

Graphs

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices.

Why are Graphs Useful?

Graphs are widely used to model any situation where entities or things are related to each other in pairs. For example, the following information can be represented by graphs:

- Family trees in which the member nodes have an edge from parent to each of their children.
- Transportation networks in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

Definition

A graph G is defined as an ordered set (V, E) , where $V(G)$ represents the set of vertices and $E(G)$ represents the edges that connect these vertices.

Example:

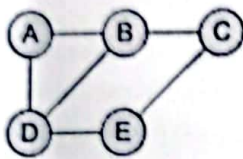


Figure: Undirected Graph.

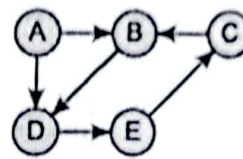


Figure: Directed Graph

Figure shows a graph with $V(G) = \{A, B, C, D \text{ and } E\}$ and $E(G) = \{(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)\}$.

A graph can be directed or undirected.

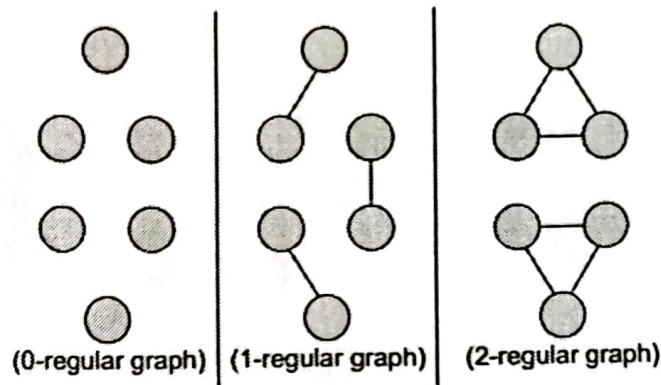
- In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A.
- In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).

Graph Terminology

Adjacent nodes For every edge, $e = (u, v)$ that connects nodes u and v , the nodes u and v are the end-points and are said to be the adjacent nodes.

Degree of a node Degree of a node u , $\deg(u)$, is the total number of edges containing the node u . If $\deg(u) = 0$, it means that u does not belong to any edge and such a node is known as an isolated node.

Regular graph It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree k is called a k -regular graph or a regular graph of degree k .



Path A path P written as $P = \{v_0, v_1, v_2, \dots, v_n\}$, of length n from a node u to v is defined as a sequence of $(n+1)$ nodes. Here, $u = v_0$, $v = v_n$ and v_{i-1} is adjacent to v_i for $i = 1, 2, 3, \dots, n$.

Closed path A path P is known as a closed path if the edge has the same end-points. That is, if $v_0 = v_n$.

Simple path A path P is known as a simple path if all the nodes in the path are distinct with an exception that v_0 may be equal to v_n . If $v_0 = v_n$, then the path is called a closed simple path.

Cycle A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).

Connected graph A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v . That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph.

Complete graph A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has $n(n-1)/2$ edges, where n is the number of nodes in G .

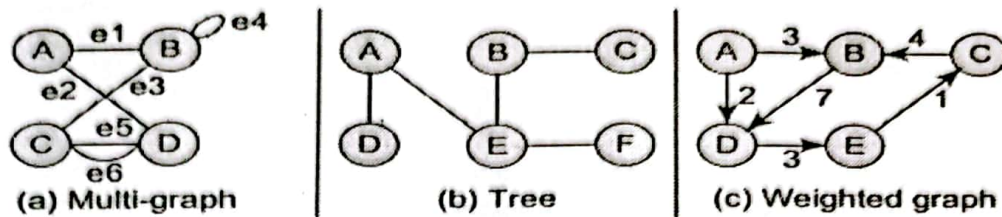
Labelled graph or weighted graph A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by $w(e)$ is a positive value which indicates the cost of traversing the edge.

Multiple edges Distinct edges which connect the same end-points are called multiple edges. That is, $e = (u, v)$ and $e' = (u, v)$ are known as multiple edges of G .

Loop An edge that has identical end-points is called a loop. That is, $e = (u, u)$.

Multi-graph A graph with multiple edges and/or loops is called a multi-graph.

Size of a graph The size of a graph is the total number of edges in it.



Directed Graphs

A directed graph G , also known as a *digraph*, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G . For an edge (u, v) , the edge begins at u and terminates at v .

Terminology of a Directed Graph

Out-degree of a node: The out-degree of a node u , written as $\text{outdeg}(u)$, is the number of edges that originate at u .

In-degree of a node: The in-degree of a node u , written as $\text{indeg}(u)$, is the number of edges that terminate at u .

Degree of a node: The degree of a node, written as $\text{deg}(u)$, is equal to the sum of in-degree and out-degree of that node. Therefore, $\text{deg}(u) = \text{indeg}(u) + \text{outdeg}(u)$.

Isolated vertex: A vertex with degree zero. Such a vertex is not an end-point of any edge.

Pendant vertex: (also known as leaf vertex) A vertex with degree one.

Cut vertex: A vertex which when deleted would disconnect the remaining graph.

Source: A node u is known as a source if it has a positive out-degree but a zero in-degree.

Sink: A node u is known as a sink if it has a positive in-degree but a zero out-degree.

Reachability: A node v is said to be reachable from node u , if and only if there exists a (directed) path from node u to node v .

Strongly connected directed graph: A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G . That is, if there is a path from node u to v , then there must be a path from node v to u .

Weakly connected digraph: A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

Parallel/Multiple edges: Distinct edges which connect the same end-points are called multiple edges. That is, $e = (u, v)$ and $e' = (u, v)$ are known as multiple edges of G .

Representation of Graphs

There are two common ways of storing graphs in the computer's memory. They are:

- **Sequential representation** by using an adjacency matrix.
- **Linked representation** by using an adjacency list that stores the neighbours of a node using a linked list.

