12

GRAPHS

As we have seen in *tree* data structures, one parent node may have no child, one child, or many child nodes. Therefore, the *tree* data structure represents one-to-many relationships. In real life, however, we frequently come across problems that can be best described by many-to-many relationships. Such problems cannot be solved using trees or other data structures we have studied so far. To deal with such problems, the graph data structure is used. Note that the tree is also a graph.

12.1 GRAPHS

Graph[1] is a non-linear data structure. It consists of a finite set of points or nodes and a set of links that connects the points. The points are called *vertices* and the links that connect the vertices are called *edges*. In a graph, the vertices are represented by small circles, while edges are represented by lines or arcs. The vertices of a graph can be used to represent objects and the edges represent the relationships between objects. For example, the vertices might represent cities and the edges represent roads that link different cities.

A set of vertices $\{v_1, v_2, .., v_n\}$ is denoted by a letter V, while a set of edges $\{e_1, e_2, .., e_n\}$ is denoted by a letter E. An edge is a pair of vertices, which is represented as:

$$e = (v_i, v_j)$$

Where 'v_i' and 'v_j' denote the start vertex and end vertex of edge 'e', respectively. They are known as the *head point* and *tail point* of the edge 'e'. They are also known as *end vertices* or *endpoints* of the edge 'e'.

Generally, a graph G is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.

Consider the following graph as shown in figure 12.1:

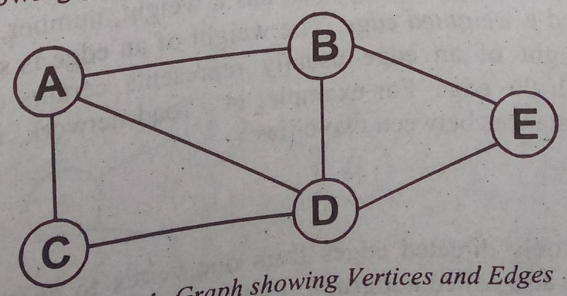


Figure 12.1: Graph showing Vertices and Edges

[1] A graph is a pictorial representation of a set of objects where some pairs are connected by links.

In figure 12.1, the labeled circles represent vertices. Thus, A, B, C, D, and E are vertices, whereas the lines from A to B, A to D, A to C, and so on represent edges. This graph G consists of 5 vertices and 7 edges. It can be defined as:

G = (V, E)

Where

V = {A, B, C, D, E}

E = {(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)}

12.1.1 Basic Terminologies Related to Graphs

The basic terminologies related to graphs are as follows:

1- Vertex

An individual data element (or node) of a graph is called the *vertex*. In a graph, as shown in figure 12.1, the labeled small circles represent vertices. Thus, A, B, C, D, and E are vertices of this graph.

2- Edge

A line that connects vertices in a graph is called an *edge*. For example, in the graph, as shown in figure 12.1, the line from A to B represents an *edge*. There are three types of edges.

- **Directed Edge:** A type of edge that has an arrow sign at its one end is called a *directed edge*. It means that a directed edge shows a direction from one vertex to another. It is a unidirectional edge. If there is a directed edge between vertices A and B, then edge (A, B) is not equal to the edge (B, A).
- **Undirected Edge**: A type of edge that has no arrow is called an *undirected edge*. It means that the undirected edge shows no direction from one vertex to another. It is a bidirectional edge. If there is an undirected edge between vertices A and B, then edge (A, B) is equal to the edge (B, A).
- **Weighted Edge**: A type of edge that has a weight, number, or any value associated with it is called a *weighted edge*. The weight of an edge is sometimes also called its *cost*. The weight of an edge usually represents certain conditions or situations associated with the edge. For example, in a road network, the weight of each road may denote distances between two cities.

3- Directed Graph

A graph that has only directed edges from one vertex to another is called a *directed graph*. A directed graph is also called a *digraph*. Figure 12.2 shows a directed graph. This graph consists of four vertices numbered 1, 2, 3, 4, and four directed edges joining vertices 1 to 2, 2 to 4, 4 to 3, and 3 to 1.

The directed graphs are commonly used to analyze electrical circuits, develop project schedules, find shortest routes, analyze social relationships, and construct models for analysis and solution of many other problems.

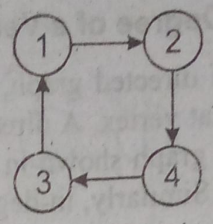


Figure 12.2: Directed graph

4- Undirected Graph

A graph that has only undirected edges is called an *undirected graph*. An undirected graph is also called an *undigraph*. Figure 12.3 shows an undirected graph.

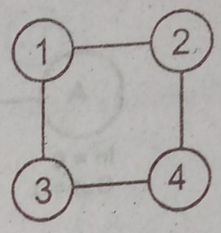


Figure 12.3: Undirected graph

Some graphs have both directed and undirected edges. These graphs are called *mixed graphs*.

5- Weighted Graph

A graph that has only weighted edges is called a *weighted graph*. Figure 12.4 shows a weighted graph. An edge of a weighted graph is represented as:

$$e = [v_i, v_j, w]$$

Where 'v_i' and 'v_j' denote the start vertex and end vertex of edge 'e' respectively, whereas 'w' represents the weight of the corresponding edge.

