DESGIN & ANALYSIS OF ALGORITHM|| DATA STRUCTURE

DAA/DS Group

Authors	Prabhat Singh, Deepali Dev, Surbhi Verma Ravi Kumar, Birender Kumar ,Akhilesh Srivastava, Punnet Goyal , Amrita Jyoti, Manish Srivastava, Amit Pandey , Asmita Dixit
Authorized by	
Creation/Revision Date	18,June 2021
Version	1.0

UNIT-4

GRAPHS

Table of Contents

4. Graph:

4.1.1 Introduction to Graph:

Consider a courier boy who has to deliver various mails/items to different addresses assigned to him. The guy picks the packets from the office, and after delivering the items, he comes back to the office to submit the reports and undeliverable items (if any). The guy will have to plan his travel such that he does not have to repeat the location he already has visited. Such a scenario can intelligently be planned if we have the representation of locations and their connections in the form of Graph.

A social graph is also an example of a graph that illustrates interconnections among people and organizations in a social network.

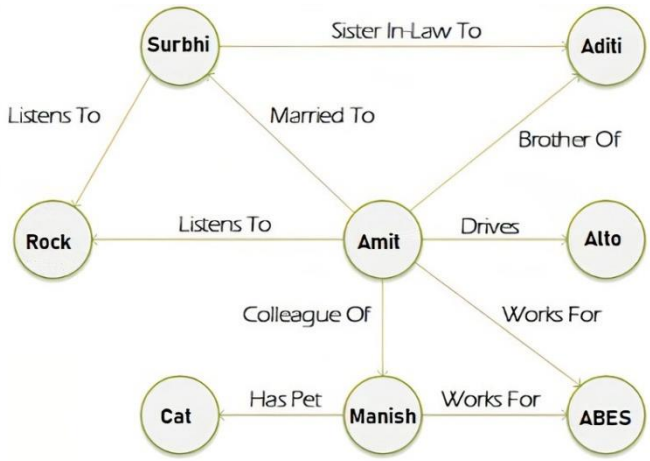


Figure 4.1: Social Graph (e.g., Facebook)

A level-up application of social Graph is LinkedIn that follows the connections of individuals with each other and the organizations and industries. It facilitates the organizations in targeting the right person for the recruitment and persons targeting the organization of their choices.

Similarly, Google and Microsoft offer various services online, e.g., calendar, mailing, document, spreadsheet, forms, etc. All these services are interconnected with each other. The same can also be considered as Graph.

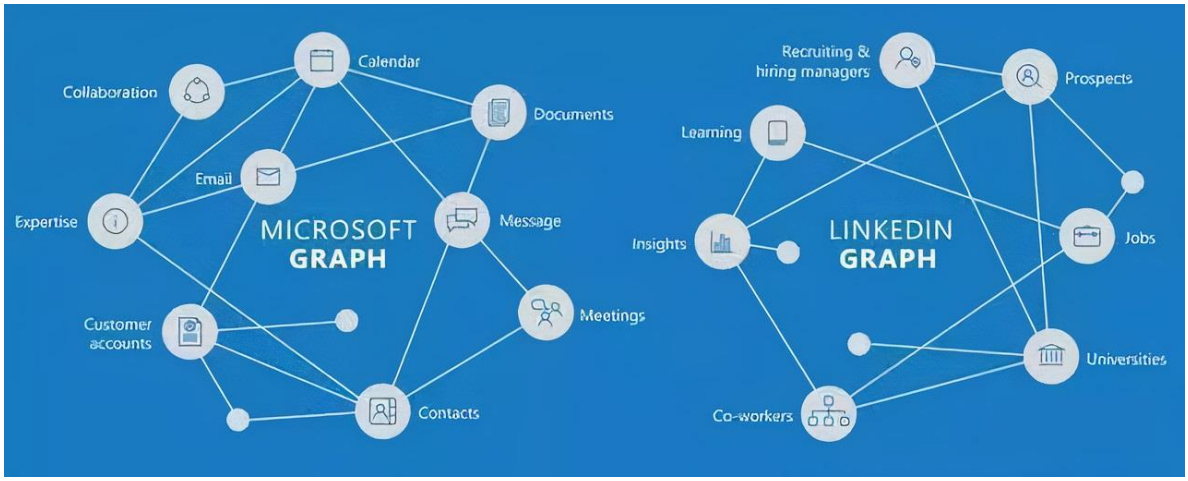


Figure 4.2: Software services Graph and Social Graph (Linked-in)

4.1.2. Definition of Graph:

A Graph is a non-linear data structure* consisting of vertices/nodes and edges/lines. A graph can be used to define pair-wise relation between the objects.
(In a non-linear data structure, elements stored in it follow some hierarchy)

A graph is defined as an ordered pair G = {V, E}, where
V = Set of vertices and
E = Set of edges.

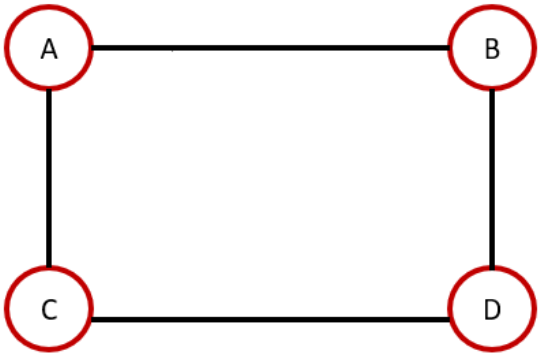


Figure 4.3: Example of Graph

For the above Graph
V = {A, B, C, D} and
E = {[A, B], [A, C], [B, D], [C, D]}

The above figure is an example of an undirected graph (A graph in which edges do not have any direction). Each edge is identified with an unordered pair [u, v] of nodes/vertices in Graph G where edge E begins at u and ends at v.

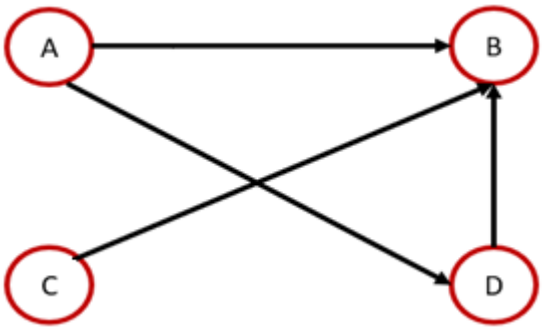


Figure 4.4: Example of directed Graph

The above figure is an example of a directed graph (in which edges have directions). Each edge E is identified with an ordered pair (u, v) of nodes in Graph G. Directed graphs are also called digraphs.
For the above Graph
V = {A, B, C, D} and
E = {(A, B), (A, D), (D, B), (C, B)}

There is direct connection from node A to B, but not from B to A. Similarly, direct connection exists from node D to B, but not from B to D.

4.1.3. Graph: Basic Terminology

1. ***Degree of vertex***

The degree of a vertex is the number of edges connecting it.

Notation – degree (V)

Degree of Vertex in a Directed Graph

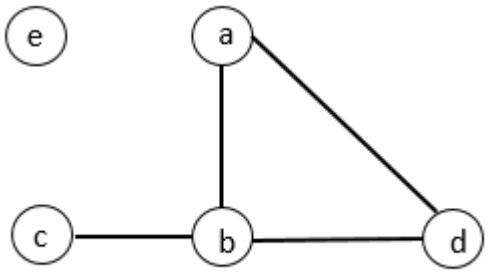


Figure 4.10 Undirected Graph

Table 1 Degree of undirected Graph for Example 1

Vertex	Degree
a	2
b	3
c	1
d	2

e	0

Degree of Vertex in a Directed Graph

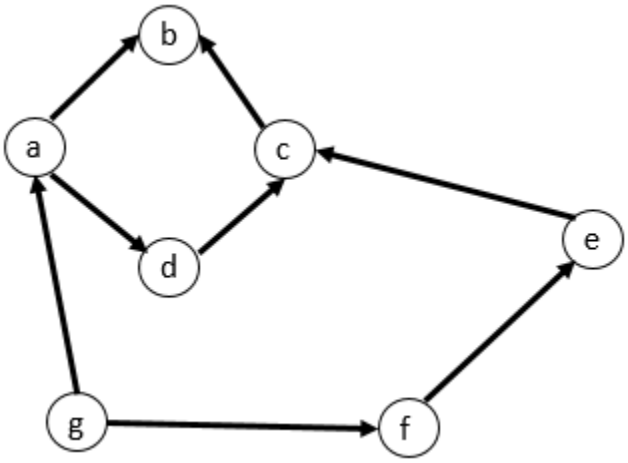
In a directed graph, each vertex has an **in-degree** and an **out-degree**.

In-degree of vertex V is the number of edges that are coming into the vertex V and Out-degree of vertex V is the number of edges that are going out from the vertex V.

In Degree Notation: $deg^-(V)$.

Out Degree Notation: $deg^+(V)$.

Consider the following example.



Vertex	In-degree	Out-degree
a	1	2
b	2	0
c	2	1
d	1	1
e	1	1
f	1	1
g	0	2

Figure 4.11. Directed Graph (Example 2)

Adjacent Vertices

Two nodes or vertices are adjacent if they are connected to each other through an edge. Vertex v1 is adjacent to a vertex v2 if there is an edge (v1, v2) or (v2, v1). Let us consider the following Graph:

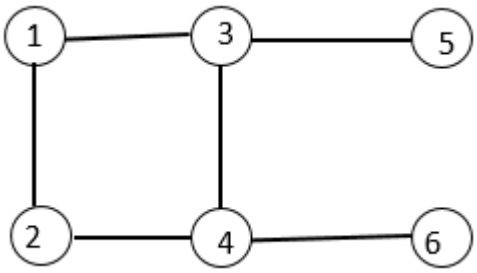


Figure 4.12

Vertices adjacent to node 2: 1 and 4

Vertices adjacent to node 4: 2, 3 and 6

Path

A path is a sequence of vertices with the property that each vertex in the sequence is adjacent to the vertex next to it.
Example: Undirected Graph

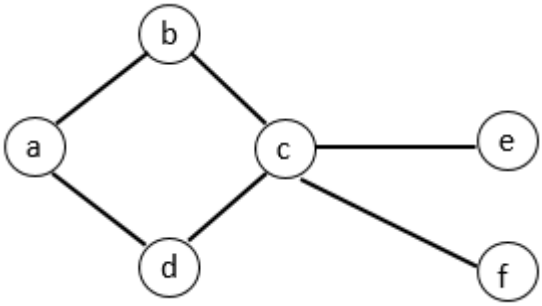


Figure 4.13. Undirected Graph

Possible Path from vertex a to f: a-d-c-f and a-b-c-f
Possible Path from vertex d to e: d-c-e
Example: Directed Graph

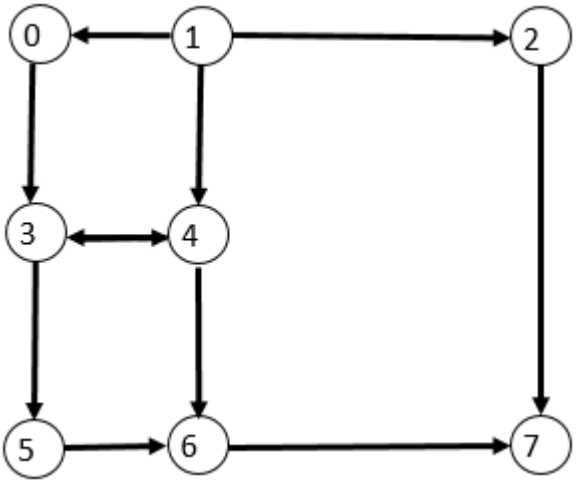


Figure 4.14. Directed Graph

Path from vertex 0 to 7: 0-3-5-6-7 and 0-3-4-6-7

Path Type

Simple Path: A path is simple path if all of its vertices are distinct.
Closed Path: A path is closed path if the first vertex is the same as the last vertex.

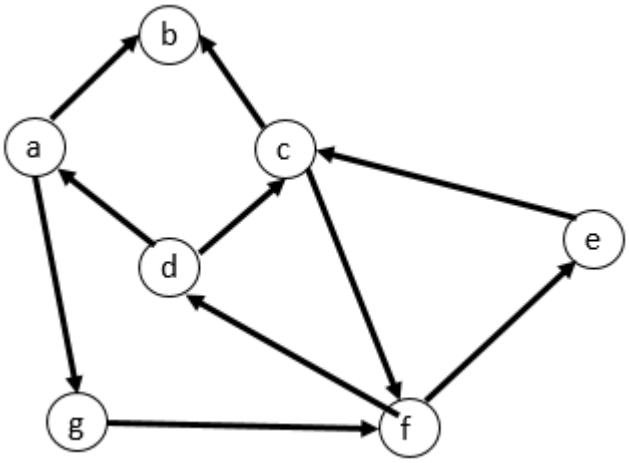


Figure 4.15: Path Type

Path a-g-f-e-c-f-d-c-b: neither simple nor closed
Path a-g-f-e-c-b: simple Path
Path a-g-f-e-c-f-d-a: closed Path

4.1.4. Types of Graph:

Null Graph

A graph is known as a null Graph if there are no edges in the Graph. In other words, a graph whose edge set is empty is called a null graph.

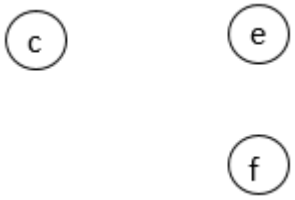


Figure 4.16: Null Graph

Trivial Graph

Graph having only a single vertex and no edges, and it is the smallest possible Graph.

Figure 4.17. Trivial Graph

Regular Graph

A graph in which the degree of all the vertices is the same is called a regular graph.
If all the vertices in a graph are of degree 'k', it is called a "k-regular graph."

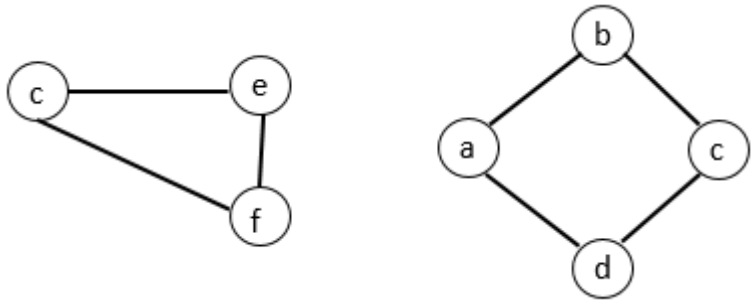


Figure 4.18: Examples of "2-Regular Graphs"

Connected Graph

The Graph in which we can visit any other node in the Graph from one node is known as a connected graph. At least one path exists between every pair of vertices in a connected Graph.