Adjacency Matrix Representation:

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, *two nodes are said to be adjacent if there is an edge connecting them.*

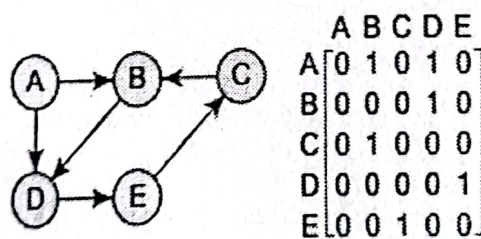
In a directed graph G , if node v is adjacent to node u , then there is definitely an edge from u to v . For any graph G having n nodes, the adjacency matrix will have the dimension of $n \times n$.

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry a_{ij} in the adjacency matrix will contain 1, if vertices v_i and v_j are adjacent to each other. However, if the nodes are not adjacent, a_{ij} will be set to zero.

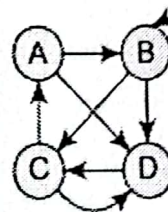
Since an adjacency matrix contains only 0s and 1s, it is called a **bit matrix** or a **Boolean matrix**.

$$a_{ij} = \begin{cases} 1 & \text{[if } v_i \text{ is adjacent to } v_j, \text{ that is} \\ & \text{there is an edge } (v_i, v_j)] \\ 0 & \text{[otherwise]} \end{cases}$$

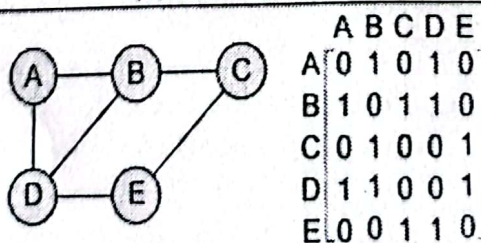
For Example: Graph and their corresponding adjacency matrices



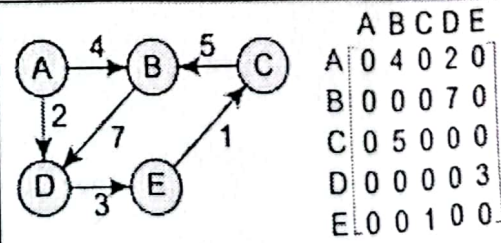
(a) Directed graph



(b) Directed graph with loop



(c) Undirected graph



(d) Weighted graph

From the above examples, we can draw the following conclusions:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Now let us discuss the powers of an adjacency matrix:

From adjacency matrix A^1 , we can conclude that an entry 1 in the i th row and j th column means that there exists a path of length 1 from v_i to v_j . Now consider, A^2 , A^3 , and A^4 .

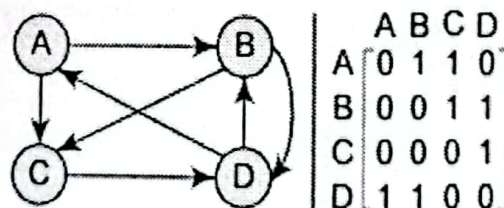
$$(a_{ij})^2 = \sum a_{ik} a_{kj}$$

Any entry $a_{ij} = 1$ if $a_{ik} = a_{kj} = 1$. That is, if there is an edge (v_i, v_k) and (v_k, v_j) , then there is a path from v_i to v_j of length 2.

Similarly, every entry in the i th row and j th column of A^3 gives the number of paths of length 3 from node v_i to v_j .

In general terms, we can conclude that every entry in the i th row and j th column of A^n (where n is the number of nodes in the graph) gives the number of paths of length n from node v_i to v_j .

Question. Consider a directed graph given in Figure. Given its adjacency matrix A , let us calculate A^2 , A^3 , and A^4 .



Solution:

$$A^2 = A^1 \times A^1$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

Now, based on the above calculations, we define matrix B as:

$$B^r = A^1 + A^2 + A^3 + \dots + A^r$$

The main goal to define matrix B is to obtain the path matrix P. The path matrix P can be calculated from B by setting an entry $P_{ij} = 1$, if B_{ij} is non-zero and $P_{ij} = 0$, if otherwise. The path matrix is used to show whether there exists a simple path from node v_i to v_j or not.

The path matrix entry is:

$$P_{ij} \begin{cases} \rightarrow 1 \text{ [if there is a path from } v_i \text{ to } v_j] \\ \rightarrow 0 \text{ [otherwise]} \end{cases}$$

Let us now calculate matrix B and matrix P:

$$B =$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} + \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 6 & 5 \\ 3 & 5 & 6 & 7 \\ 2 & 3 & 3 & 5 \\ 6 & 8 & 7 & 8 \end{bmatrix}$$

Now the path matrix P can be given as:

$$P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

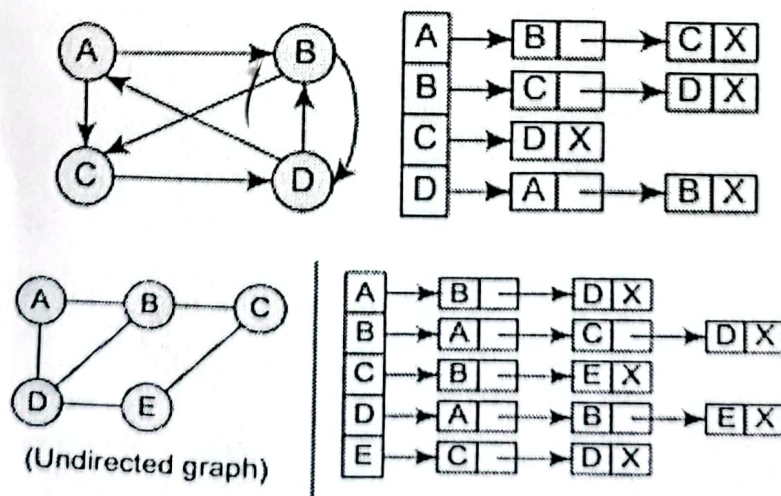
Adjacency List Representation:

An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, *every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.*

The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

Consider the graph and see how its adjacency list is stored in the memory.