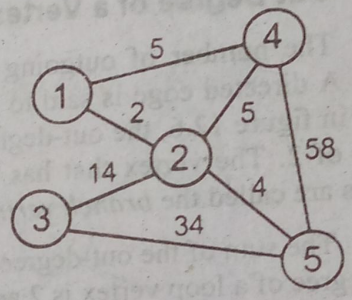


Figure 12.4: Weighted graph

6- Degree of a Vertex

The number of edges connected to a vertex is called the *degree* of that vertex. In the graph shown in figure 12.5, vertices A, B & D have a degree of 3, vertex E has a degree of 2, and vertex C has a degree of 1.

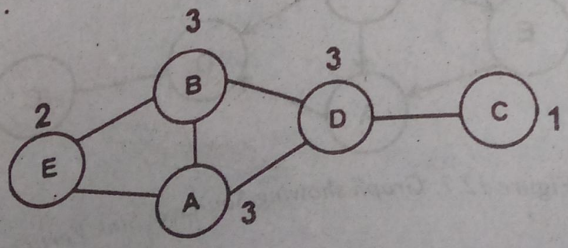


Figure 12.5: Graph showing Degree of Vertex

A vertex that has degree 0 is called an *isolated vertex*. A graph that has only one isolated vertex is called a *null graph*.

7- In-Degree of a Vertex

In a directed graph, the number of incoming edges connected to a vertex is called *in-degree* of that vertex. A directed edge is said to be an incoming edge on its destination vertex. In the directed graph shown in figure 12.6, the in-degree of vertex A is 0, and vertex B has an in-degree of 2. Similarly, in-degrees of other vertices are shown.

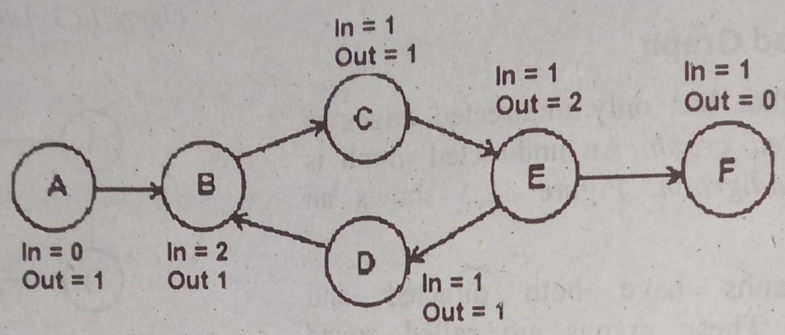


Figure 12.6: Graph showing In-Degree and Out-Degree

8- Out-Degree of a Vertex

The number of outgoing edges connected to a vertex is called the *out-degree* of that vertex. A directed edge is said to be an outgoing edge on its source vertex. In the directed graph shown in figure 12.6, the out-degree of vertices A, B, C & D is 1. Similarly, vertex E has an out-degree of 2. The vertex that has an out-degree of 0 is called *terminal vertex* or *leaf* and other vertices are called the *branch vertices*. In figure 12.6, F is the terminal vertex.

The sum of the out-degree and in-degree of a vertex is the *total degree* of that vertex. The total degree of a loop vertex is 2 and that of an isolated vertex is 0.

9- Source & Sink Vertices

The vertex that has a positive out-degree but zero in-degree is called the *source vertex*. In the graph shown in figure 12.7, vertex B is the source vertex. This vertex has a positive out-degree of 3 and its in-degree is zero.

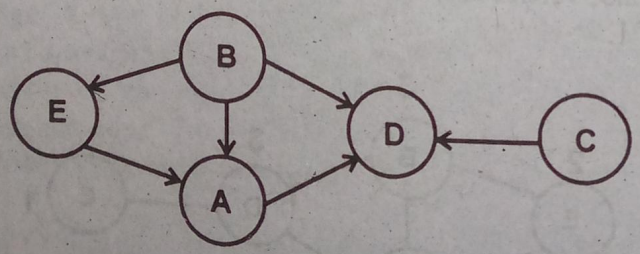


Figure 12.7: Graph showing Source & Sink Vertices

The vertex that has zero out-degree but a positive in-degree is called *sink vertex*. In the graph shown in figure 12.7, vertex D is the sink vertex. This vertex has a positive in-degree of 3 and its out-degree is zero.

10- Path & Length of a Graph

A *path* in a graph is a sequence of edges between vertices. In a graph shown in figure 12.8, ABCD represents the path from vertex A to vertex D. Similarly, AFG represents the path from vertex A to vertex G.

The number of edges in a path of a graph is called the *length of the graph*. For a path consisting of 'n' vertices, the path length will be 'n–1'.

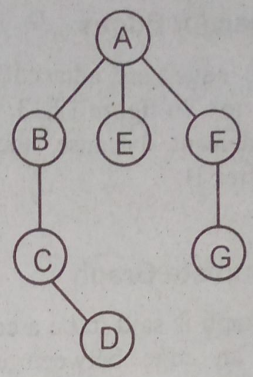


Figure 12.8: Graph showing Path

11- Loop Edge

An edge 'e' is said to be a loop edge if it has the same vertex at its tail and head. A loop edge is shown in figure 12.9.

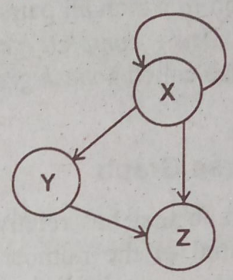


Figure 12.9: Loop Edge

12- Multiple or Parallel Edges

The edges that have the same tail and head vertices are known as multiple or *parallel edges*. Multiple edges are shown in figure 12.10.