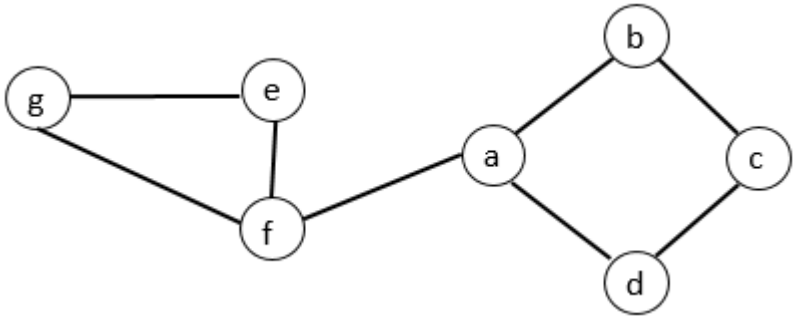


Figure 4.19. Connected Graph

Disconnected Graph

The Graph in which at least one node is not reachable from a node is a disconnected graph.

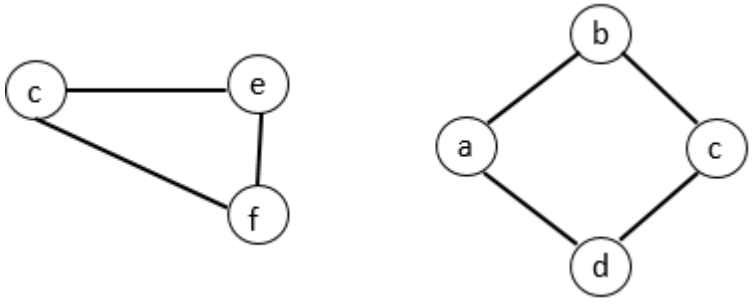


Figure 4.20. Disconnected Graph

The Graph in Fig 12 consists of two independent components that are disconnected.

Complete Graph

A graph is a complete Graph in which precisely one edge is present between every pair of vertices.

A complete graph of ‘n’ vertices contains exactly ${}^{n}C_{2}$edges.

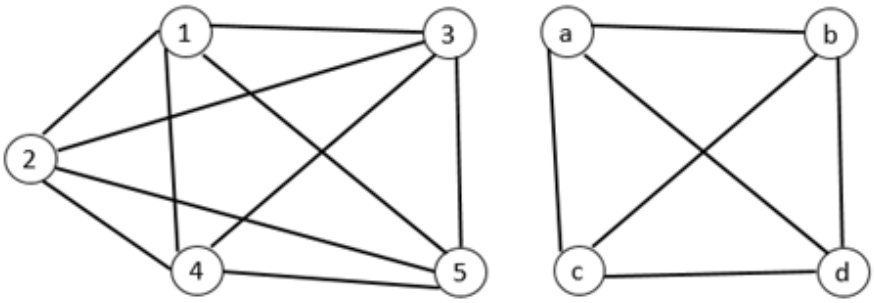


Figure 4.21. Examples of Complete Graph

Cycle Graph

A simple graph of 'n' vertices (n>=3) and n edges forming a cycle of length 'n' is called a cycle graph. In a cycle graph, all the vertices are of degree 2. In other words, a cycle is a simple closed path.

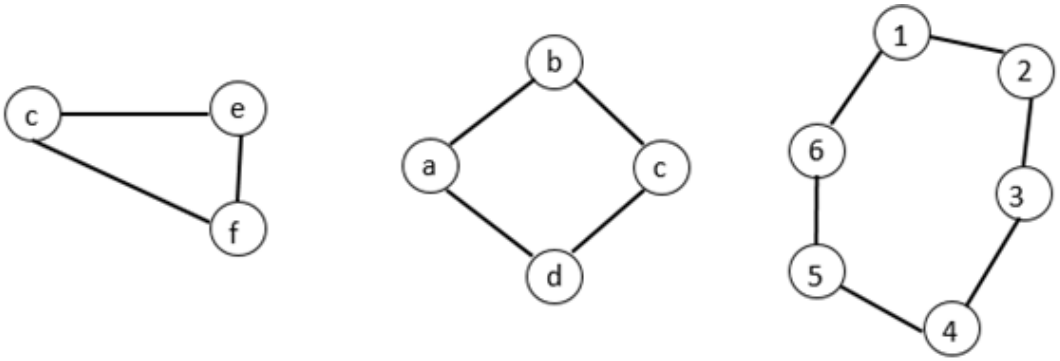


Figure 4.22: Three Examples of Cycle Graph

Cyclic Graph

A graph containing at least one cycle is known a cyclic graph.

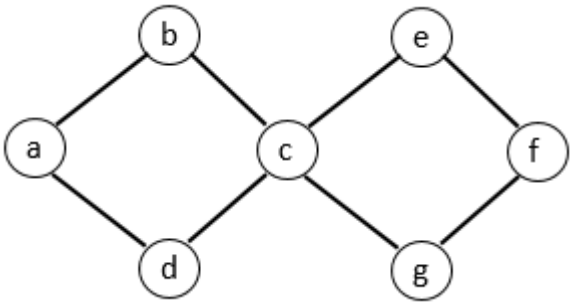


Figure 4.23. Cyclic Graph

This Graph in Figure 15 contains two cycles. Therefore, it is a cyclic graph.

Acyclic Graph

A graph not containing any cycle in it is called an acyclic graph.

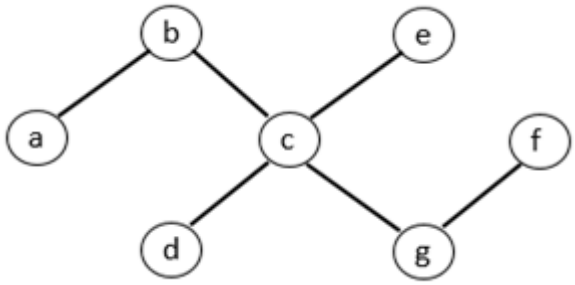


Figure 4.24. Acyclic Graph

Multi Graph

A graph that consists of parallel edges and self-loops is called a multigraph. Parallel edges are those which connect the same pair of vertices, and self-loops have the same vertex as the endpoints

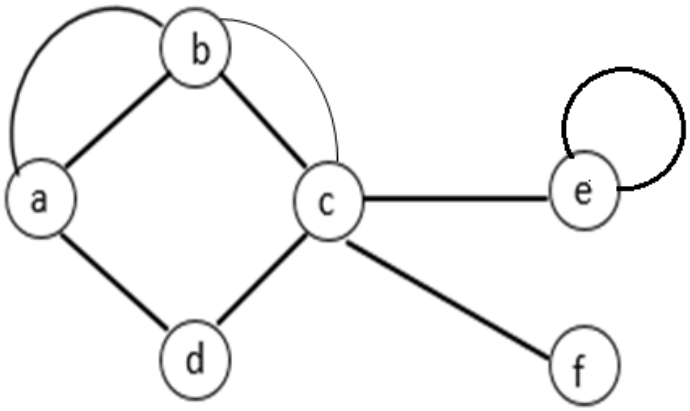


Figure 4.25. Multi Graph

The Graph above has multiple edges between a to b and b to c. It has self-loop from e to e.

Labelled Graph

A graph that has labels associated with each edge or each vertex is called Labelled Graph.

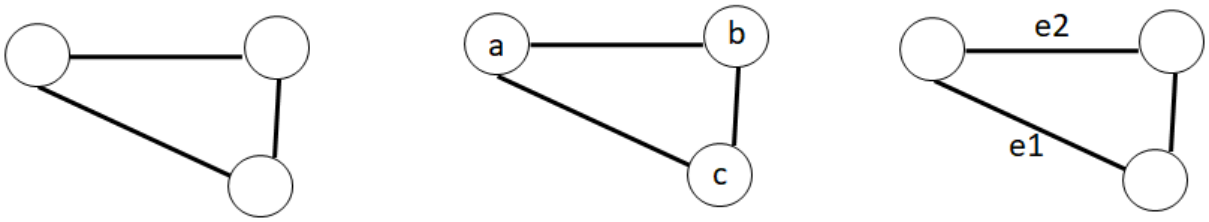


Figure 4.26 (a) Unlabelled Graph (b) Vertex-labelled Graph (c) edge Labelled Graph

Weighted Graph

A graph having a weight, or number, associated with each edge is called Weighted Graph.

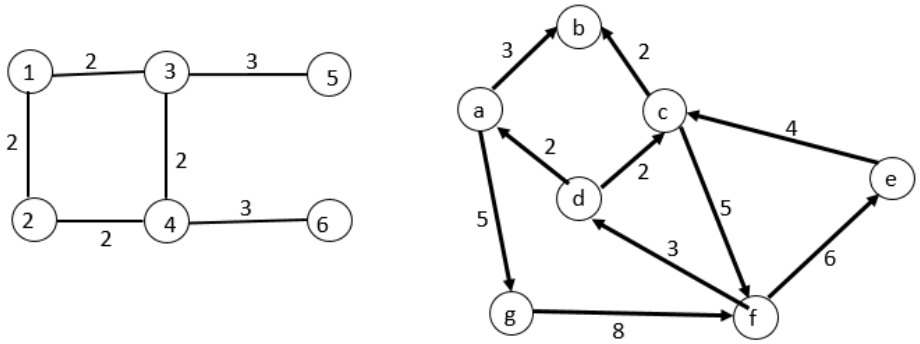


Figure4.27: (a) Weighted undirected Graph (b) Weighted directed Graph

Bi-Partite Graph

A bipartite graph is a set of graph vertices decomposed into two disjoint sets such that no graph vertices within the same are adjacent. It is also called a bi-graph.

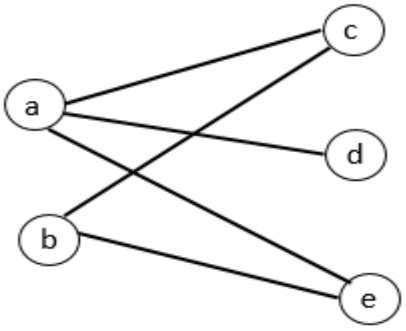


Figure 4.28 Bi-Partite Graph

Set 1 (vertices): {a, b} Set 2 (vertices): {c, d, e}

Euler Graph

A connected graph G is an Euler graph if all vertices of G are of even degree. In other words, a connected graph G is an Euler graph, if and only if its edge set can be decomposed into cycles.

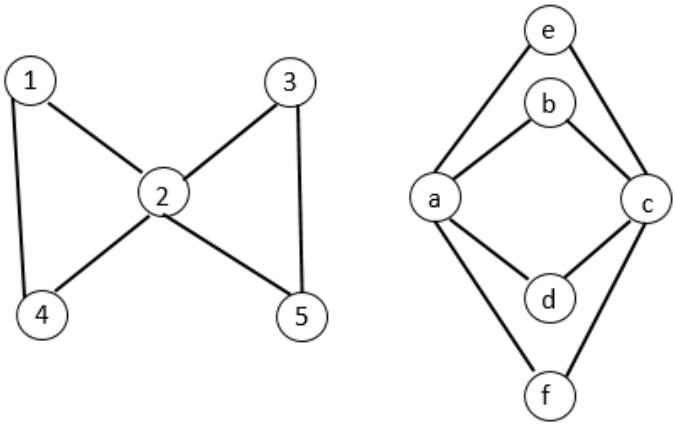


Figure 4.29. Two examples of Euler Graph

Hamiltonian Graph

A connected graph G is called a Hamiltonian graph if there is a cycle that includes every vertex of G and the obtained cycle is called the Hamiltonian cycle.

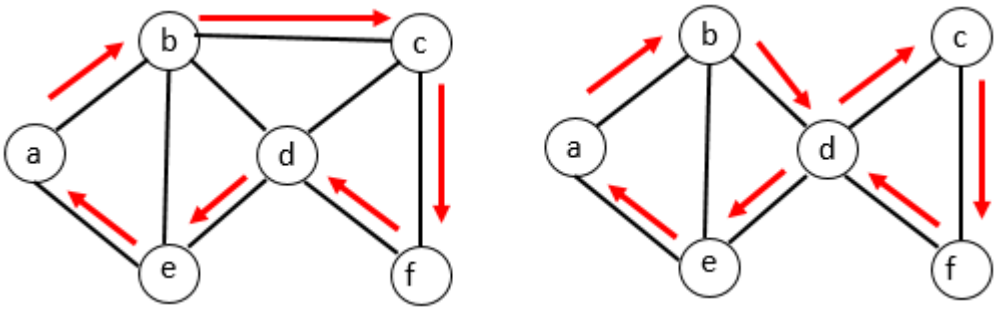


Figure: 4.30 (a). Hamiltonian Graph 19(b.) Non-Hamiltonian Graph

Figure 19(a) graph contains a cycle a-b-c-f-d-e-a that visits each vertex exactly once. Therefore, the cycle is a Hamiltonian cycle. It implies that the Graph is a Hamiltonian Graph.

A cycle in Figure 4.30(b) graph visits all vertices, a-b-d-c-f-d-e-a but vertex d appears twice in the cycle. Thus, the Graph does not contain a Hamiltonian cycle. It implies that the Graph is a Non-Hamiltonian Graph.

4.1.5. Similarity and difference between Tree and Graph:

Similarity: Both Tree and Graph are non-linear data structures that contain nodes and edges (used to connect nodes).

Difference:

- The Tree does not contain any Cycle but the Graph may or may not contain a cycle.
- A root is always defined in a Tree that is primarily used to identify the starting point in Tree. In a Graph it is not essential to define any vertex as a root.

4.1.6. Applications of Graph:

a) **Google maps:** One of the most common applications of the Graph is the Google maps, where cities are considered vertices and connection between cities are represented as edges. Whenever we have to find a route between two stations, the shortest path algorithm works to select the shortest route between the pair of vertices (e.g. Dijkstra Algorithm, to be discussed in subsequent text in this chapter).