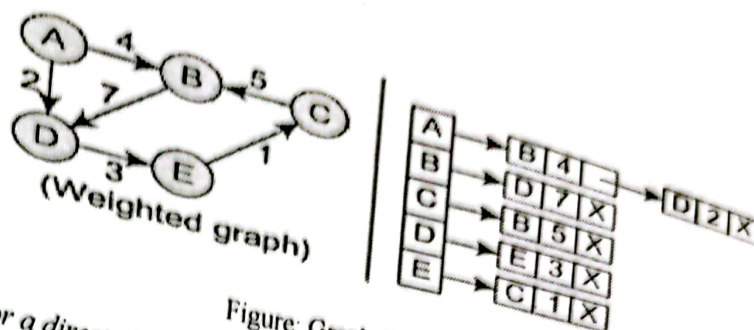


Figure: Graph G and its adjacency list

- For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G.
- For an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u .
- Adjacency lists can also be modified to store weighted graphs.

Graph Traversal Algorithms:

Traversing a graph, means the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal. They are:

1. Breadth-first search
2. Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack.

1. Breadth-First Search Algorithm

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour's nodes, and so on, until it finds the goal. It follows the rule:

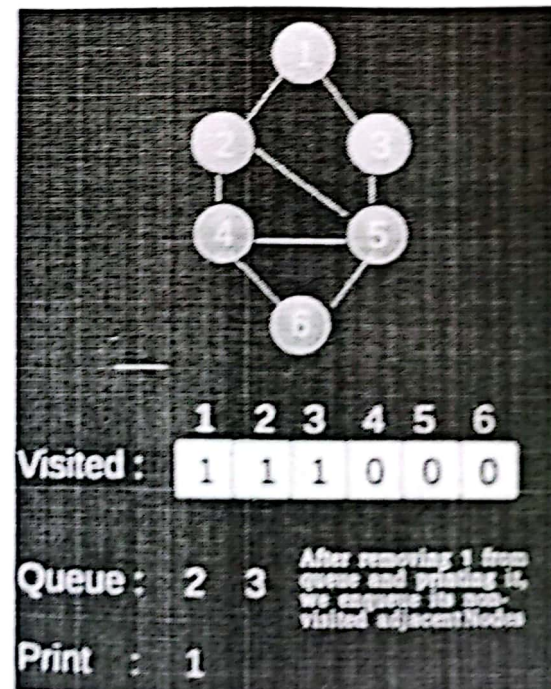
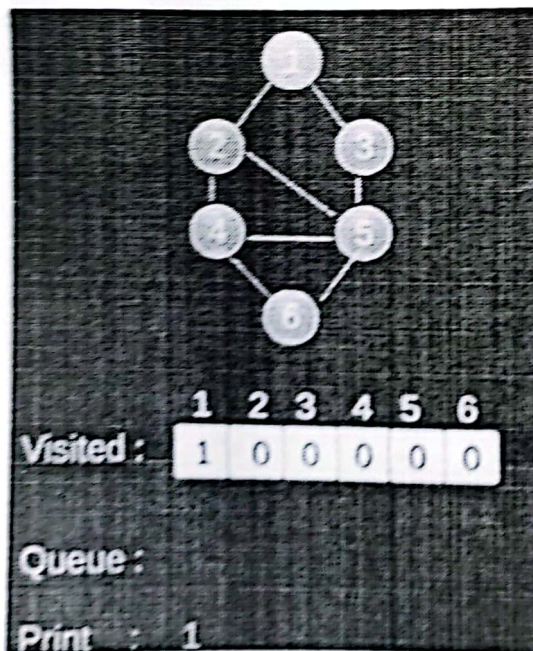
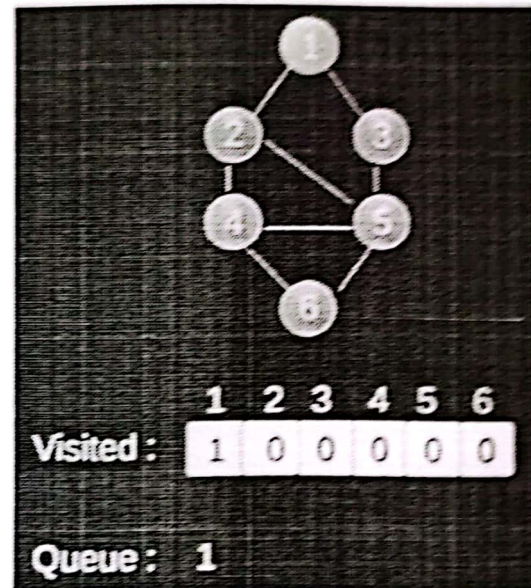
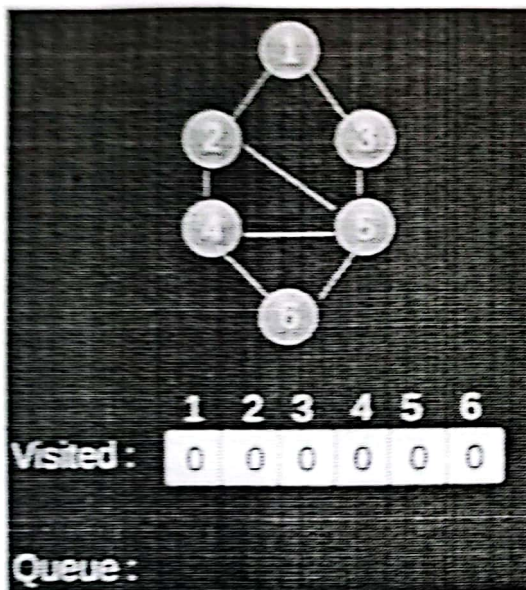
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