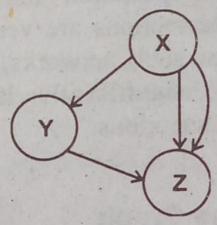


Figure 12.10: Multiple Edges

13- Simple Graph

A graph is said to be a simple graph if it does not contain any loop-edge (or self- loop).

14- Cycle & Acycle

A path that originates or begins and ends at the same vertex is called a *cycle*. In other words, a path from a particular vertex up to itself is called a *cycle*. It is also known as a *circuit*. The length of a cycle must be at least 1. In figure 12.12, an example of a cycle is given.

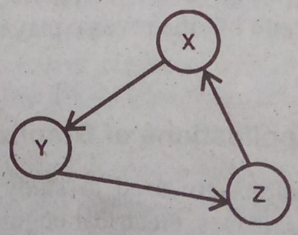


Figure 12.12: Graph showing Cycle

The directed graph that has no cycles is called an *acyclic graph*. A directed acyclic graph is also known as *DAG*.

15- Adjacent Vertices

Two vertices are adjacent if they are connected to each other by an edge. In a graph shown in figure 12.8, vertices A and B are adjacent but vertices B and E are not adjacent.

16- Adjacent Edges

Two edges are adjacent if they share the same vertex. In figure 12.13, edges AB and BC are adjacent because they share the common vertex B.

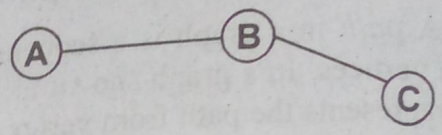


Figure 12.13: Adjacent Edges

17- Complete Graph

A graph is said to be a complete graph if there is an edge between every pair of vertices, as shown in figure 12.14. In other words, a graph in which all pairs of vertices are adjacent is called a *complete graph*. This type of graph is also called a *mesh graph*.

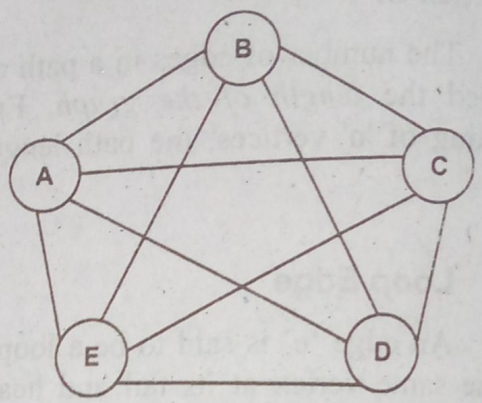


Figure 12.14: Complete Graph

18- Sparse Graph

A graph that has relatively few edges when compared to the number of vertices is called a *sparse graph*. Examples of sparse graphs are transportation and road networks where the intersections are vertices and roads are edges. For such networks, the number of roads is not significantly larger than the number of intersections.

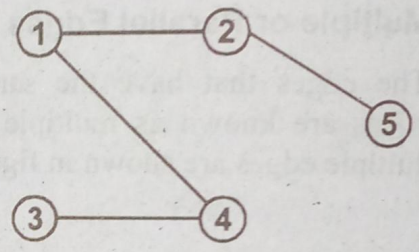


Figure 12.15: Sparse Graph

19- Dense Graph

A graph that has relatively many edges when compared to the number of vertices is called a *dense graph*. A complete graph can be a dense graph but the reverse may not be true.

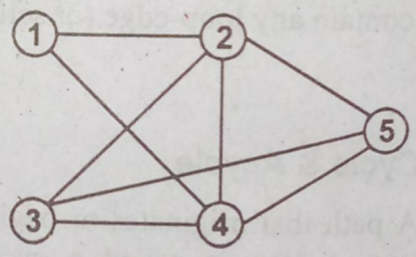


Figure 12.16: Dense Graph

12.1.2 Applications of Graphs

Graphs are used to study and modeling of real-world systems in different areas like computer sciences, electrical engineering, chemistry, physics, biology, computer chip design, and many other fields. Following are some examples where graphs are commonly used:

- ★ **Computer Networks:** Graphs are used for modeling computer networks of communication and computational devices. For instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another. Similarly, in networks' routing, servers represent vertices, data links represent edges, and connection speed represents weight.

Graphs are also used in social networks like Facebook and LinkedIn. For example, on Facebook, each person is represented with a vertex (or node). Each vertex is a structure and contains information like a person's id, name, gender, and locale, etc.

★ **Finding Path:** Graphs are used for finding a path from a starting condition to a goal condition; for example, problem-solving in artificial intelligence. Graphs are also used in GPS navigation in which road intersections represent vertices, roads represent edges, and time required to go from one intersection to another represents weight.

★ **Computer Algorithms:** Graphs are used for modeling computer algorithms, howing transitions from one program state to another.

★ **Family Tree:** Graphs are also used for modeling family trees.

★ **Modeling City Map:** A map of a city can be modeled by a weighted graph. On each street, the edge is compared with a length, corresponding to the length of the street, and direction – the direction of movement. If a street is two-way, it can be compared to two edges in both directions. At each intersection, there is a vertex. In such a model there are natural tasks such as searching for the shortest path between two intersections, checking whether there is a road between two intersections, checking for a loop (if we can turn and go back to the starting position) searching for a path with a minimum number of turns, etc.