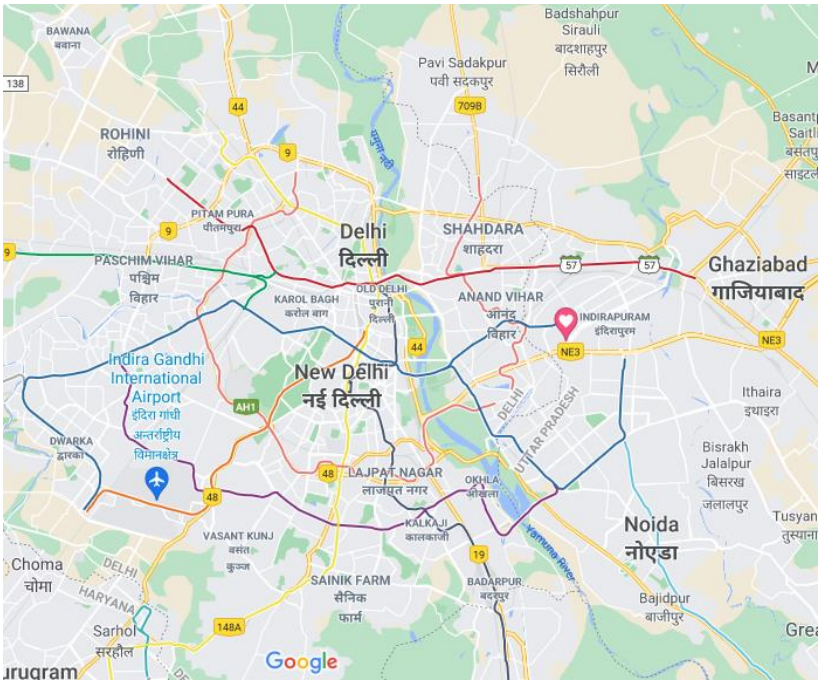


Figure 4.5: - Google map

b) Social networks like Facebook, Instagram, LinkedIn etc, are another example of Graph in which people are connected with each other just like edges connected in the Graph. It is an example of an undirected Graph. In Facebook, every user is considered a vertex in Graph. When user1 sends a friend request to user2, an edge is formed between user1 and user2.

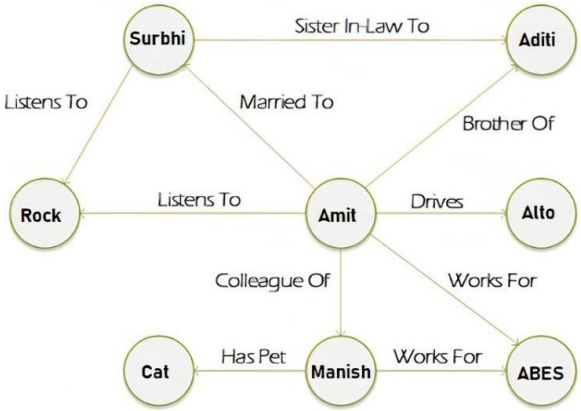


Figure 4.6: - Facebook as Graph

c) World wide Web: The world wide web (www) has become the primary information source. It consists of a large number of pages that are tied together using hyperlinks. This www can be considered as a directed graph in which a web page can be considered a node, and the hyperlink can be considered an edge.

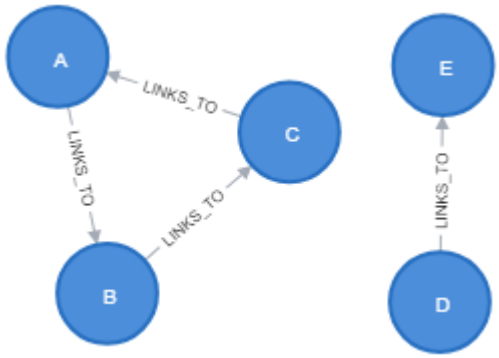


Figure 4.7: - www as graph

d) In operating system, Resource Allocation graph are also example of Graph in which resources and processes are considered as vertices and there will be either request edge (if a process request for resource) or assignment edge (if resource is allocated to a process).

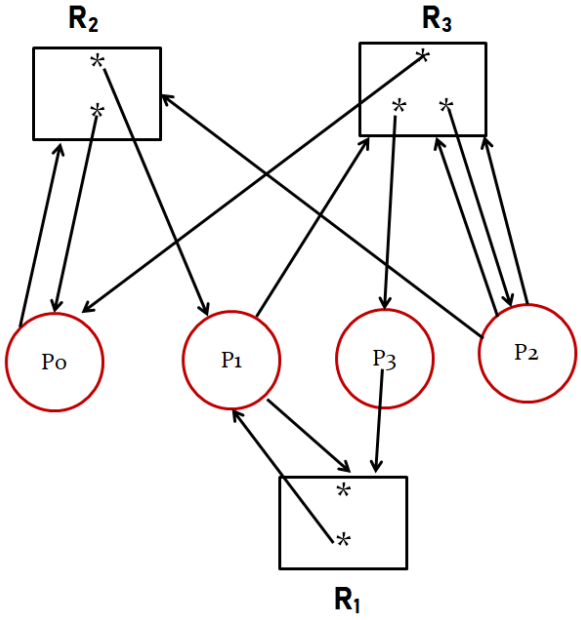


Figure 4.8: - Resource Allocation Graph (Rectangles represent Resources and Circles represent Processes)

4.1.7. Graph Representation:

In practice, it is not possible to represent the Graph by using node structure just like binary tree representation. There are three types of representation of graphs in the memory.

1. **Sequential Representation-**

The Graph can be represented sequentially by using matrices. It is of two types-

Adjacency –Matrix Representation-

In this representation, take a two-dimensional integer type array whose order depends on the number of vertices in the given Graph. Let us suppose that if the given Graph contains v vertex, then the order of the two-dimensional array is vXv.

v[i][j]

1	if a single edge exists between vertex i to vertex j (adjacent node)
2	if self-loop exists (In case of multigraph)
n	if number of parallel edges is n
w	if assigned weight is w (weighted Graph)
0	otherwise

Adjacency matrix is also called bit matrix or Boolean matrix. If an edge exists between two nodes, it stores true (represented by 1) otherwise false (represented by 0).

v[i][j]

True or 1	if edge exist between I to j
False or 0	otherwise

Example (Directed Graph)

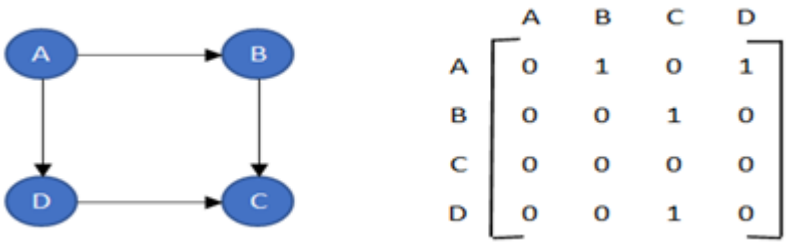


Figure 4.31: Graph Adjacency matrix Representation

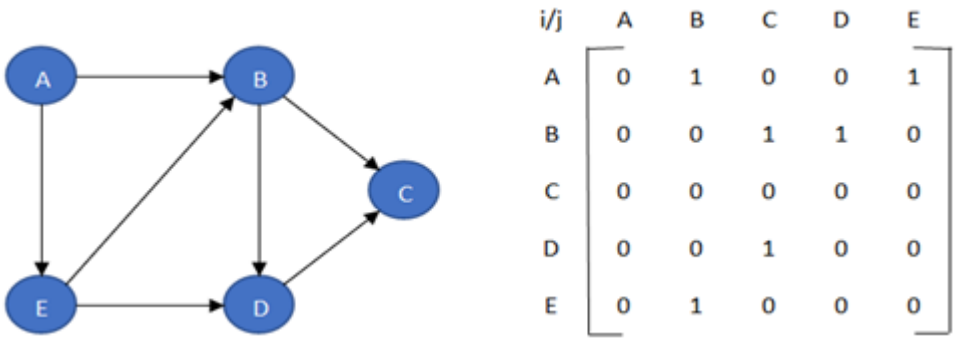


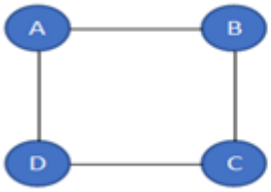
Figure 4.32: Graph Adjacency matrix Representation

In the above diagram A to B edge exist, so in the matrix A to B is 1. A to C edge does not exist, so in the matrix, A to C is 0.

Points to Remember-

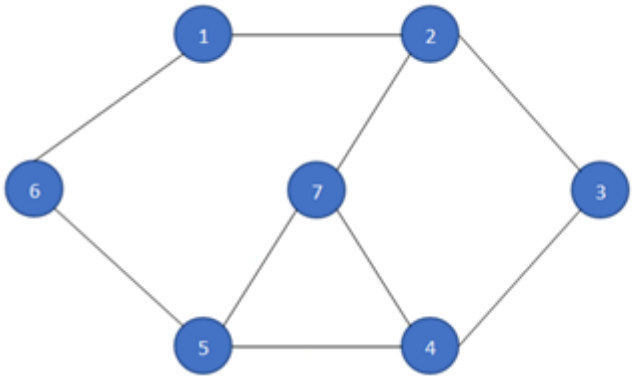
In the case of a directed Graph, the sum of the matrix is always equal to the number of edges |E| or the sum of in-degree of each vertex.
In above diagram
Number of edges= 6
Sum of matrix = 6
Number of edges = Sum of Matrix

Example (Undirected Graph)



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

Figure: 4.33 Undirected Graph



	1	2	3	4	5	6	7
1	0	1	0	0	0	1	0
2	1	0	1	0	0	0	1
3	0	1	0	1	0	0	0
4	0	0	1	0	1	0	1
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	1	0	1	1	0	0

Figure 4.34: Graph Adjacency matrix Representation

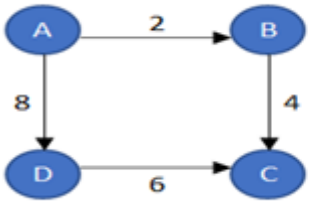
In the above diagram A to B edge exist, so in the matrix A to B is 1. A to C edge does not exist, so in the matrix, A to C is 0.

Points to Remember-

In the case of an undirected Graph, the sum of the matrix is always equal to twice the number of edges i.e., 2|E| or sum of degree of each vertex.

In above diagram,
Number of edges= 9, Sum of matrix= 18
Sum of matrix=2|E|

Example (Weighted Directed Graph)

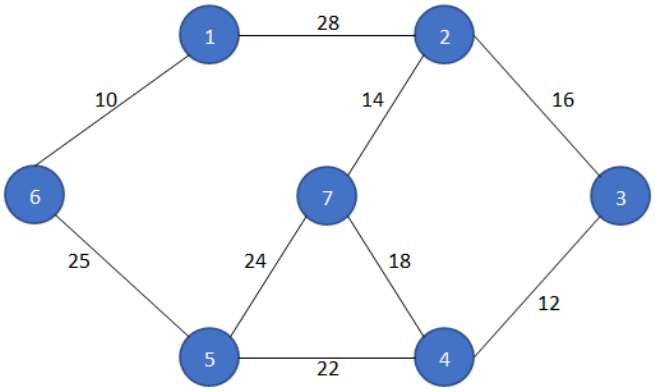


	A	B	C	D
A	0	2	0	8
B	0	0	4	0
C	0	0	0	0
D	0	0	6	0

Figure 4.35: Graph Adjacency matrix Representation

In the above diagram A to B edge exist, so in the matrix A to B is 2. A to C edge does not exist, so in the matrix A to C is 0.

Example (Weighted Undirected Graph)

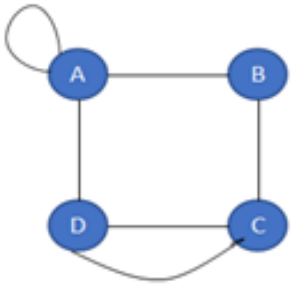


	1	2	3	4	5	6	7
1	0	28	0	0	0	10	0
2	28	0	16	0	0	0	14
3	0	16	0	12	0	0	0
4	0	0	12	0	22	0	18
5	0	0	0	22	0	25	24
6	10	0	0	0	25	0	0
7	0	14	0	18	24	0	0

Figure 4.36: Graph Adjacency matrix Representation

In the above diagram, edge exists between 1 to 2 edge, so in the matrix 1 to 2 entry is 28. Edge does not exist between 1 to 3, so in the matrix 1 to 3 entry is 0.

Example (Multi Graph)



	A	B	C	D
A	2	1	0	1
B	1	0	1	0
C	0	1	0	2
D	1	0	2	0

Figure 4.37 : Graph Adjacency matrix Representation

If self-loop exists then its count is 2 because of following reason-
A(source)→A (destination or sink) + A (destination or sink) →A(source)=1+1=2

If multiple edge exists then it counts 2 and if single edge exists then it counts only 1.

4.1.8. Graph primitive Operations

Let us suppose that there are four persons A, B, C and D stores in Array v of characters
v [4] = {‘A’ , ‘B’ , ‘C’ , ‘D’}

Let us suppose there is a 2-dimensional array named e, which stores connections between persons.

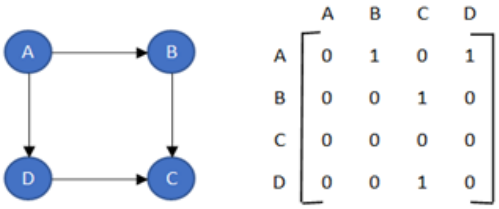


Figure 4.38 : Graph operations

```
e[4][4]={ {0 ,1 ,0, 1},
          {0 , 0,1, 0},
          {0 , 0,0, 0},
          {0 , 0,1, 0}  };
```

Let us assume that the index of the person in v array represents the unique number of that person.

ALGORITHM uniqueNumber(ver)

```
// name as Input and returns it’s unique no equivalent
BEGIN:
    FOR i=0 TO N DO
        IF v[i]==ver THEN
            RETURN i
    RETURN -1
END;
```

Explanation-

This algorithm takes the vertex name as input and returns its index number. For example, if the input parameter is 'B' then the algorithm returns 1, which is the index number of vertex 'B.'