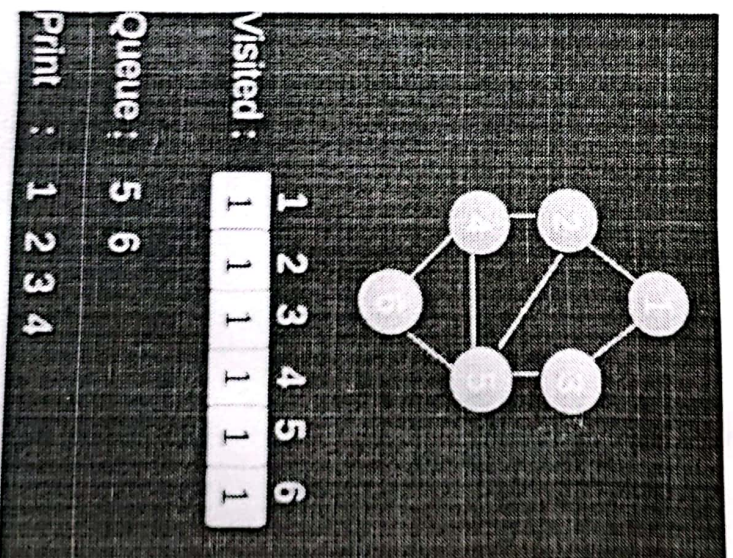
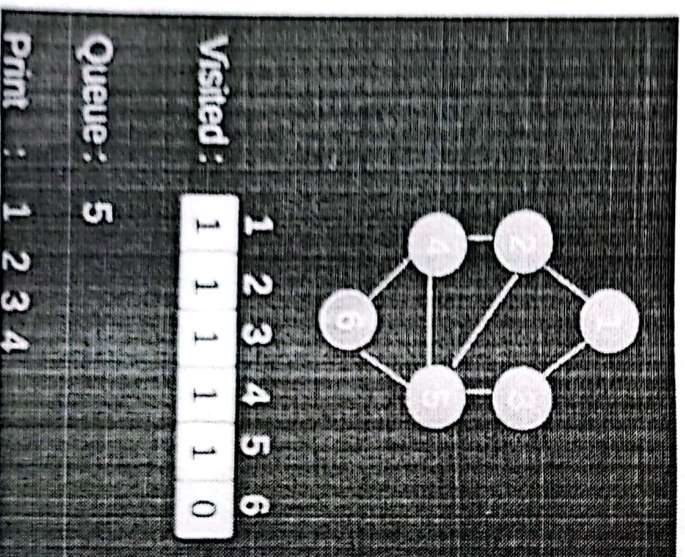
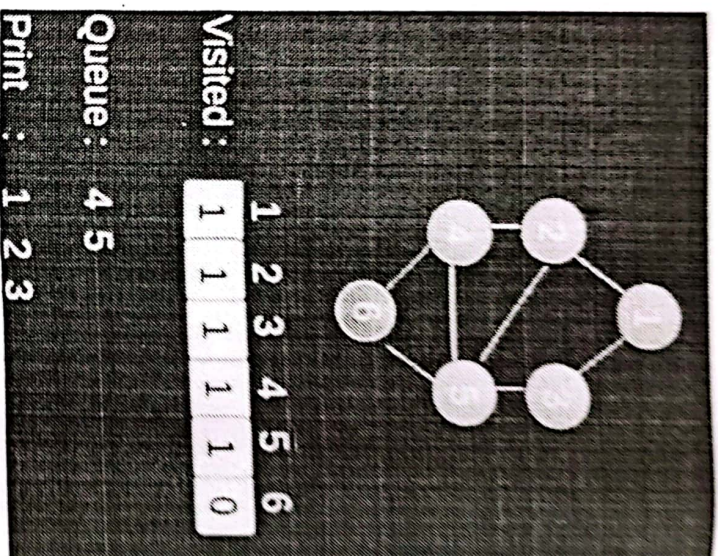
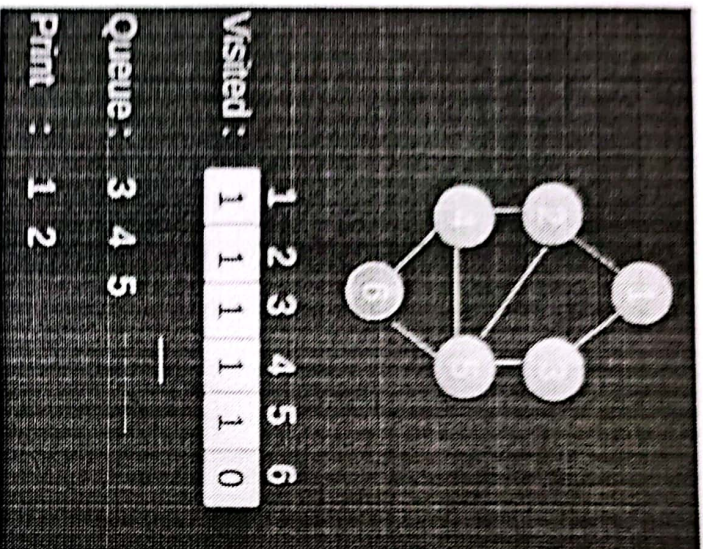
Algorithm for BFS