★ **Modeling River System:** River system in a given region can be modeled by a weighted directed graph, where each river is composed of one or more edges, and each vertex represents the place where two or more rivers flow into another one. On the edges, values can be set, showing the amount of water that flows through them. Naturally, in this model, there are tasks such as calculating the volume of water, passing through each vertex, and anticipation of the possible flood where the volume of water is increasing.

★ **Transportation Network:** In transportation networks, graph vertices correspond to locations where people or goods are moved around. The locations may represent cities, warehouses, airports, and terminals. The edges correspond to the connections between the vertices. The edges may represent roads, railway tracks, air routes, etc. along which transportation in cities takes place. Figure 12.17 represents transportation links between ten cities. The vertices represent cities and the edges correspond to the roads that link the cities. The graph shown in this figure has 10 vertices and 15 edges.

★ **Chemistry and Physics:** Graph theory is also used to study molecules in chemistry and physics. In condensed matter physics, the three-dimensional structure of complicated simulated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties related to the topology of the atoms. In chemistry, a graph can show a natural model of a molecule, where vertices represent atoms, and edges represent bonds. This approach is especially used in computer processing of molecular structures, ranging from chemical editors to database searching. In statistical physics, graphs can represent local connections between interacting parts of a system, as well as the dynamics of a physical process on such systems. Similarly, in computational neurosciences, graphs can be used to represent

functional connections between brain areas that interact to give rise to various cognitive processes, where the vertices represent different areas of the brain and the edges represent the connections between those areas.

★ **Biology:** Graph theory is useful in biology and conservation efforts where a vertex can represent regions where certain species exist (or inhabit) and the edges represent migration paths or movement between different regions. This information is important when looking at breeding patterns or tracking the spread of disease, parasites, or how changes to the movement can affect other species.

★ **Mathematics:** In mathematics, graphs are useful in geometry and certain parts of topology such as knot theory. Algebraic graph theory has close links with group theory.

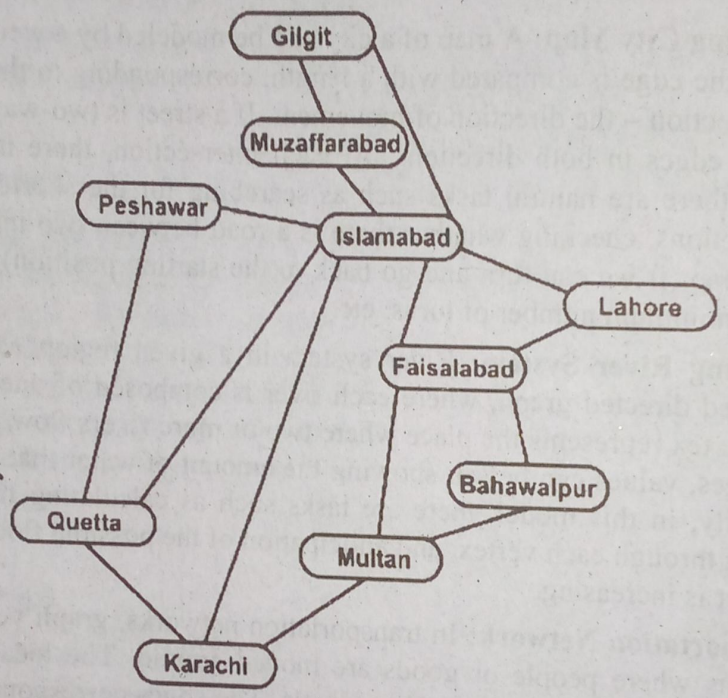


Figure 12.17: Transportation links between important cities of Pakistan

12.2 REPRESENTATION OF GRAPHS

Graphs are unstructured. One vertex in a graph might be adjacent to many other vertices. Similarly, a vertex might only be adjacent to just one vertex. This property of adjacency is used to represent graphs in computer memory. There are two common ways of representing graphs. One way uses the adjacency matrix and the other uses the adjacency list.

12.2.1 Adjacency Matrix Representation

The adjacency matrix is one of the common ways to represent a graph. It is a two-dimensional array of size V x V, where V is the number of vertices in the graph. It means that a graph of 5 vertices can be represented using a matrix (2D) of size 5x5. The adjacency matrix shows which vertices are adjacent to one another.

In the adjacency matrix, rows and columns both represent vertices. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. For directed and undirected graphs, each cell of the matrix contains value 1, if a pair of vertices is adjacent; otherwise, the entry is 0.

12.2.1.1 Representation of Undirected Graph

Consider an undirected graph that contains five vertices 1, 2, 3, 4, and 5. The vertex 1 is linked to vertex 4. The vertex 2 is linked to vertices 4 and 5. The vertex 3 is linked to vertex 5. The vertex 4 is linked to vertices 1, 2, and 5. The vertex 5 is linked to vertices 2, 3, and 4.