ALGORITHM isAdjacent(ver1,ver2)

```
// Returns true if two vertices are adjacent.
BEGIN:
        i=uniqueNumber(ver1)
        j=uniqueNumber(ver2)
        IF i == –1 || j== –1 THEN
                RETURN false
        RETURN e[i][j]
END;
```

Explanation-

This algorithm takes two vertices as parameters and returns the corresponding value in e[i][j] if vertex names exist in a given Graph. For example, if the input parameter is 'A' and 'B', it returns 1.

ALGORITHM displayAdjacentNode(v)

```
// Displays all adjacent vertices for the given vertex
BEGIN:
        i=uniqueNumber(v)
        IF i == –1  THEN
                RETURN
        FOR J = 0 TO N DO
                IF e[i][j] != 0 THEN
                        WRITE(v[j])
END;
```

Explanation-

This algorithm takes the vertex name as a parameter and displays all its adjacent vertices. For example, if input parameter is 'A', it displays 'B' and 'D'.

Complexity-

If uniqueNumber() is implemented using hash table then isAdjacent() takes constant time. This is one of the biggest advantage of adjacency matrix implementation.

Power of Adjacency Matrix-

Let us suppose that A be the adjacency matrix of given Graph then A to the power of N (A^N) gives the number of paths of length N from V_i to V_j.

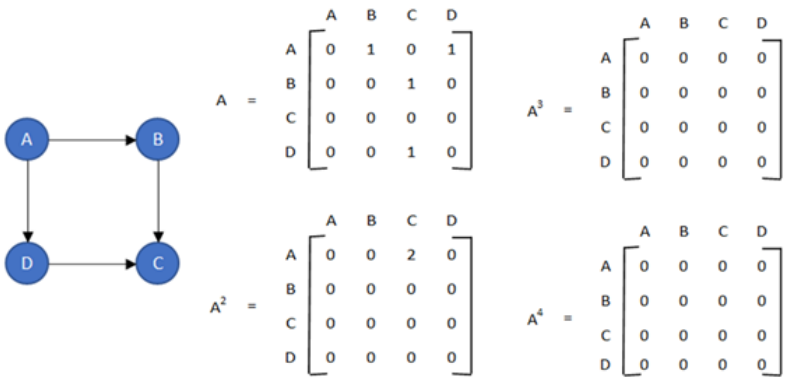


Figure 4.39 : Adjacency Matrix

A^2_{AC} = 2, so there is two paths of length 2 from A to C.

First Path = A->B->C

Second Path= A->D->C

Path Matrix-

Path matrix is defined as of order V x V, where V is the number of vertex defined as-

A[i][j] = { 1 if there is a path from Vi to Vj ; 0 otherwise

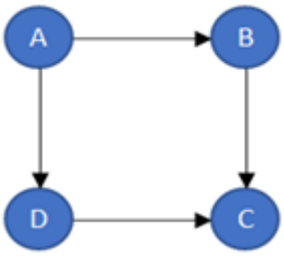


Figure 4.40 : Path Matrix

Steps- 1- Find Adjacency matrix A of given graph.

Step2- $B^N=A + A^2 + \ldots + A^N$

Step3- Path matrix is defined as=

{ 1 for all non zero value of in B^N

P[i][j]=

0 otherwise

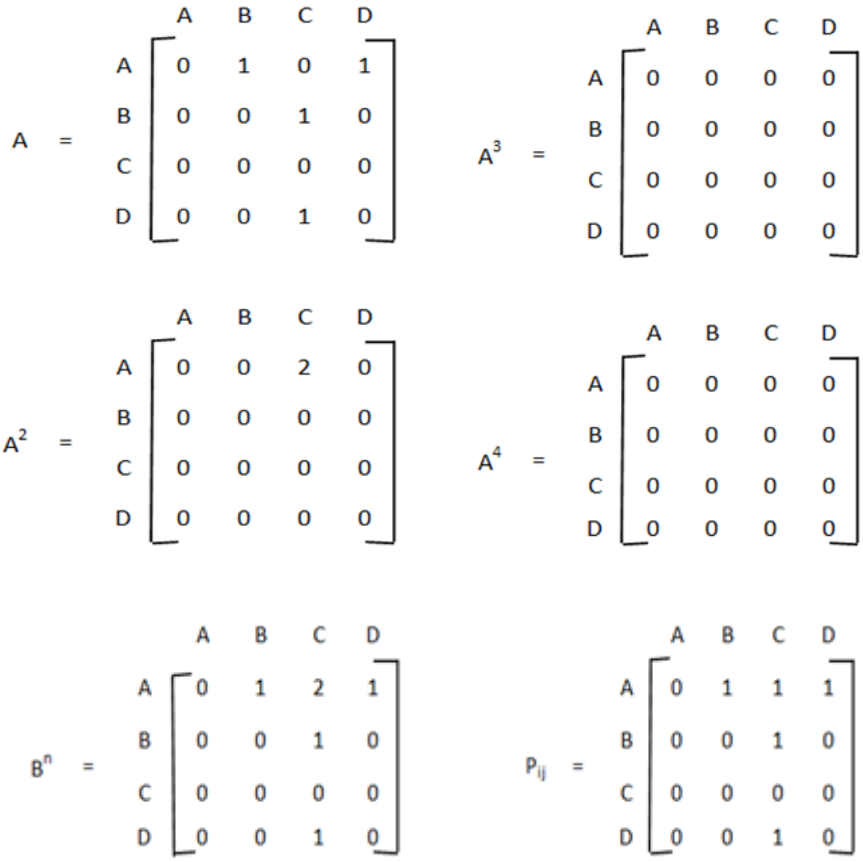


Figure 4.41 : Explanation of Path MAtrix

B) Incidence Matrix-

Incidence matrix of order V x E, where V is the number of vertex and E is number of edges defined as-

Im[i][j] = 1 if the j^{th} edge E_j is incident on i^{th} vertex V_i

0 otherwise

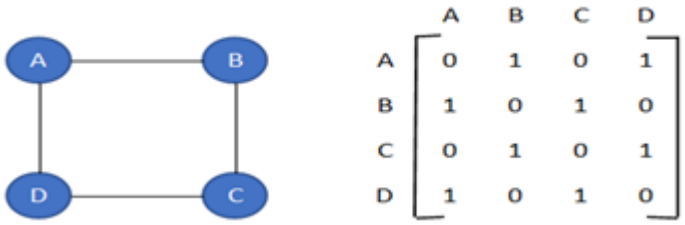
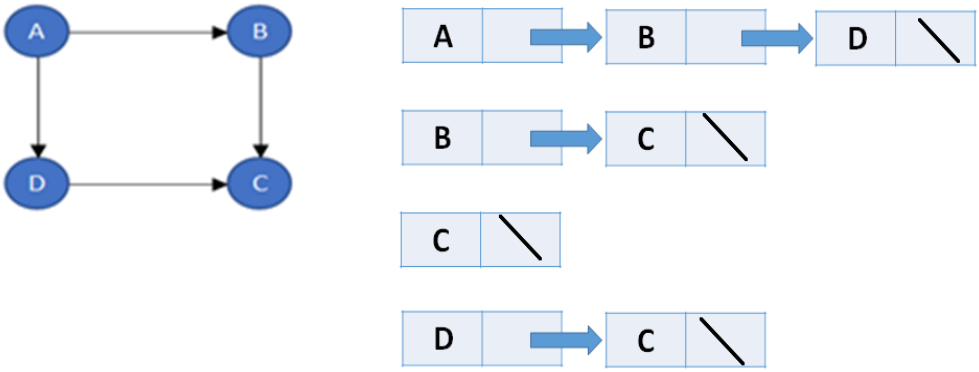


Figure 4.42. : Graph Incidence Matrix

2-Adjacency List Representation to be formatted

Adjacency list defined as – A link list for each adjacent node of the given vertex.



The node contains the data field and next field.

Data	Next

Another node named vertex contains data field and node field.

Data	Node Head

```
ALGORITHM Insert (v[ ], n)   // Algorithm performs Insertion operation in a graph
BEGIN:
        FOR i=1 TO n DO
                v[i].data=character      //read character
                FOR j = 1 TO m          //read m for adjacent node for each node
                        IF v[i].HEAD==NULL
                                v[i].HEAD=P
                                P→data=item
                                P→next=NULL
                                RETURN P
                        ELSE
                                Q=v[i].HEAD
                                WHILE Q→next != NULL
                                        Q=Q→next
                                P→data=item
                                P→next=NULL
                                Q→next=P
                                RETURN v[i].HEAD
END;
```

Explanation-

In this algorithm, input parameter v [ ] is the array of structure and structure element contains data (char type) and head (a pointer which is node type). P and Q are node type structures (node type structure contains char type data and address field of type node).

ALGORITHM Delete(v[ ], n)

```
// Algorithm performs deletion operation on graph
BEGIN:
        FOR i=1 TO n DO
                FOR j=1 TO m
                        P= v[i].HEAD
                        WHILE P!=NULL
                                Q=P
                                P=P→next
                                Q→next=NULL
                                FREE(P)
                                P=NULL
        END;
```

ALGORITHM isAdjacent(ver1,ver2)

```
//Algorithms returns true if two vertices are adjacent
BEGIN:
        i=uniqueNumber(ver1)
        j=uniqueNumber(ver2)
        IF i ==-1 || j==-1 THEN
                RETURN false
        START= v[i].HEAD
        WHILE START !=NULL DO
                IF START→DATA==j THEN
                        RETURN true
                START=START→next
        RETURN false
END;
```

ALGORITHM displayAdjacentNode(ver)

```
// Algorithm displays all adjacent vertices
BEGIN:
        i=uniqueNumber(ver)
        IF i == – 1 THEN
                RETURN
        START= v[i].HEAD
         WHILE START !=NULL DO
```

WRITE V[START]→data
START=START→next
END;

Complexity- This algorithm takes O (n x v), where n is the number of vertices and v is the number of an adjacent node.

Map vertex into hash table

In the adjacency list representation of Graph, assign index number as the unique number for each vertex for convenience. In real life vertex name is not represented by single character. Take the example of city names. For the storage or implementation point of view big names are not suitable. To get rid of this problem, convert the vertex name into numbers and map it into the hash table.

Method-

Let us take an example Allahabad, and the hash table size is 100.

Sum the ASCII value of each character. If the sum is more than 99 then again sum each digits until it is less than or equal to 99

This is the simplest method to map city names into hash table. You can choose any method according to your requirement.

Google also maps city names into a hash table by taking the longitude and latitude of the city name.

Set Representation

An algebraic structure G(v, e) where v is a non-empty set of vertices and e is the non-empty set of edges called graph. Therefore Graph also represented as-

V = {A, B, C, D}

E = {AB, BC, AC, CD}

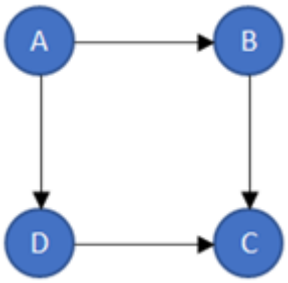


Figure 4.43: Representation of Graph

4.1.9. Multiple Choice Questions (Gate/Company based Questions)

1	Which of the following is an advantage of adjacency list representation over adjacency matrix representation of a graph?
A	In adjacency list representation, space is saved for sparse graphs.

B	DFS and BSF can be done in O(V + E) time for adjacency list representation. These operations take O(V^2) time in adjacency matrix representation. Here is V and E are the number of vertices and edges, respectively.
C	Adding a vertex in adjacency list representation is easier than adjacency matrix representation.
D	All Of Above
AN	All Of Above

	GATE 2005

2	A sink in a directed graph is a vertex i such that there is an edge from every vertex j ≠ i to i, and there is no edge from i to any other vertex. A directed graph G with n vertices is represented by its adjacency matrix A, where A[i] [j] = 1 if an edge is directed from vertex i to j and 0 otherwise. The following algorithm determines whether there is a sink in graph G.
	i = 0
	do {
	j = i + 1;
	while ((j < n) && E1) j++;
	if (j < n) E2;
	} while (j < n);
	flag = 1;
	for (j = 0; j < n; j++)
	if ((j! = i) && E3)
	flag = 0;
	if (flag)
	printf("Sink exists");
	else
	printf("Sink does not exist");

A	(A[i][j] && !A[j][i])
B	(!A[i][j] && A[j][i])
C	(!A[i][j] | | A[j][i])
D	(A[i][j] | | !A[j][i])
AN	(A[i][j] | | !A[j][i])

4.1.10. Competitive Coding Problem:

Edge Existence Problem

Problem Statement

You have been given an undirected graph consisting of *N* nodes and *M* edges. This Graph can consist of self-loops as well as multiple edges. In addition, you have also been given *Q* queries. For

each query, you shall be given 2 integers *A* and *B*. You just need to find if there exists an edge between node *A* and node *B*. If yes, print "YES" (without quotes) else, print "NO"(without quotes).

Input Format:
The first line consists of 2 integers *N* and *M*, denoting the number of nodes and edges respectively. Each of the following *M* lines consists of 2 integers *A* and *B* denoting an undirected edge between node *A* and *B*. The next line contains a single integer *Q* denoting the number of queries. The next Line contains 2 integers *A* and *B* denoting the details of the query.

Output Format

Print Q lines, the answer to each query on a new line.

Constraints

$1 <= N < 10^3$

$1 <= M < 10^3$

$1 <= A,B < N$

$1 <= Q < 10^3$

Approach-

Take a 2-D matrix to store information if an edge exists between two nodes or not.

Initialize matrix with each value 0.

When we get two values in m lines, then we update matrix[a][b] = matrix[b][a] = 1. Both matrices are to be updated because it is undirected graph.

```
ALGORITHM edge (Adjacent[ ][ ],n,e)
BEGIN:
        FOR i=0 TO e DO
                READ x
                READ y
                Adjacent [x][y]=1
                Adjacent [y][x]=1

Read Q
        FOR i=0 TO Q DO
                Read x,y
                IF Adjacent[x][y] == 1 THEN
                        RETURN TRUE
                ELSE
                        RETURN FALSE
END;
```

Problem Statement

You are given an integer n. Determine if there is an unconnected graph with n vertices containing at least two connected components and the number of edges that are equal to the number of vertices. Each vertex must follow one of these conditions:

Its degree is less than or equal to 2- it's a cut vertex.

Note

The Graph must be simple.

Loops and multiple edges are not allowed.

Input format

First line: n

Output format

Print Yes if it is an unconnected graph. Otherwise, print No.