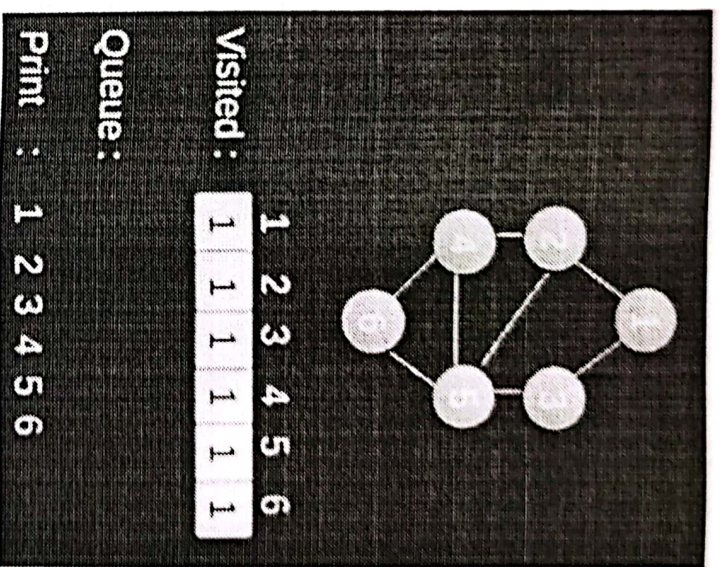
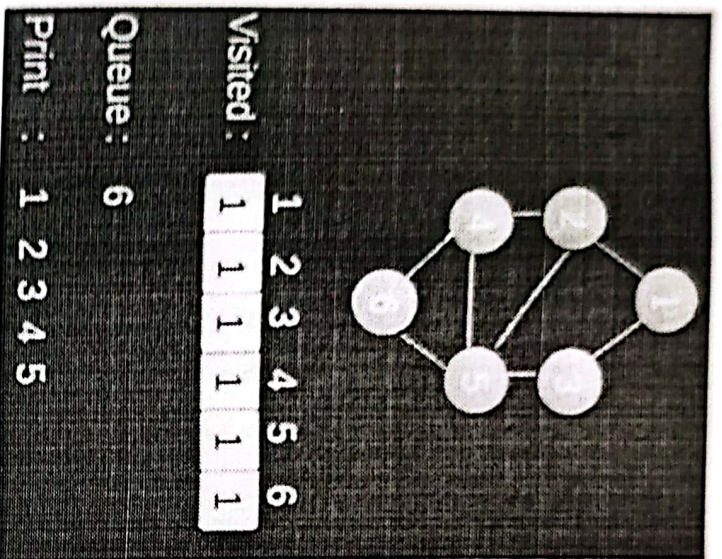
1. Start with one node of graph. It may be traverse from left to right.
2. So there are two factors which are important for traversing. first that the node is visited or not and second one is there any adjacent nodes for current node.
3. If we are visiting current node then add node into queue.

4. If the current node having adjacent nodes then add those nodes into queue.
5. If there are no more adjacent are found then print the current node and place the pointer to first node of queue.
6. Repeat these until all nodes get traversed

Example:







2. Depth-First Search Algorithm

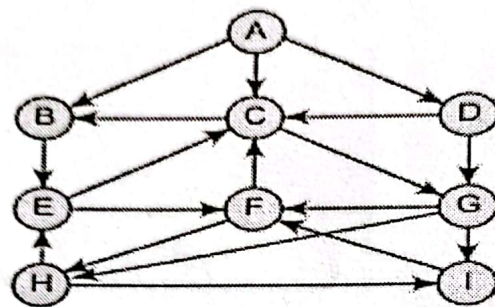
The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored. It follows the rule:

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

Algorithm for DFS

1. Start from one of the graph node it may be start from left to right.
2. Visit the current node and push it to stack.
3. Check the top of the stack having adjacent nodes.
4. If yes then push the adjacent nodes into stack.
5. If no then print the top of the stack and pop it from stack.
6. Repeat these steps until you will visit all node.

Example: Consider the graph G given in Figure. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H.



Adjacency lists

A: B, C, D
 B: E
 C: B, G
 D: C, G
 E: C, F
 F: C, H
 G: F, H, I
 H: E, I
 I: F

Solution:

- (a) Push H onto the stack.

STACK: H

- (b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H

STACK: E, I

- (c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I

STACK: E, F

- (d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F

STACK: E, C

- (e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C

STACK: E, B, G

- (f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G

STACK: E, B

- (g) Pop and print the top element of the **STACK**, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The **STACK** now becomes

PRINT: B

STACK: E

- (h) Pop and print the top element of the **STACK**, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The **STACK** now becomes empty.

PRINT: E

STACK:

Since the **STACK** is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

Shortest Path Algorithms:

We will discuss three different algorithms to calculate the shortest path between the vertices of a graph G. These algorithms include:

- Minimum Spanning Tree
- Dijkstra's Algorithm
- Warshall's Algorithm

While the first two use an adjacency list to find the shortest path, Warshall's algorithm uses an adjacency matrix to do the same.

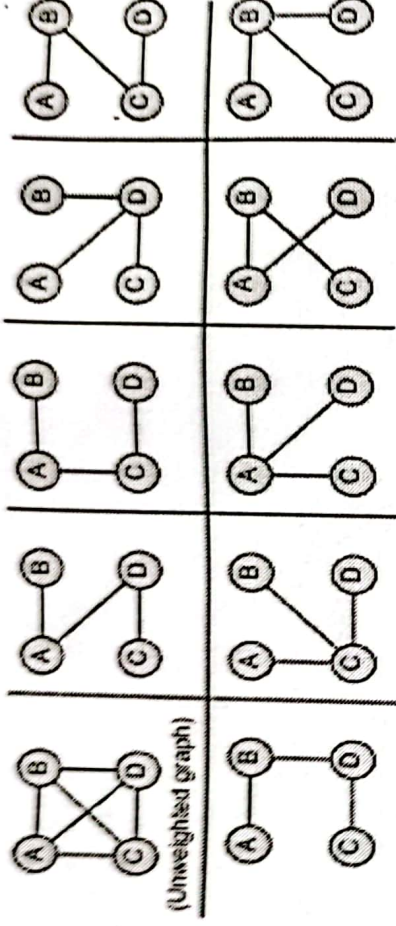
1. Minimum Spanning Trees:

A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together. A graph G can have many different spanning trees. We can assign *weights* to each edge and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree.