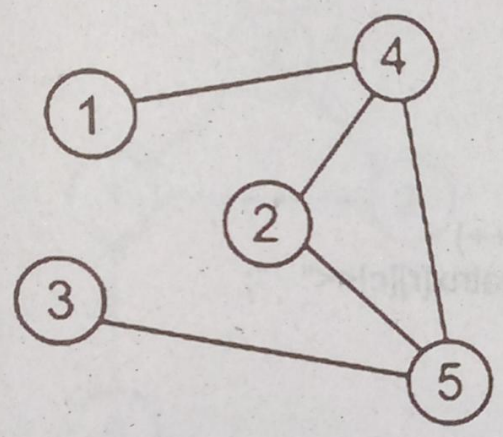
1 - [4]

2 - [4, 5]

3 - [5]

4 - [1, 2, 5]

5 - [2, 3, 4]



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

Adjacency Matrix

Program Code 12.1

```
// This program implements the above mentioned adjacency matrix representation
// of undirected graph of 5 vertices

#include <iostream.h>
```

12.2.1.2 Representation of Directed Graph

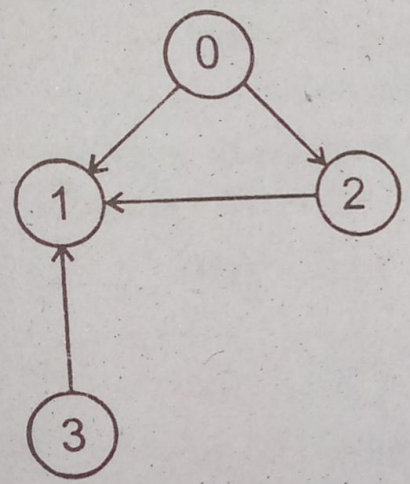
Consider a directed graph that contains four vertices 0, 1, 2, and 3. The vertex 0 is linked to vertices 1 and 2. The vertex 1 is not directly linked to any other vertex. The vertex 2 and vertex 3 are linked to vertex 1.

0 - [1, 2]

1 - [NULL]

2 - [1]

3 - [1]



	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	1	0	0
3	0	1	0	0

Adjacency Matrix

Program Code 12.2

...m implements the above mentioned adjacency matrix representation

0 1 0 0

12.2.1.3 Representation of Weighted Graph

For a weighted graph, the adjacency matrix will be populated using the weight as an entry. Consider a weighted graph that contains five vertices 0, 1, 2, 3, and 4. The vertex 0 is linked to vertices 1 and 2. The vertex 1 is linked to vertices 0, 2, and 3. The vertex 2 is linked to vertices 0, 1, and 4. The vertex 3 is linked to vertices 1 and 4. The vertex 4 is linked to vertices 2 and 3.

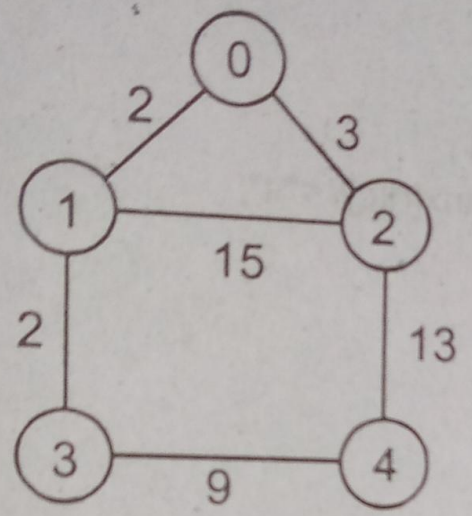
0 - [1, 2]

1 - [0, 2, 3]

2 - [0, 1, 4]

3 - [1, 4]

4 - [2, 3]



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix

0	2	0	0	9
0	0	13	9	0

12.2.1.4 Advantages and Disadvantages of Adjacency Matrix Representation

Following are the main advantages of adjacency matrix representation:

- It is very simple to implement.
- Adding or removing the time of an edge can be done in O(1) time (constant time). Similarly, the same time is required to check, if there is an edge between two vertices.
- It is very convenient and simple to program.

Following are some disadvantages of adjacency matrix representation:

- It consumes a huge amount of memory for storing big graphs. It takes $O(n^2)$ in memory.
- It requires huge efforts for adding or removing a vertex.
- In many algorithms, we need to know the edges, adjacent to the current vertex. We have to get such information from the adjacency matrix by reading the corresponding row, which results in O(|V|) complexity.

12.2.2 Adjacency List Representation

Adjacency List is one of the most common ways to represent graphs. In the adjacency list, an array of linked list is used. The size of the array is equal to the number of vertices of the graph. Each element of the array represents a specific vertex and a linked list of vertices that are adjacent to that vertex. This kind of graph representation is one of the alternatives to the adjacency matrix. It requires less amount of memory than the adjacency matrix.

Let us take an undirected graph that contains five vertices 0, 1, 2, 3, and 4. The vertex 0 is linked to vertices 1, 2, 3, and 4. The vertex 1 is linked to vertices 0 and 3. The vertex 2 is linked to vertices 0, 3, and 4. The vertex 3 is linked to vertices 0, 1, 2, and 4. The vertex 4 is linked to vertices 0, 2, and 3.

