as follows:
   Adjacency Matrix – $O(V^2)$
   Adjacency List – $O(V+E)$

For DFS algorithm complexity is as follows:
Adjacency Matrix – $O(V^2)$
Adjacency List – O(V+E)

2. Application where DFS can be used:
   - Finding connected component of the graph
   - Finding shortest path between two vertices

   Application where BFS can be used:

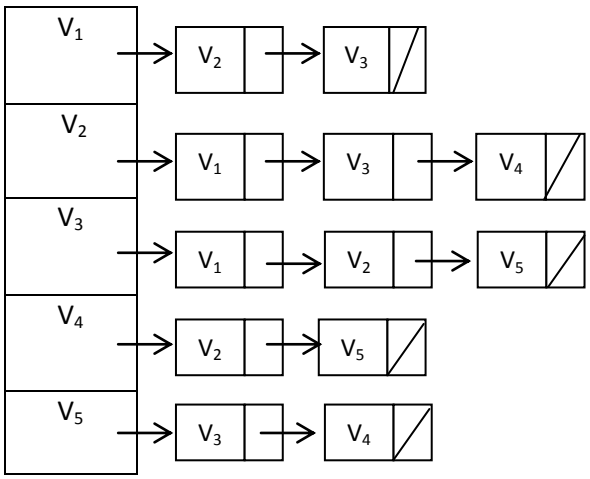
   - Finding connected component of the graph
   - Topological sorting
   - For finding cycle existence in the graph or not

3.

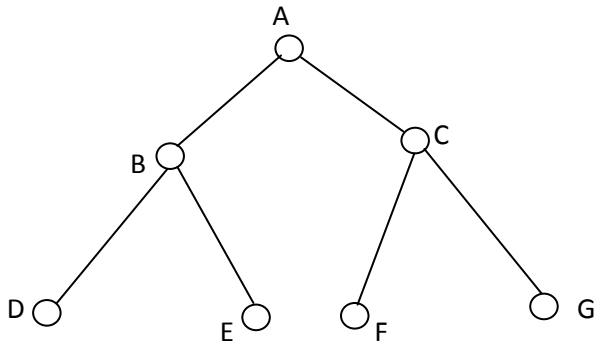
Adjacency Matrix

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	1	0	0
V_2	1	0	1	1	0
V_3	1	1	0	0	1
V_4	0	1	0	0	1
V_5	0	0	1	1	0

Adjacency List



4. For the given graph



BFS traversal sequence is A B C D E F G

DFS traversal sequence is A B D E C F G

3.7 FURTHER READINGS

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, Clifford Stein, "Introduction to Algorithms", 2 nd Ed., PHI, 2004.
2. Robert Sedgewick, "Algorithms in C", , Pearson Education, 3rd Edition 2004
3. Ellis Horowitz, Sartaj Sahani, Sanguthevar Rajasekaran, "Fundamentals of Computer algorithms", 2nd Edition, Universities Press, 2008
4. Anany Levitin, "Introduction to the Design and Analysis of Algorithm", Pearson Education, 2003.