A **minimum spanning tree (MST)** is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree.

In other words, *a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.*

Example: Consider an unweighted graph G given below. From G, we can draw many distinct spanning trees.



Applications of Minimum Spanning Trees:

- MSTs are widely used for designing networks. For instance, people separated by varying distances wish to be connected together through a telephone network.
- MSTs are used to find airline routes. While the vertices in the graph denote cities, edges represent the routes between these cities.
- MSTs are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires or telephone lines.
- MSTs are applied in routing algorithms for finding the most efficient path.

Prim's Algorithm:

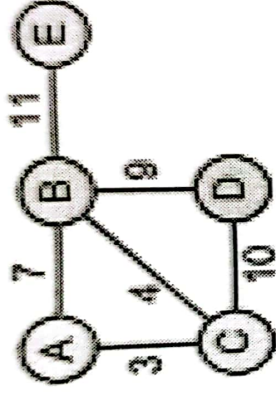
Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph. In other words, the algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized.

The steps involved in the Prim's algorithm are:

- Step 1: Select a starting vertex
- Step 2: Repeat Steps 3 and 4 until there are fringe vertices
- Step 3: Select an edge e connecting the tree vertex and
 fringe vertex that has minimum weight
- Step 4: Add the selected edge and the vertex to the
 minimum spanning tree T
- [END OF LOOP]
- Step 5: EXIT

Fringe vertices: Vertices that are currently not a part of T , but are adjacent to some tree vertex.

Example: Construct a minimum spanning tree of the given graph.



Solution:

Step 1: Choose a starting vertex A.

Step 2: Add the fringe vertices (that are adjacent to A). The edges connecting the vertex and fringe vertices are shown with dotted lines.

Step 3: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting A and C has less weight, add C to the tree. Now C is not a fringe vertex but a tree vertex.

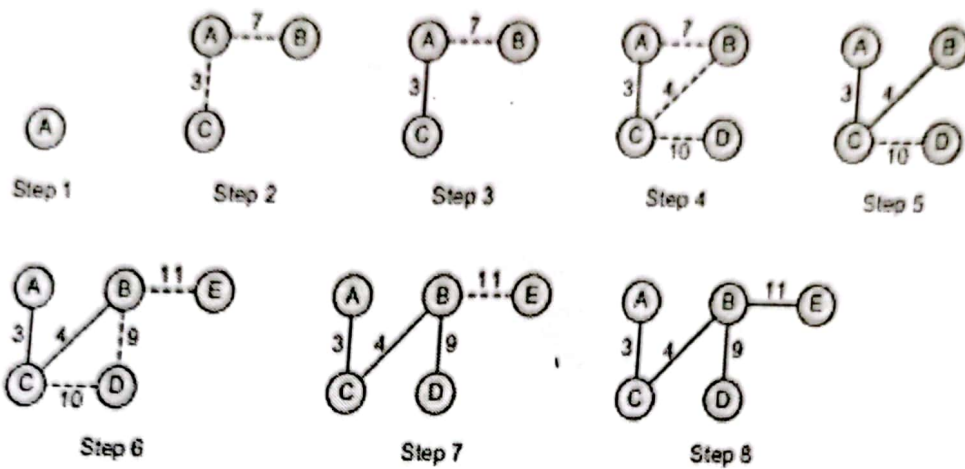
Step 4: Add the fringe vertices (that are adjacent to C).

Step 5: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting C and B has less weight, add B to the tree. Now B is not a fringe vertex but a tree vertex.

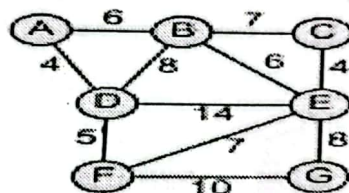
Step 6: Add the fringe vertices (that are adjacent to B).

Step 7: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting B and D has less weight, add D to the tree. Now D is not a fringe vertex but a tree vertex.