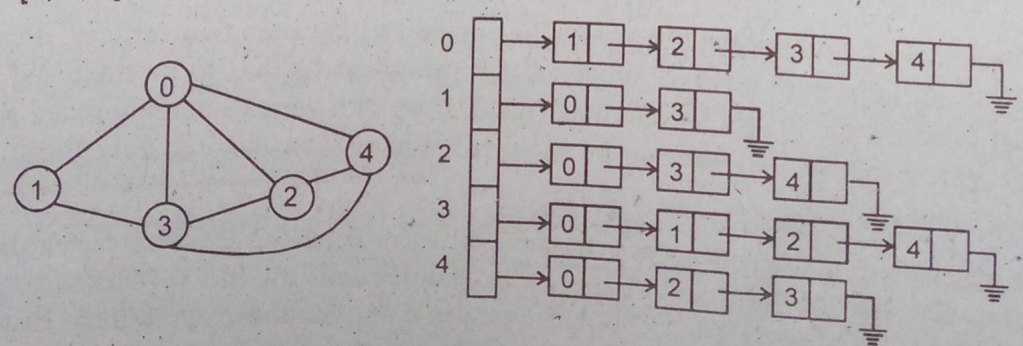
0 - [1, 2, 3, 4]

1 - [0, 3]

2 - [0, 3, 4]

3 - [0, 1, 2, 4]

4 - [0, 2, 3]



Following is the structure of the vertex or node of the graph:

12.2.2.1 Advantages and Disadvantages of Adjacency List Representation

Following are the main advantages of adjacency list representation:

- ★ It does not need to allocate list size and any number of vertices can be created dynamically whenever these are required. Therefore, adjacency list representation saves memory space and easy to maintain.
- ★ It allows to store graph in a more compact form than the adjacency matrix.
- ★ It allows to get the list of adjacent vertices in O(1) time.

Following are some disadvantages of adjacency list representation:

- ❖ Adding or removing edge to and from the adjacency list is not so easy as for adjacency matrix.
- ❖ It does not allow making an efficient implementation if dynamically change of vertices number is required. Adding a new vertex can be done in O(1), but removal results in O(E) complexity.

12.2.3 Lists vs Matrices

The representation of a graph is selected depending upon the purpose of the graph. If the graph is sparse, i.e. there are not many edges, then the matrix will take up a lot of space, but the adjacency list does not have that problem, because it only keeps track of which edges are actually in the graph. On the other hand, if there are a lot of edges in the graph, or if it is fully connected, then the list requires more space because of all of the references needed.

If it is to be found out whether or not an edge exists between two vertices, it can be found out directly in an adjacency matrix. Whereas, a long linked list has to be traversed in order to know that it is not in the graph. On the other hand, if all of a vertex's neighbors are to be looked at, then the entire matrix will have to be scanned. Whereas, in the list, only the linked list of neighbors is scanned.

In matrices, it is difficult to insert a new vertex or delete an existing vertex. For insertion and deletion, the size of the matrix has to be changed and all the vertices may have to be reordered. Thus, a lot of work has to be performed to carry out these operations. However, these operations can easily and properly managed with adjacency lists.

12.3 GRAPH TRAVERSAL

Graph traversal means visiting all the vertices or nodes of the graph. This operation on the graph is similar to the tree traversal. Graph traversal is the technique used for searching a vertex in a graph. Graph traversal is also used to decide the order of vertices to be visited in a search process. There are two standard graph traversal techniques. These are as follows:

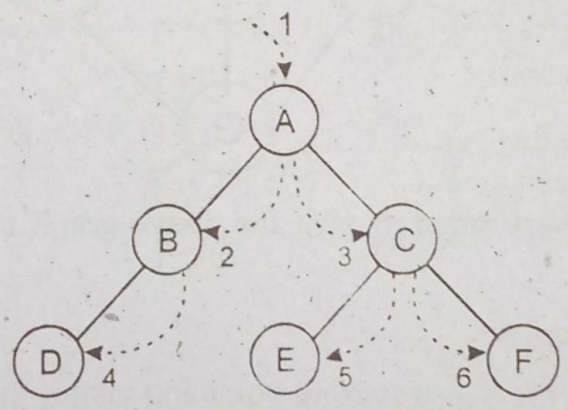
i) Breadth-First Search (BFS)

ii) Depth-First Search (DFS)

12.3.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) is the most commonly used algorithm for traversing or searching a graph. It traverses the vertices of a graph breadth-wise. Suppose the search process starts at some vertex 'v'. After visiting 'v', it visits all its neighbors (that has edges from 'v'), then unvisited neighbors of the neighbors, etc. This process continues until all the vertices in the graph are visited. Recursion is not used in BFS because traversing is done one level at a time. A queue data structure is generally used to implement the BFS algorithm.

For example, an undirected graph is given below which has 6 vertices. Suppose we start the traversal process from vertex A. After visiting this vertex, we will see its adjacent vertices. It has two adjacent vertices which are B and C. We will visit these unvisited vertices, from left to right. In this case, we will first visit vertex B and then C. Now we will visit D, then E and F. So all the vertices of the given graph have been visited.



Output: A, B, C, D, E, F

Time Complexity of BFS

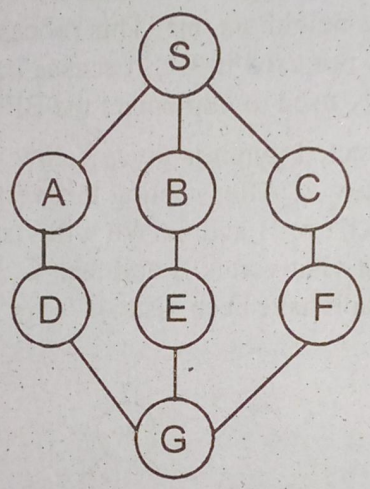
The time complexity of BFS depends on the data structure used to represent the graph. If it is an adjacency matrix, it will be $O(V^2)$, and if it is an adjacency list, it will be $O(V+E)$, where V is the number of nodes and E is the number of edges.