Constraints

1<=n<=100

4.2. Graph Traversal:

Graph traversal is also known as Graph search, is a technique to visit each vertex in the Graph. This traversal can be classified into different categories based on the order of visiting vertices. Tree traversal is also a type of graph traversal. There are two techniques of Graph Traversals, the first is Breadth-First Search, and another is Depth-First Search.

4.2.1. Breadth-First Search (BFS)

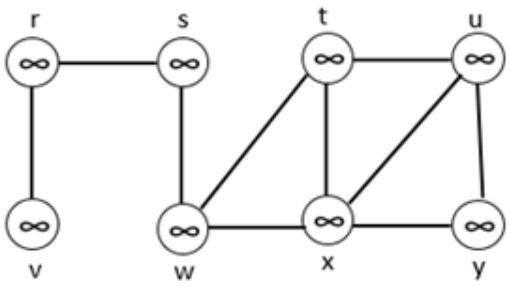
The BFS algorithm is a fundamental search algorithm used to explore the vertices and edges of a graph. The BFS algorithm is beneficial to finding the shortest path on an unweighted Graph.
A BFS starts at some arbitrary node and explores the neighbour node first before moving to the next level of neighbours. This way, it explores the Graph in layer fashion; that is why it is also known as a layered ordered traversal algorithm. It explores all neighbours of all the nodes and ensures that each node is visited exactly once and that no node is visited twice.

It also uses a first-in, first-out queue Q to manage the set of undiscovered adjacent nodes of discovered nodes.

The BFS algorithm finds a tree embedded in the Graph; this is called the BFS tree.
It initially contains only its root, which is the source vertex.

Search discovers an adjacency list of an already discovered vertex u, then the vertex v and the edge (u,v) are added to the tree.
u is the predecessor or parent of v in the BFS tree since a vertex is discovered at most once it has at most one parent.
Let us discuss the implementation of BFS on an undirected graph. In BFS, each vertex is colored in white, gray, or black.

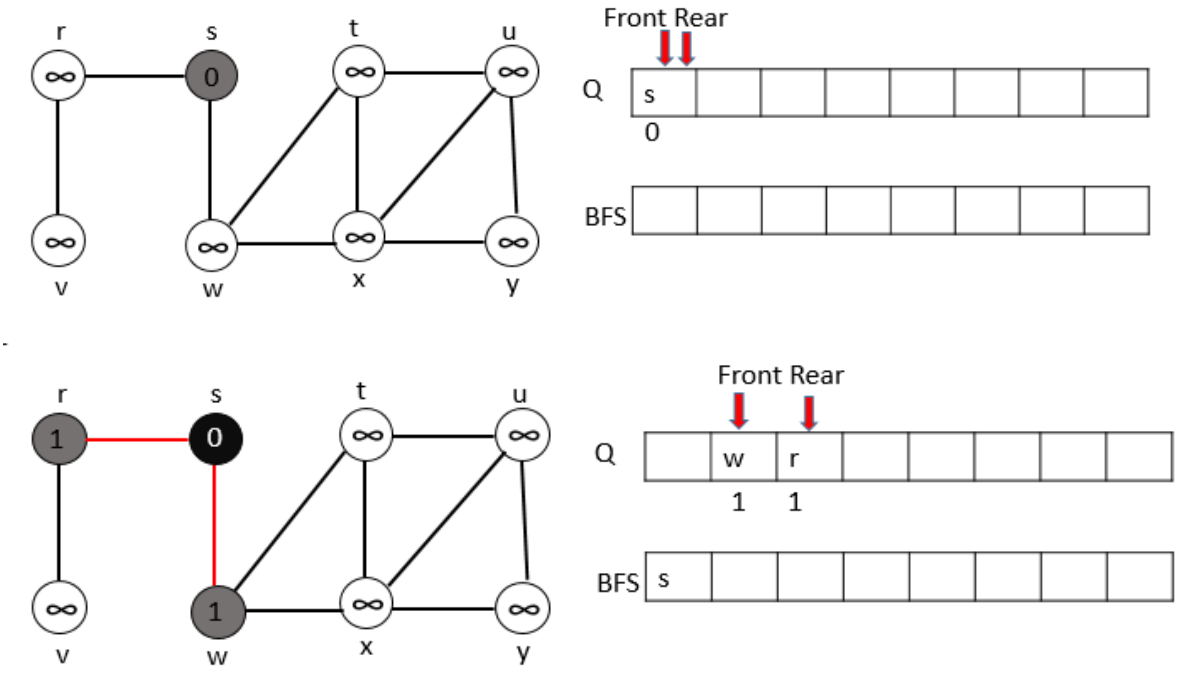
WHITE Color: not yet discovered vertices
GRAY Color: discovered vertices but still some undiscovered adjacent vertices (in the queue)
BLACK Color: discovered vertices and all adjacent vertices discovered (deleted from the queue)
Initially, all edges are black in color, and the edges of the produce BFS tree are red in color. The value of distance appears within each vertex. The size of queue Q is equal to the number of vertices in the Graph to manage the set of grey vertices. Vertex distance appears below vertices in the queue.

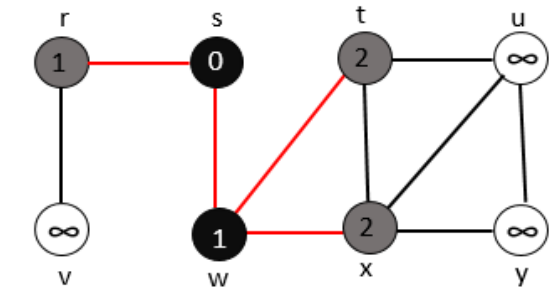


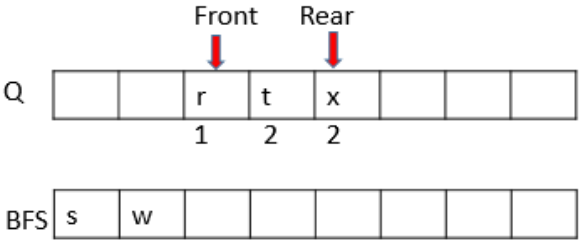
node	adjacent nodes
r	s, v
s	w, r
t	w, x, u
u	t, x, y
v	r
w	s, t, x
x	w, t, u, y
y	x, u

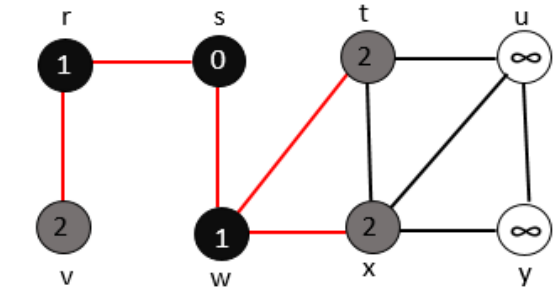
Table1 . Adjacency List for above-undirected Graph

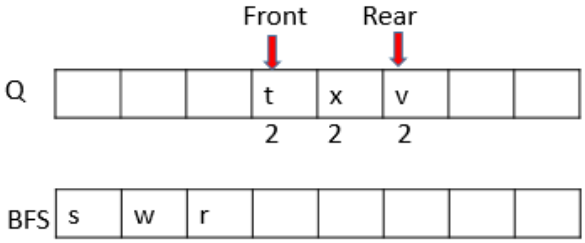
Figure: Initial Queue

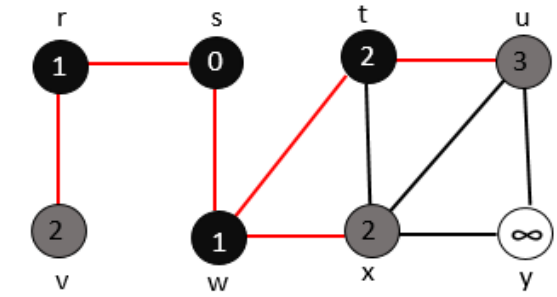


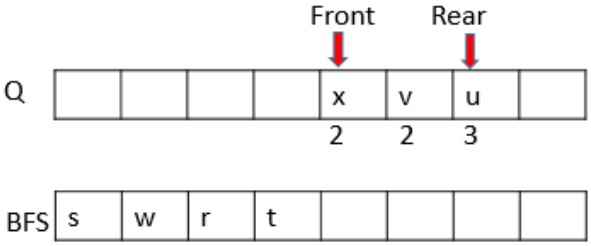

r
s
t
u
1
0
2
∞
∞
1
2
∞
v
w
x
y

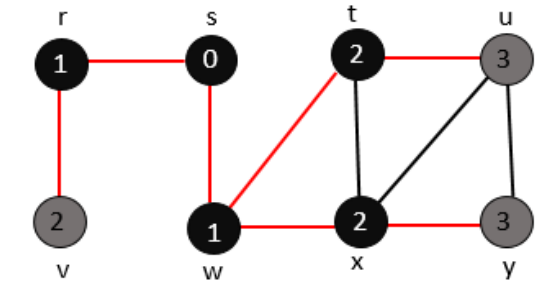

Front
Rear
Q
r
t
x
1
2
2
BFS
s
w

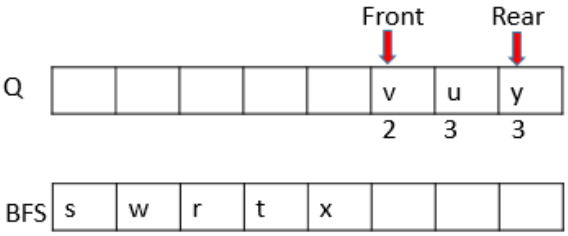

r
s
t
u
1
0
2
∞
2
1
2
∞
v
w
x
y

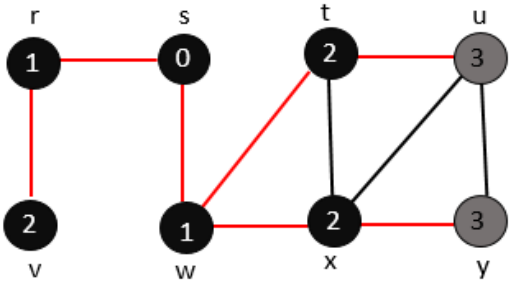

Front
Rear
Q
t
x
v
2
2
2
BFS
s
w
r

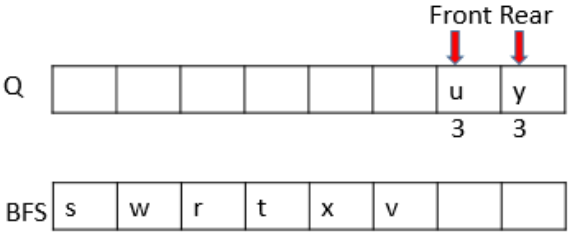

r
s
t
u
1
0
2
3
2
1
2
∞
v
w
x
y


Front
Rear
Q
x
v
u
2
2
3
BFS
s
w
r
t


r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y


Front
Rear
Q
v
u
y
2
3
3
BFS
s
w
r
t
x


r
s
t
u
1
0
2
3
2
1
2
3
v
w
x
y


Front Rear
Q
u
y
3
3
BFS
s
w
r
t
x
v

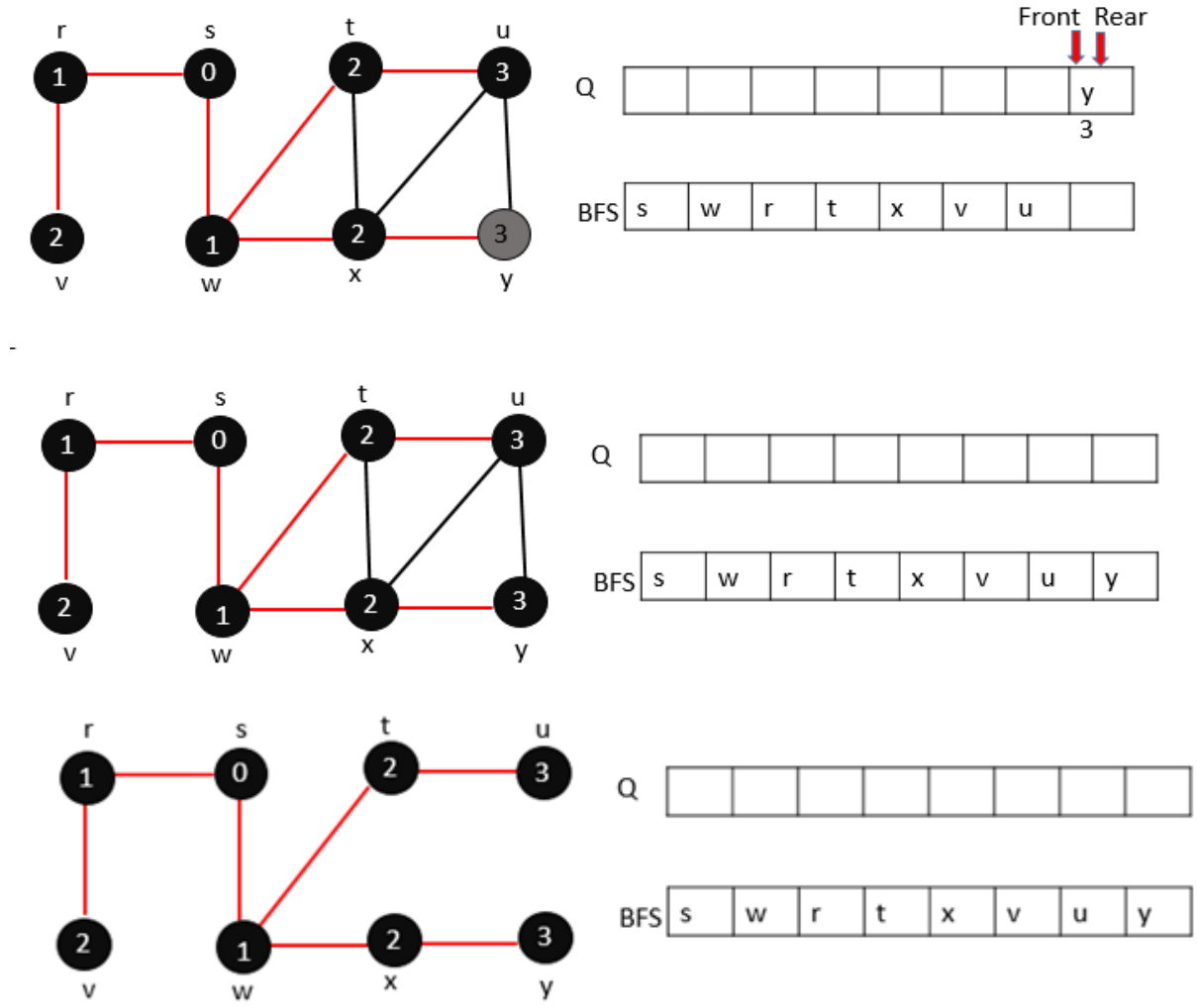


Figure: Operation of BFS on an undirected Graph

Algorithm

Assume that the input graph G= (V,E) is represented using adjacency lists. It attaches three additional attributes to each vertex in the graph colors (color), predecessor (π), and distance (d).
BFS (G, r)

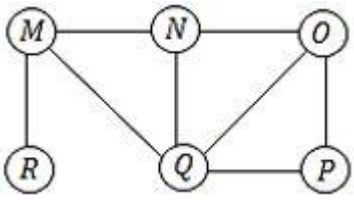
BEGIN:
For all u ∈ V[G] DO
    Color [u] = WHITE
    d[u]=∞
    π[u]=NIL
d[r]=0
Queue Q
Initialize (Q)
EnQueue (Q, r)
Color[r]= GREY

```
WHILE! Empty (Q) DO
        u= DeQueue (Q)
        For all v ∈Adj [u] DO
                IF color[v]= WHITE THEN
                        EnQueue (Q, v)
                        Color [v] = GREY
                        d[v] = d[u] +1
                        π[v] =u
        color[u]= BLACK
        WRITE (u)
END;
```

Complexity

Insert and Delete operation from queue take O(1) time, and so the total time used for these operations is O (V). The total time used in scanning adjacency lists is O(E). The overhead for initialization is O(V), so the time complexity of the BFS algorithm is O(V+E).