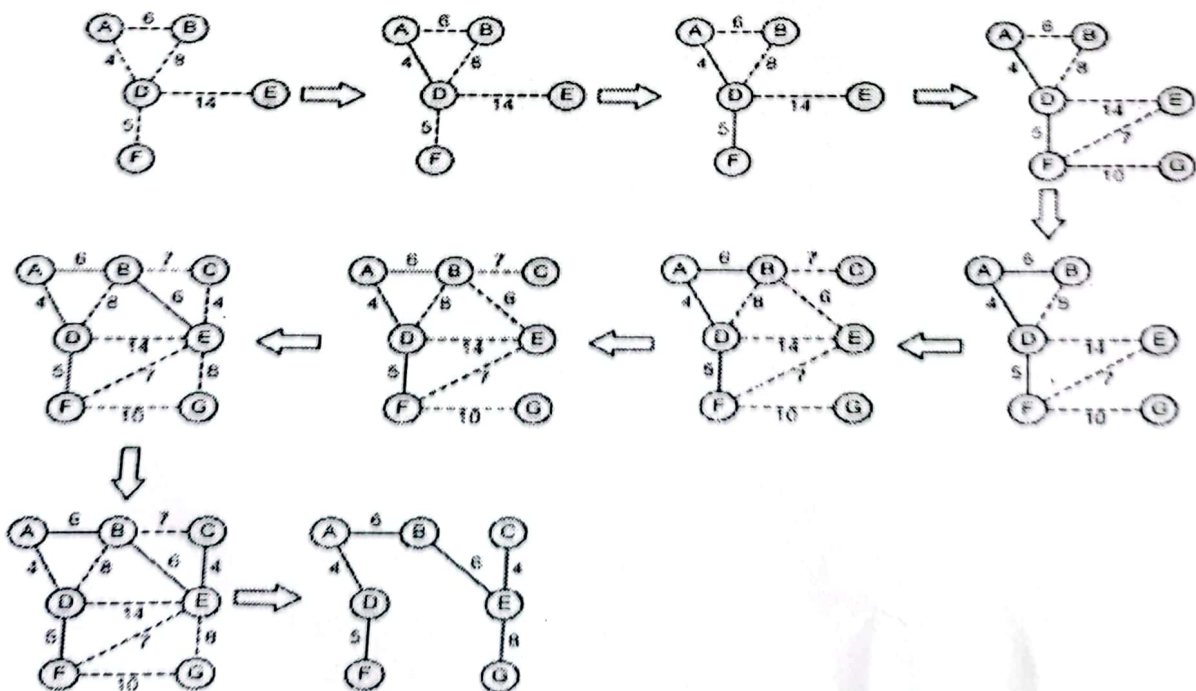
Step 8: Note, now node E is not connected, so we will add it in the tree because a minimum spanning tree is one in which all the n nodes are connected with n-1 edges that have minimum weight. So, the minimum spanning tree can now be given as,



Example 13.8 Construct a minimum spanning tree of the given graph in. Start the Prim's algorithm from vertex D.



Solution:



Kruskal's Algorithm:

Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph. If the graph is not connected, then it finds a minimum spanning forest. Note that a forest is a collection of trees. Similarly, a *minimum spanning forest is a collection of minimum spanning trees*.

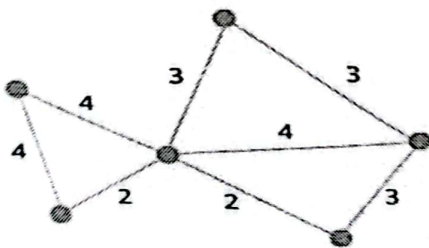
The algorithm is:

1. Sort all the edges from low weight to high weight.
2. Take the edge with the lowest weight and use it to connect the vertices of graph.
3. If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.
4. Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

Example: Kruskal's algorithm

1

Start with a weighted graph



2

Choose the edge with least weight, if there are more than 1, choose any one.



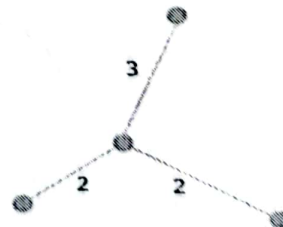
3

Choose the next shortest edge and add it



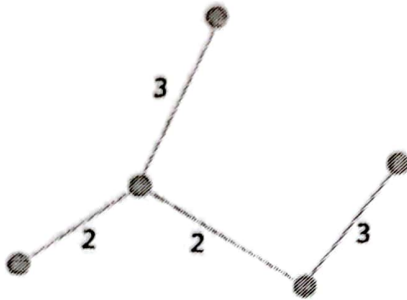
4

Choose the next shortest edge that doesn't create a cycle and add it



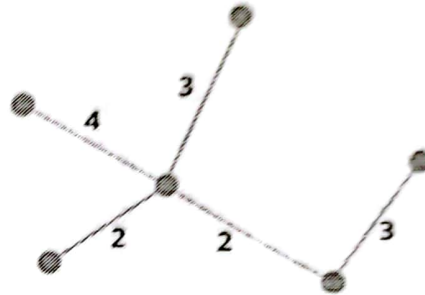
5

Choose the next shortest edge that doesn't create a cycle and add it

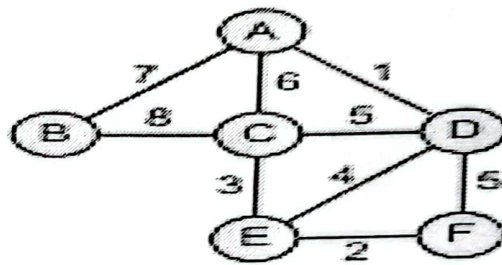


6

Repeat until you have a spanning tree



Q. Find the MST of given graph using Kruskal's algorithm.



Dijkstra's Algorithm:

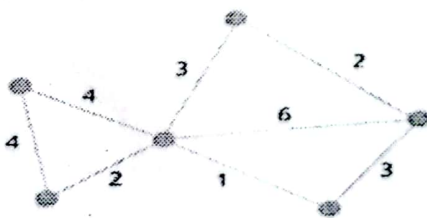
Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra's in 1959, is used to find the shortest path tree. This algorithm is widely used in network routing protocols. *Dijkstra's algorithm is also used for finding the costs of the shortest paths from a source node to a destination node.*

Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.