Algorithm - Breadth-First Search

3. Select any vertex as a starting point for traversal. Visit that vertex and insert it into the Queue.
4. Visit all the adjacent vertices of the vertex which is in front of the Queue. Insert the vertices into the queue which are not visited.
5. Check the unvisited vertices that are adjacent to the vertex at front of the Queue. If there are no unvisited vertices adjacent to that vertex then delete it from the queue.
6. Repeat steps 4 and 5 until the queue becomes empty.
7. EXIT

Example:

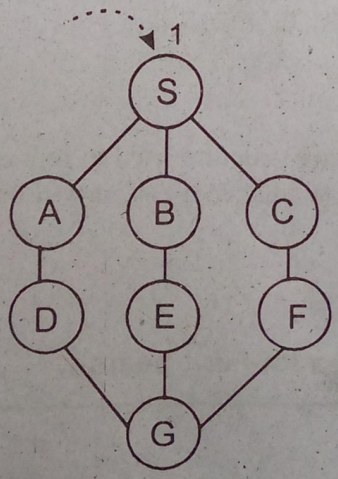
Consider the following graph:



Following steps are performed to visit the above graph using the Breadth-First Search (BFS) technique:

1- Initialize the queue.

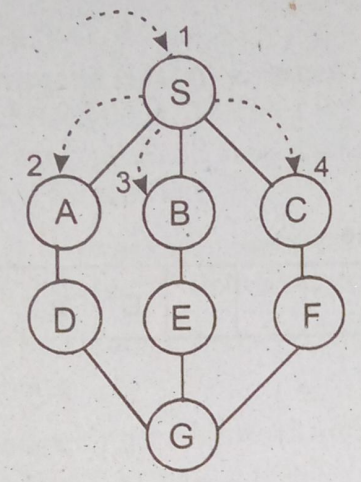
2- Select the Vertex S as a starting point and visit it. Insert S into the queue.



Queue

S							

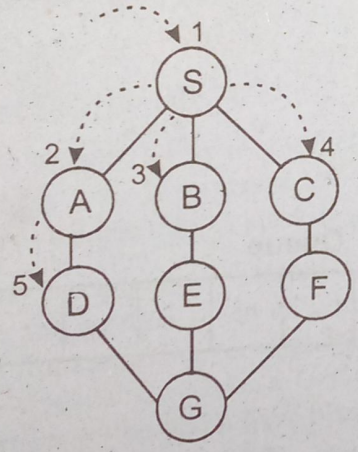
3- Visit all adjacent vertices of S which are not visited. These vertices are A, B and C. Insert these vertices into the queue and then delete S from the queue.



Queue

	A	B	C				

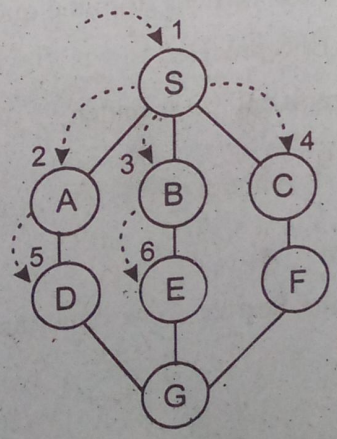
4- Visit all adjacent vertices of A which are not visited. There is only one vertex, which is D. Insert this vertex into the queue and then delete vertex A from the queue.



Queue

		B	C	D			

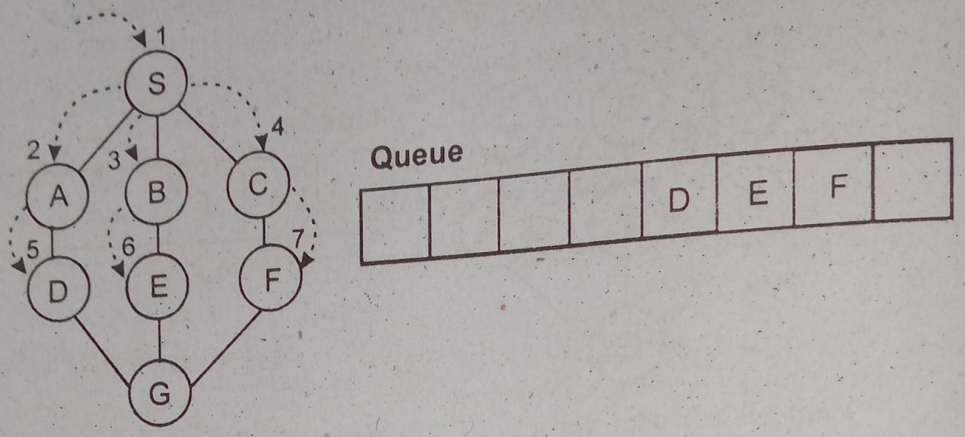
5- Visit all adjacent vertices of B which are not visited. There is only one vertex, which is E. Insert this vertex into the queue and then delete vertex B from the queue.