4.2.2. Multiple Choice Questions (GATE/ Company Based)

1.	BFS algorithm has been implemented using the queue data structure. Which one of the following is a possible order of visiting the nodes in the Graph below? 
A	MNOPQR
B	QMNROP
C	NQMPOR
D	POQNMR
AN	B
2.	Given two vertices in a graph s and t, which of the two traversals (BFS and DFS) can be used to find if there is path from s to t?
A	Only BFS
B	Only DFS
C	Both BFS and DFS
D	Neither BFS nor DFS
AN	C

3.	Consider the tree arcs of a BFS traversal from a source node W in an unweighted, connected, undirected graph. The tree T formed by the tree arcs is a data structure for computing. GATE CS 2014
A	the shortest path between every pair of vertices.
B	the shortest path from W to every vertex in the Graph.
C	the shortest paths from W to only those nodes that are leaves of T.
D	the longest path in the Graph
AN	B
4.	BFS is started on a binary tree beginning from the root vertex. There is a vertex t at a distance of four from the root. If t is the n-th vertex in this BFS traversal, then the maximum possible value of n is _____ GATE CS 2016
A	15
B	16
C	31
D	32
AN	C
5.	Consider a complete binary tree with 7 nodes. Let A denote the set of first 3 elements obtained by performing Breadth-First Search (BFS) starting from the root. Let B denote the set of first 3 elements obtained by performing Depth-First Search (DFS) starting from the root. The value of |A−B| is _____ .
A	1
B	2
C	3
D	4
AN	A
6.	Which of the following algorithms can be used to most efficiently determine the presence of a cycle in a given graph?
A	DFS
B	BFS
C	Prim's Minimum Spanning Tree Algorithm
D	Kruskal' Minimum Spanning Tree Algorithm
AN	A
7.	Given two vertices in a graph s and t, which of the two traversals (BFS and DFS) can be used to find if there is path from s to t?

A	Only BFS
B	Only DFS
C	Both BFS and DFS
D	Neither BFS nor DFS
AN	C

4.2.3. Depth First Search

DFS algorithm is used to search or traverse graph data structures. As the name suggests, the depth-first search method is to search "deeper" in the Graph.

The algorithm starts at the source vertex or most recently explored vertex U.
Now DFS explores the edges which are adjacent to the vertex U and still has not explored.
Once all the edges of the vertex U have been processed or explored, the search “backtracks” to explore edges leaving the vertex from which U was discovered or which was the predecessor of U.

This process will be repeated continuously till we have exposed all the vertices which can be reached from the given source vertex.
If any vertex remains undiscovered, then DFS picks one of the vertices as a new source and continues to discover the other vertices that are reachable from that given vertex.
Until DFS explored each vertex, this entire process will be repeated since DFS uses backtracking to use a stack data structure to implement it.

In DFS, suppose scan a vertex v as an adjacent node of the already discovered vertex U then it represents the predecessor node U of v as
$\pi[v]=U$
The output of the DFS is the DFS forest which consists of several depth-first trees.

DFS Implementation

The depth-first search uses colour schemes for vertices to indicate their state during the search. Initially, every vertex is considered white; when discovered, it is converted into Gray color. When the node is completely processed i.e., when its adjacency list has been observed completely, then the node is colored into black color.
This method ensures that each vertex finishes as exactly one depth-first tree; therefore all these trees are considered disjoint.
DFS basically defines the timestamps for every vertex. There are two timestamps that are associated with every vertex:
The first timestamp d[u] indicates when vertex u is discovered first time (colored as greyed),
second timestamp f[u] indicates when the search finishes the discovery of all nodes adjacent to u (colored as black).

These timestamps hold the value between 1 and 2 |V |, as these represent one discovery event and one finishing event for every vertex.
For every vertex u,
d[u] < d[f]
Vertex u is WHITE before time d[u], GRAY between time d[u] and f[u], and BLACK after that.
Algorithm: The input is either the undirected or directed graph G. The time variable is declared a global variable that records each vertex's timestamps.

ALGORITHM DFS(G)

```
BEGIN:
FOR each vertex u ∈ V[G] DO
Color[u] = WHITE
π[u] = NIL
time = 0
FOR each vertex u ∈V[G] DO
IF Color[u] == WHITE THEN
DFS_VISIT(G, u)
END;

DFS_VISIT(G, u)
BEGIN:
time =  time + 1                    // white vertex u has just been discovered
d[u] = time
Color[u] = GRAY
FOR each v ∈Adj[u] DO
        IF Color[v] == WHITE THEN
                π[v] = u
                DFS_VISIT(G, v)
Color[u] = BLACK                    //Blacken u, it is finished
time = time + 1
f[u] = time
END;
```

Complexity of DFS Algorithm

Time Complexity: In DFS we traverse each node exactly once, so the running time of DFS will be O(V)as there are total V nodes in the Graph but it does not include the time of exploring adjacent nodes of each called vertex.

Therefore, an additional complexity comes is also calculated, which depends on how to discover all the outgoing edges for each node which depends on the way the Graph is implemented.

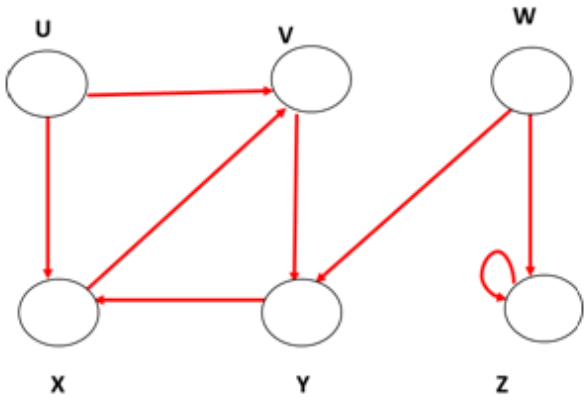
Implemented Graph using adjacency list: Here, each node keeps a list that contains all the adjacent nodes of the given node. So just traverse the adjacency list of a given node to discover all neighbor nodes of that given node.

In the case of a directed graph, the sum of sizes of the adjacency list of all the nodes is E.
Therefore the running time of DFS will be O(V + E) when Graph is directed graph.
In the case of an undirected graph, each edge appear twice so that the time complexity will be O(V + 2E) i.e., almost O(V + E).

If the Graph is implemented as adjacency matrix as VxV array, then for each node, there is a need to traverse an entire row of length V in the matrix. Also, each row in an adjacency matrix corresponds to a node in the Graph. Therefore the running time of DFS will be O(V *V).
Space Complexity

Since stack data structure is used to implement DFS, here stack keeps track of all visited nodes. The worst-case stack could take up to the total number of vertices V. So, the space complexity will be O (V).

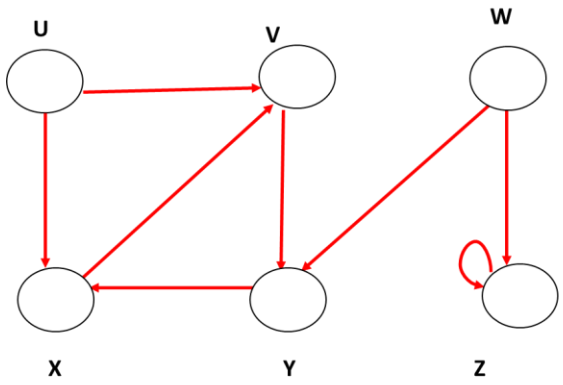
Example: Figure



node	adjacent nodes
u	v, x
v	y
w	y, z
x	v
y	x
z	-

Figure: Graph Adjacency list

Solution:



In initial phase for all the vertices of the graph, initialize the variables as

Color[v] = WHITE

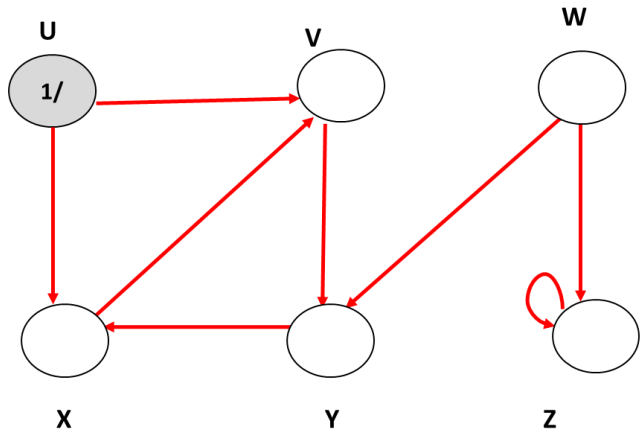
π[v] = NIL

time = 0 (Global)

Now select the initial or starting vertex U.

Figure 1

Step a)

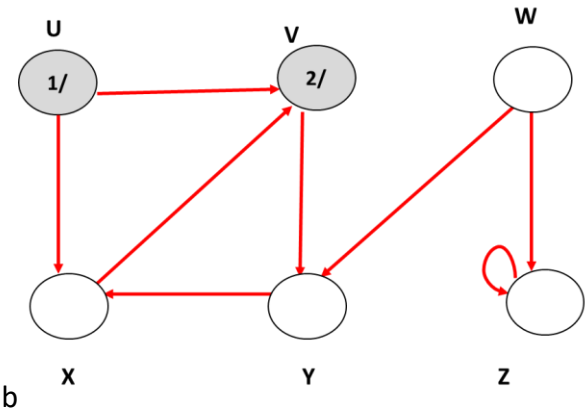


Color[U] = GREY

time = 0 + 1 = 1

d[U] = 1 Here d[U] represents a timestamp when a vertex 'U' is discovered.

Fig-2



π[V] = U

Color[V] = GREY

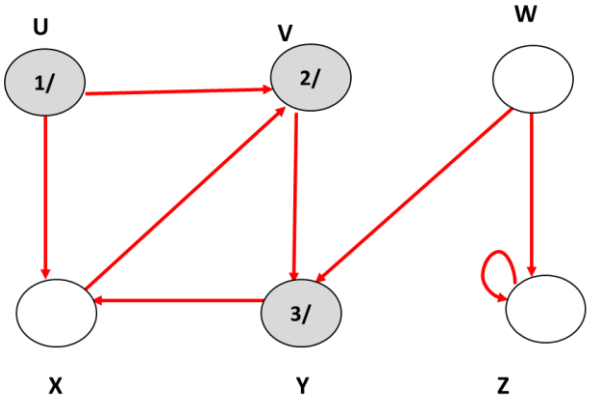
time = 1 + 1 = 2

d[V] = 2

Step b

Fig-3

Step c



π[Y] = V

Color[Y] = GREY

time = 2 + 1 = 3

d[Y] = 3

Fig-4

Step d

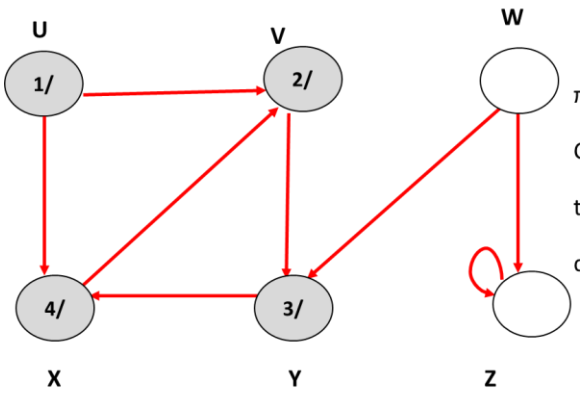
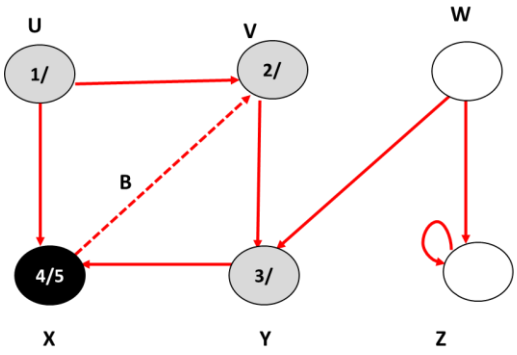


Fig-5

Step e



Here when we extend the visit from vertex X to vertex V then it can be shown

Vertex V is already discovered and it is an ancestor of vertex X

Also, Vertex V is GREY in color.

Therefore, XV is a back edge. It is represented as dashed line and labelled as **B**.

Also perform the following updation

color[X] = BLACK

time = 4 + 1 = 5

f[X] = 5 Here f[X] represents a timestamp when the processing of vertex 'X is finished.