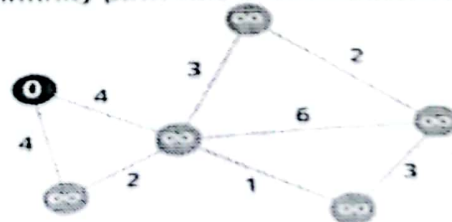
1

Start with a weighted graph



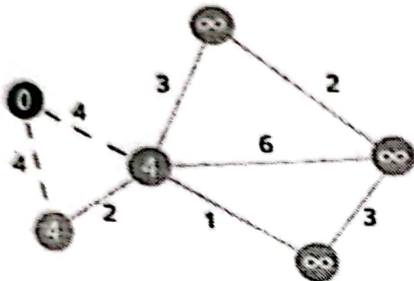
2

Choose a starting vertex and assign infinity path values to all other vertices



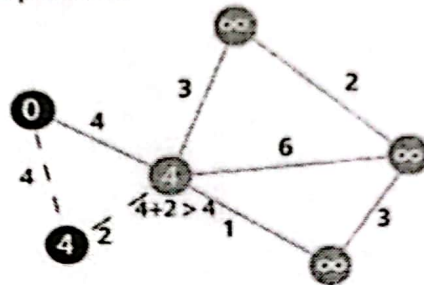
3

Go to each vertex adjacent to this vertex and update its path length



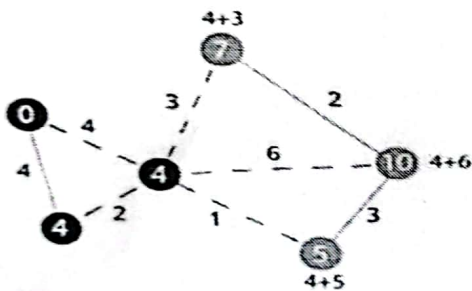
4

If the path length of adjacent vertex is lesser than new path length, don't update it.



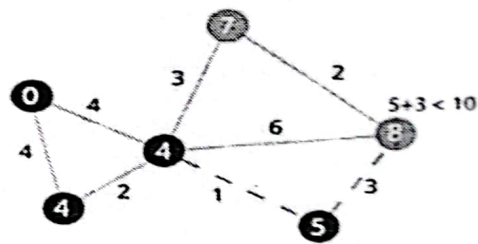
5

Avoid updating path lengths of already visited vertices



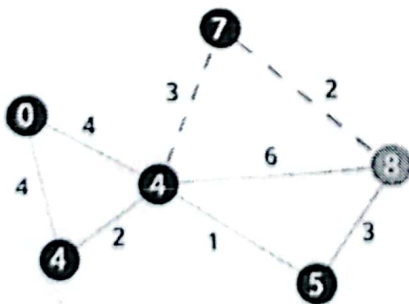
6

After each iteration, we pick the unvisited vertex with least path length. So we chose 5 before 7



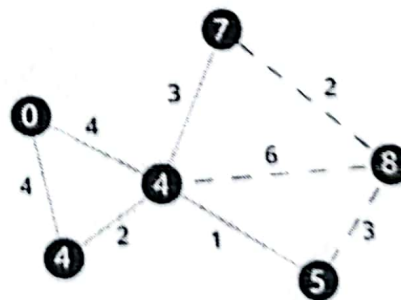
7

Notice how the rightmost vertex has its path length updated twice



8

Repeat until all the vertices have been visited



Dijkstra's Algorithm Pseudo Code:

```
function dijkstra(G, S)
  for each vertex V in G
    distance[V] ← infinite
    previous[V] ← NULL
  If V != S, add V to Priority Queue Q
  distance[S] ← 0

  while Q IS NOT EMPTY
    U ← Extract MIN from Q
    for each unvisited neighbour V of U
      tempDistance ← distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] ← tempDistance
        previous[V] ← U
    return distance[], previous[]
```

Floyd-Warshall Algorithm

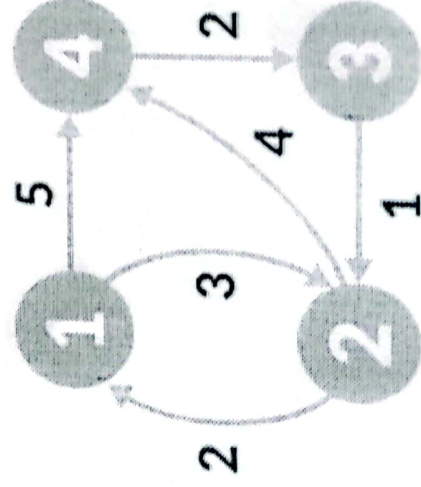
Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm or WFI algorithm.

How Floyd-Warshall Algorithm Works?

Let the given graph be:



Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A^1 of dimension $n \times n$ where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell $A[i][j]$ is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

$$A^0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

2. Now, create a matrix A^1 using matrix A^0 . The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. $A[i][j]$ is filled with $(A[i][k] + A[k][j])$ if $(A[i][j] > A[i][k] + A[k][j])$.

That is, if the direct distance from the source to the destination is greater than the path through the vertex k , then the cell is filled with $A[i][k] + A[k][j]$.

$$A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & 4 \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since $4 < 7$, $A^0[2, 4]$ is filled with 4.

3. In a similar way, A^2 is created using A^1 . The elements in the second column and the second row are left as they are.

In this step, k is the second vertex. The remaining steps are the same as in step 2.

$$A^2 = \begin{array}{c|ccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & & \\ 2 & 2 & 0 & 9 & 4 \\ 3 & & 1 & 0 & \\ 4 & & \infty & & 0 \end{array} \longrightarrow \begin{array}{c|ccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

4. Similarly, A^3 and A^4 is also created.

$$A^3 = \begin{array}{c|ccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & \infty & \\ 2 & & 0 & 9 & \\ 3 & \infty & 1 & 0 & 8 \\ 4 & & & 2 & 0 \end{array} \longrightarrow \begin{array}{c|ccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

$$A^4 = \begin{array}{c|ccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & & 5 \\ 2 & & 0 & & 4 \\ 3 & & & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array} \longrightarrow \begin{array}{c|ccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

5. A^4 gives the shortest path between each pair of vertices.

Floyd-Marshall Algorithm

```
n = no of vertices
A = retrix of dimension n*n
for k = 1 to n
  for i = 1 to n
    for j = 1 to n
       $A^k[i, j] = \min (A^{k-1}[i, j], A^{k-1}[i, k] + A^{k-1}[k, j])$ 
return A
```

Applications of Graph:

- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.