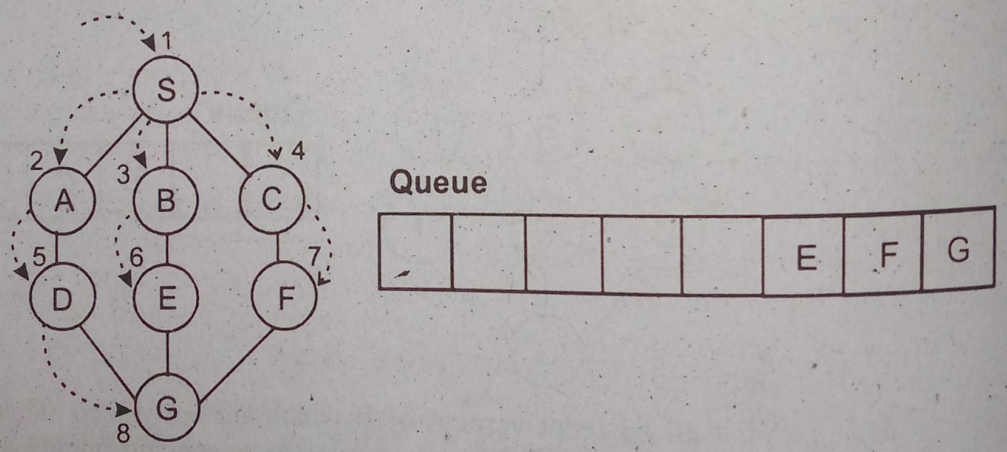
Queue

			C	D	E		

6- Visit all adjacent vertices of C which are not visited. There is only one vertex, which is F. Insert this vertex into the queue and then delete vertex C from the queue.



7- Visit all adjacent vertices of D which are not visited. There is only one vertex, which is G. Insert this vertex into the queue and then delete vertex D from the queue.



8- Now there are three entries of the vertices in the queues. These vertices have no unvisited adjacent vertices. Delete these vertices from the queue.

9- Queue became empty. So, stop the BFS process.

10- The output of BFS for the above graph will be as under:

S, A, B, C, D, E, F, G

12.3.1.1 Applications of Breadth-First Search

There are many applications of Breadth-First Search; some important applications are as follows:

1- **Finding Shortest Path:** BFS is generally used for finding the shortest path in a graph and solving puzzle games.

2- **Peer-to-Peer Networks:** In peer-to-peer networks, it is used to find all neighbor nodes.

3- **Crawlers in Search Engines:** Crawlers build an index using the Breadth-First Search technique. The idea is to start from the source page and follow all links from the source and keep doing the same. However, Depth First Search can also be used for crawlers.

3- **Social Networking Websites:** In social networks, we can find people within a given distance 'k' from a person using Breadth-First Search till 'k' levels.

4- **GPS Navigation Systems:** Breadth-First Search is used to find all neighboring locations.

5- **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth-First Search to reach all nodes.

6- **Cycle Detection in Undirected Graph:** In undirected graphs, Breadth-First Search can be used to detect a cycle. However, in a directed graph, only Depth First Search can be used.

12.3.2 Depth-First Search (DFS)

Depth First Search (DFS) is another way of traversing a graph. It is similar to the preorder traversal of a tree. It traverses the depth of any particular path before exploring its breadth. It is a recursive algorithm. A stack is generally used when implementing this algorithm.

The traversal process can be started from any vertex in the graph. Suppose a vertex 'v' is selected, processed, and marked. Then all unmarked vertices adjacent to 'v' are traversed. We go as deep as possible from neighbor to neighbor in a single path. When there is no more vertex to visit in a path, we trackback or take a U-turn and visit the next vertices. We will repeat the process until all vertices of the graph have been visited.

For example, an undirected graph is given below which has 6 vertices. Suppose we start the traversal process from vertex A. After visiting this vertex, we will see its adjacent vertices. It has two adjacent vertices which are B and C. We always select the next adjacent unvisited vertex in ascending order (or alphabetical order). In this case, we will select vertex B (not C). After visiting vertex B, we will see its adjacent vertices. It has one adjacent vertex which is D. We will visit this vertex i.e. vertex D. Now we will take a U-turn and go back to vertex A and see its next unvisited vertices which is only one i.e. vertex C. We will visit vertex C and then we will see its adjacent vertices. It has two adjacent vertices which are E and F. We will visit vertex E and then F. Now all the vertices of the given graph have been visited.

11- The vertices C, F, G, D, A, and S do not have any unvisited adjacent vertex, so we pop the Stack until it is empty.

12- Therefore, the output of DFS for the above graph will be as under:

S, A, D, G, E, B, F, C

12.3.2.1 Applications of Depth First Search

There are many applications of Depth First Search; some important applications are as follows:

1- **Path finding:** We can use the DFS algorithm to find a path between two given vertices such as V1 and V2.

2- **Topological sorting:** Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when re-computing formula values in spreadsheets, etc.

3- **Finding strongly connected components of a graph:** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.

4- **Solving puzzles with only one solution, such as mazes:** DFS can be used to find all solutions to a maze by only including vertices or nodes on the current path in the visited set.

12.3.3 Difference between BFS & DFS

Following table shows the main differences between Breadth-First Search (BFS) and Depth First Search (DFS):

Sr#	BFS	DFS
1.	BFS is useful in finding the shortest path. It finds the shortest path to the destination.	DFS is not so useful in finding the shortest path. It goes to the deepest path in searching the vertices and then trackbacks.
2.	BFS requires more memory than DFS.	DFS requires less memory than BFS.
3.	BFS is slower than DFS.	DFS is faster than BFS.
4.	BFS is implemented using a queue (FIFO) data structure.	DFS is implemented using a stack (LIFO) data structure.
5.	BFS is a non-recursive algorithm.	DFS is a recursive algorithm.