Fig-6

Step f

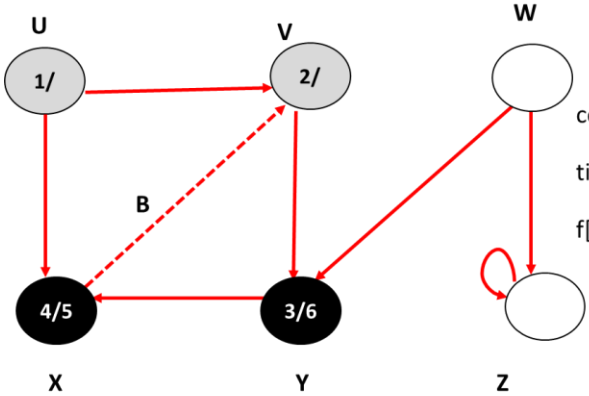
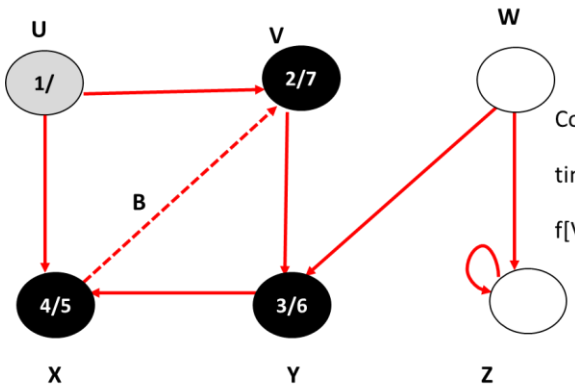


Fig-7

Step g



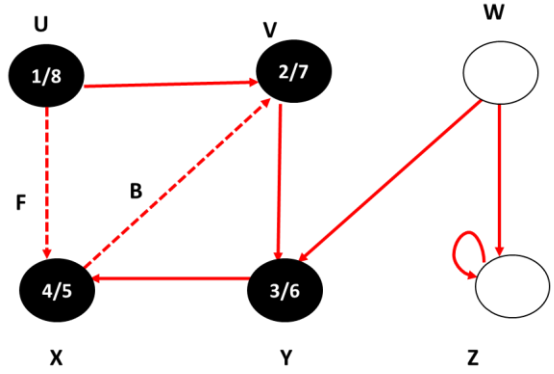
Color[V] = BLACK

time = 6 + 1 = 7

f[V] = 7

Fig-8

Step h



When DFS start the visit from vertex U to vertex X, it finds-

Color(X) = BLACK

d(X) > d(U) i.e. vertex X has already completely processes but vertex U has not finished yet.

Thus, edge UX is a forward edge. It is represented as dashed line and labelled as **F.**

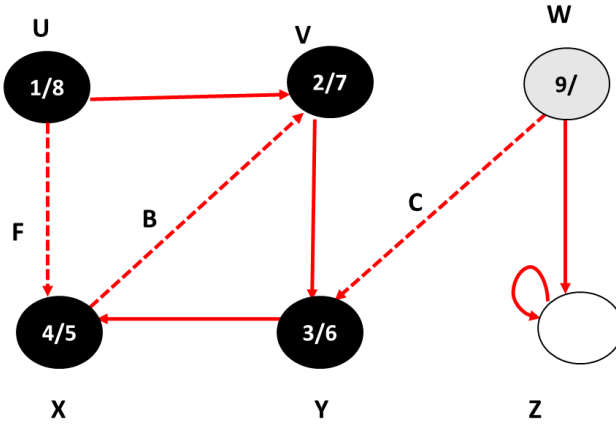
Color[U] = BLACK

time = 7 + 1 = 8

f[U] = 8

Fig-9

Step i



Color[W] = GREY

time = 8 + 1 = 9

d[W] = 9

When DFS start the visit from vertex W to vertex Y, it finds-

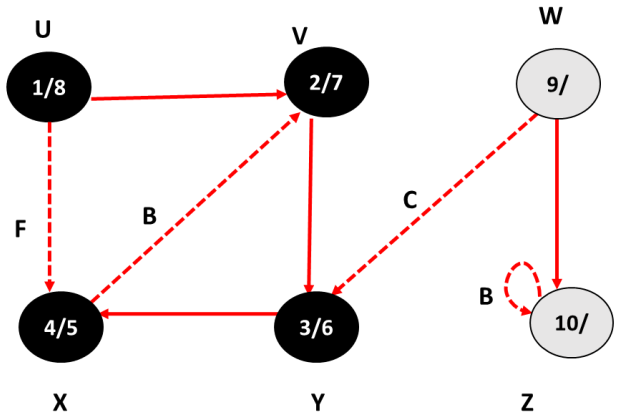
Color(Y) = BLACK i.e. vertex Y is completely processed.

d(Y) < d(W) i.e. vertex Y is discovered earlier than vertex W.

Therefore, edge WY is a cross edge. It is represented as dashed line and labelled as **C.**

Fig-10

Step j



π[Z] = W

Color[W] = GREY

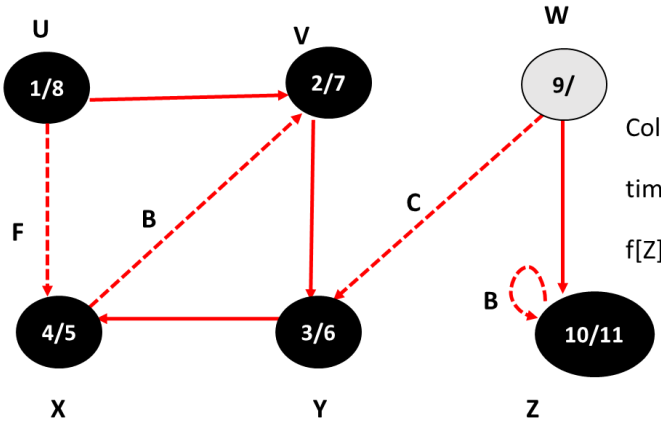
time = 9 + 1 = 10

d[W] = 10

Here vertex Z has self loop so it is represented as Back edge and labelled as **B.**

Fig-11

Step k



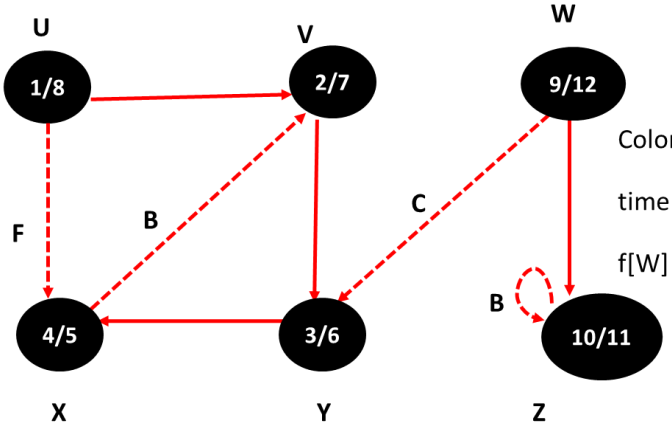
Color[Z] = BLACK

time = 10 + 1 = 11

f[Z] = 11

Fig-12

Step l



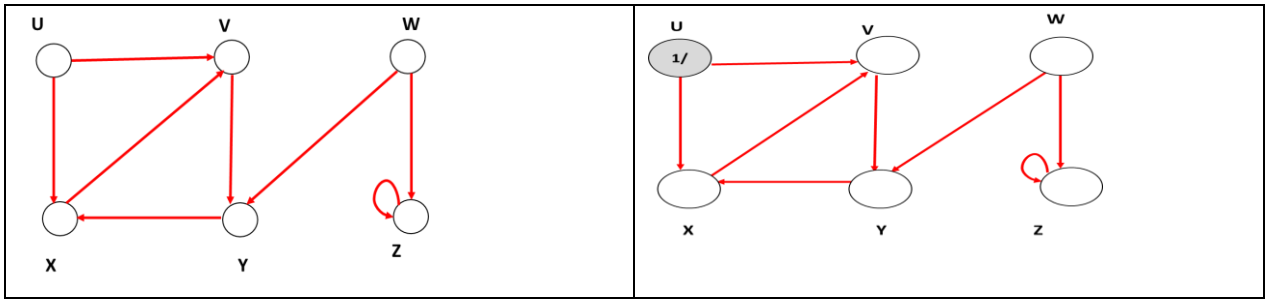
Color[W] = BLACK

time = 11 + 1 = 12

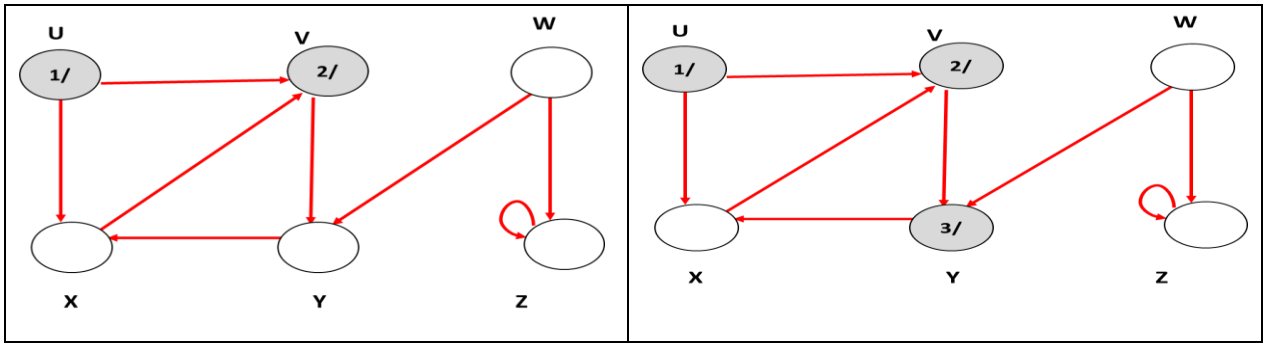
f[W] = 12

Fig-13

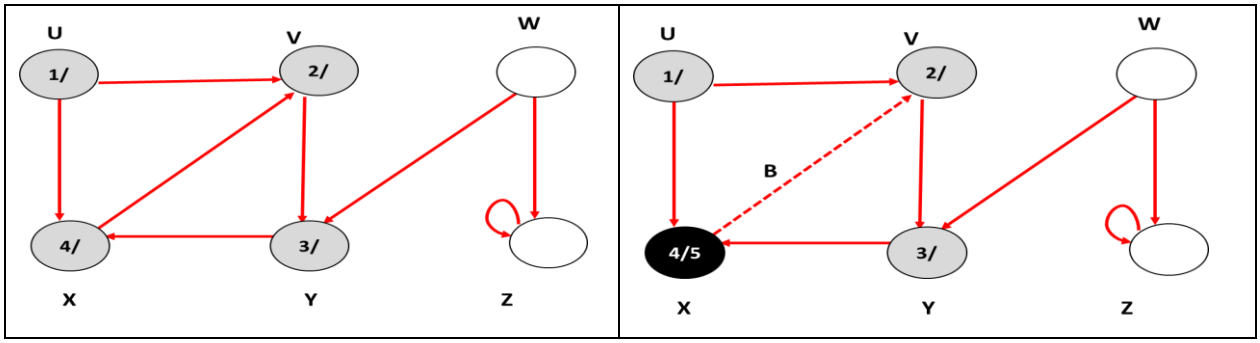
Here we are summarizing the working of the DFS Algorithm using the above example



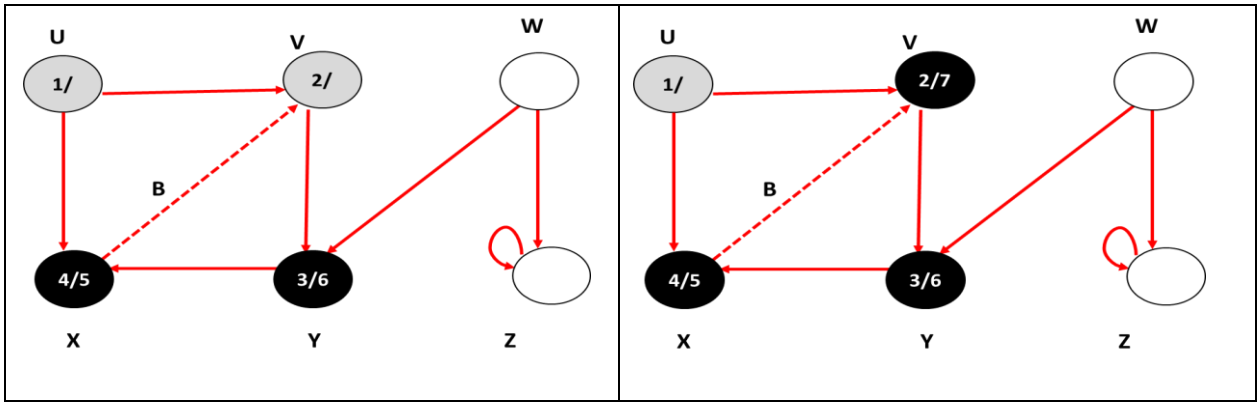
(b)



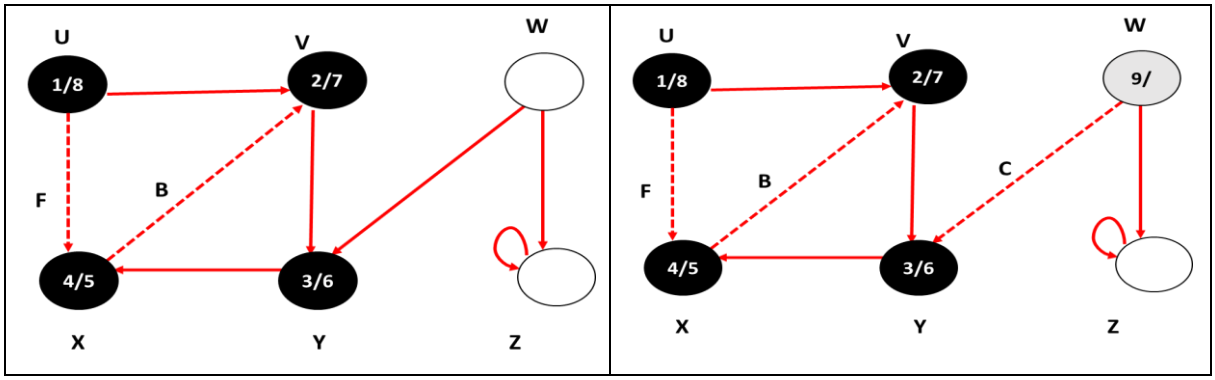
(c) (d)



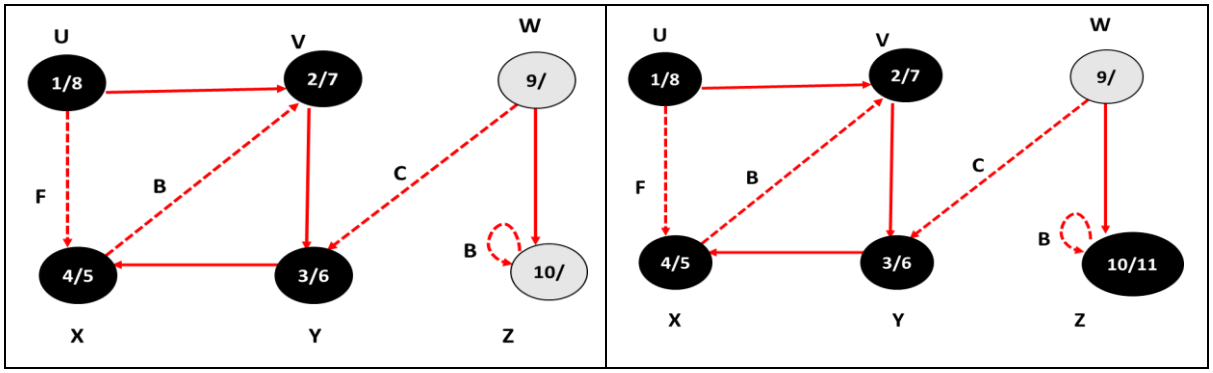
(e) (f)



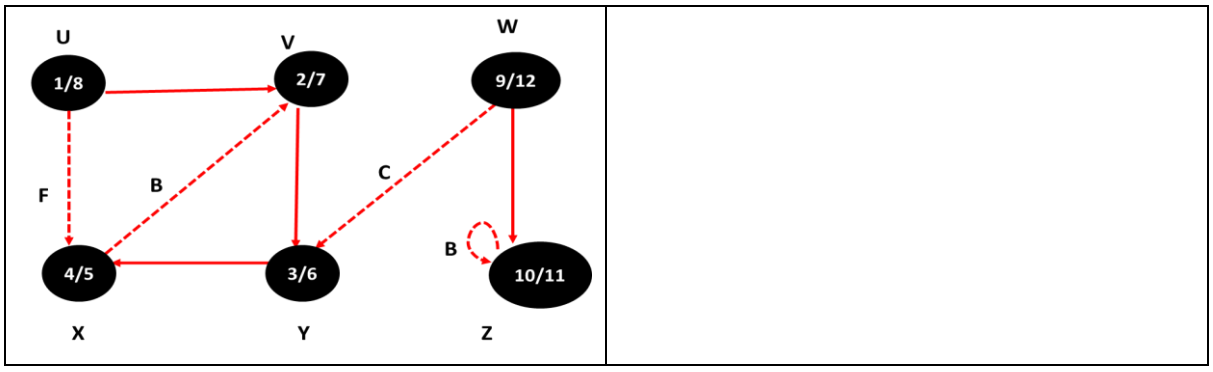
(g) (h)



(j)



(k) (l)



(m)

4.2.4. Multiple-choice Questions on DFS

1.	Let G be a graph with n vertices and m edges. What is the tightest upper bound on the running time of Depth First Search on G, when G is represented as an adjacency matrix? (GATE CS 2014)
A	O(n)
B	O(m+n)
C	$O(n^2)$
D	O(mn)
AN	C

2.	Let G be a simple undirected graph. Let TD be a depth-first search tree of G. Let TB be a breadth first search tree of G. Consider the following statements. (I) No edge of G is a cross edge with respect to TD. (A cross edge in G is between two nodes, neither of which is an ancestor of the other in TD). (II) For every edge (u, v) of G, if u is at depth i and v is at depth j in TB, then $\|i - j\| = 1$. Which of the statements above must necessarily be true?
A	I only
B	II only
C	Both I and II
D	Neither I nor II
AN	C
3.	Which of the following algorithms can be used to most efficiently determine the presence of a cycle in a given graph?
A	Depth-First Search
B	Breadth-First Search
C	Prim's Minimum Spanning Tree Algorithm
D	Kruskal' Minimum Spanning Tree Algorithm
AN	A
4.	Traversal of a graph is different from tree because
A	There can be a loop in Graph, so we must maintain a visited flag for every vertex
B	DFS of a graph uses the stack, but in-order traversal of a tree is recursive
C	BFS of a graph uses a queue, but a time-efficient BFS of a tree is recursive.
D	All of the above
AN	A
5.	Let G be an undirected graph. Consider a depth-first traversal of G, and let T be the resulting depth-first search tree. Let u be a vertex in G and let v be the first new (unvisited) vertex visited after visiting u in the traversal. Which of the following statements is always true? (GATE CS 2000)
A	{u,v} must be an edge in G, and u is a descendant of v in T
B	{u,v} must be an edge in G, and v is a descendant of u in T
C	If {u,v} is not an edge in G then u is a leaf in T
D	If {u,v} is not an edge in G then u and v must have the same parent in T
AN	C

6.	Is the following statement true/false? If a DFS of a directed graph contains a back edge, any other DFS of the same Graph will also contain at least one back edge.
A	True
B	False
AN	A
7.	Given two vertices in a graph s and t, which of the two traversals (BFS and DFS) can be used to find if there is path from s to t?
A	Only BFS
B	Only DFS
C	Both BFS and DFS
D	Neither BFS nor DFS
AN	C
8.	If the DFS finishing time f[u] > f[v] for two vertices u and v in a directed graph G, and u and v are in the same DFS tree in the DFS forest, then u is an ancestor of v in the depth first tree.
A	True
B	False
AN	B

4.2.5. Coding Questions on Graph Traversal

Depth First Search (DFS) Iterative Implementation.
Depth First Search (DFS) Recursive Implementation.
Generate list of possible words from a character matrix.
Find all occurrences of the given string in a character matrix.
Find path from source to destination in a matrix that satisfies given constraints

4.3. Disjoint Set Data Structure:

4.3.1. Introduction to Disjoint Set Data Structure:

Let us consider a scenario where we are going to organize a meeting with a variety of persons in order to discuss a particular problem and following operations to be performed on them.

Add a new relation, (x, y), where the opinion of x matched with opinion of y
Find if opinion of x matches with opinion of y either directly or indirectly.
Given ten attendees, namely,
Amit , Bunty, Chotu, Deepak, Ekta, Farukh, Gopal, Hari, Indu, Jaya
Following relationships are given whose opinions are matched.

Amit<->Bunty
Bunty <->Deepak
Chotu<->Farukh
Chotu<->Indu
Jaya<->Ekta
Gopal<->Jaya

If the problem in the hand is that if the opinion of Amit matched with that of Deepak or not.
Make 4 groups based on given relationship-
G1 = {Amit, Bunty, Deepak}
G2 = {Chotu, Farukh, Imam}
G3 = {Ekta, Gopal, Jaya}
G4 = {Hari}

Problem: To find whether Amit and Deepak belong to the same group or not, i.e., to find if the opinion of Amit matched with the opinion of Deepak or not (either directly or indirectly)
Solution: Divide the persons into different sets according to the groups in which they fall.

Procedure:

step1
Initially, all elements belong to different sets. After working on the given relations, select a member as a representative.
Step2
If representatives of two groups are the same, then they will become friends, i.e. opinions are matched.

ALGORITH find(i)

```
BEGIN:
        IF parent[i] == i THEN
                RETURN i
        ELSE
                RETURN find(parent[i])
END;
```

Union:

```
ALGORITHM union( i,  j)
BEGIN:
        i_represent= find(i)
j_represent = find(j)
        parent[i_represent] = j_represent;
END;
```

When above algorithm is processed then we get the four connected component as-
{Amit, Bunty, Deepak}
{Chotu, Farukh, Indu}
{Ekta, Gopal, Jaya}
{Hari}

4.3.2. Definition of Disjoint Set:

The disjoint set data structure maintains a set of elements into a number of disjoint sets. In another term a disjoint set data structure maintains a collection $s = \{s_1, s_2, s_3, \ldots, s_n\}$ into a number of disjoint sets. Each set is identified by its representative (one of the members of the same set).

Operations

Make Set(x)-
Creates a new set whose only member and representative is x.
Union(x, y)-
This operation joins two sets into a single set.
Find(x)-
This operation finds a unique set which contains x.

4.3.3. Application of Disjoint Set Data Structure:

Connected Component (Application of disjoint set data structure)

This is one of the most important applications of disjoint set data structure, which is used to find the connected component in the undirected Graph-

ALGORITHM ConnectedComponent(G)

```
BEGIN:
        FOR each vertex v∈ V(G) DO
                MakeSet(v)
        FOR each edge (u , v)∈E(G) DO
                IF Find(u) != Find(v) THEN
                        Union(u,v)
END;
ALGORITHM SameComponent(u,v)
BEGIN:
        IF Find(u) != Find(v) THEN
                RETURN FALSE
        ELSE
                RETURN TRUE
END;
```

Example:

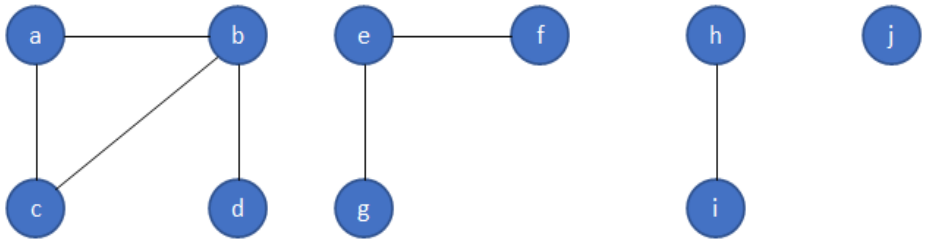


Figure:

Explanation-

Connected component

Step1- Make Set algorithm defines a new set of all vertices. Initially, all vertices are in a different set.

{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}, {j}

Edge Processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}

Step2- Find() algorithm to find the set of edges. If both vertices are in a different set, then union operation performs between both vertices. It happens only when an edge exists between both vertices.

First select {u} = b and {v} = d we can select any vertices. There is an edge between b and d. Both are in a different set, so union operation is performed.

Edge Processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}

Step 3- Select {u} = e and {v} = g There is an edge between e and g. Both are in different set so union operation performed.

Edge Processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}

Step 4- Select {u} = a and {v} = c There is an edge between a and c. Both are in different set so union operation performed.

Edge Processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}

Step 5- Select {u} = h and {v} = i There is an edge between h and i. Both are in different set so union operation performed.

Edge Processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}

Step 6- Select {u} = a and {v} = b There is an edge between a and b. Both are in different set so union operation performed.

Edge Processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}

Step 7- Select {u} = e and {v} = f There is an edge between e and f. Both are in different set so union operation performed.

Edge Processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

Step 8- Select {u} = b and {v} = c There is an edge between b and c. Both are in different set so union operation performed.

Edge Processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

If we select any two vertices, there is no edge between them than we do not consider that.

4.3.4. Multiple Choice Questions (GATE/Company Based)

1	The most efficient algorithm for finding the number of connected components in an undirected graph on n vertices and m edges has time complexity.
A	theta(n)
B	theta(mn)
C	theta(m+n)

D	theta(mn)
AN	theta(m+n)

2	Let G be a simple graph with 20 vertices and 8 components. If we delete a vertex in G, then the number of components in G should lie between ____.
A	8 and 20
B	8 and 19
C	7 and 19
D	7 and 20
AN	7 and 19

	GATE 97
3	Let G be the Graph with 100 vertices numbered 1 to 100. Two vertices i and j are adjacent iff $\|i-j\|=8$ or $\|i-j\|=12$. The number of connected components in G is
A	8
B	4
C	12
D	25
AN	4

4.3.5. Competitive Coding Problem:

Transportation Network

Problem Statement

Problem

The country of Byte land consists of *n* cities. Between any *2* cities, it is possible to have a rail 8 way track and a road.
Railway tracks are bidirectional, meaning if there exists a railway track between *u* and *v* then you can take a train from *u* to *v* as well as from *v* to *u*. Similarly, roads are bidirectional, meaning if there exists a route between *u* and *v* then you can drive from *u* to *v* as well as from *v* to *u*.
2 cities, *u* and *v* are called **railway-connected** if it is possible to travel between *u* and *v* using railway tracks.

2 cities, *u* and *v* are called **road-connected** if it is possible to travel between *u* and *v* using roads.
The transportation network is called **balanced** if for all pairs of cities *u*, *v*:
u,*v* are railway-connected if and only if *u*,*v* are road-connected.

Initially, there are *n* cities and no roads or railways in Byteland. You will be given *q* instructions asking you to build either a railway track or a road between some *2* cities. After each instruction, you must report whether the transportation network is balanced.

Input format

The first line of input will contain *2* integers, *n* and *q*. *q* lines will follow. Each line will contain *3* space-separated integers in one of the following formats:

1 u v : build a railway track between *u* and *v*

2 u v : build a road between *u* and *v*

Output format

You must print *q* lines. The *i*th line contains an answer to the question whether the transport network is balanced after the *i*th instruction. If it is **balanced** print "YES" (without quotes) otherwise, print "NO" (without quotes)

Constraints

$1 \le n \le 10^5$

$1 \le q \le 2 \times 10^5$

$1 \le u, v \le n$

Sample Input

5 8
1 1 2
1 2 3
2 1 3
2 1 2
1 3 4
2 2 5
1 4 5
2 1 4

Sample Output

NO
NO
NO
YES
NO
NO
NO
YES

ALGORITHM F(v1[ ],v2[ ], n)
BEGIN:
FOR i TO N DO

```
        v1[i]=v2[i]=0
END;
```

ALGORITHM Q(q,v1[ ],v2[ ], n)

```
BEGIN:
WHILE q- - DO // q used as condition. If q is nonzero then it is true otherwise false
c=1
        IF num ==1 THEN
                IF v1[u]==0 THEN
                        v1[u]=1
                        c1++
                IF v1[v]==0 THEN
                        v1[v]=1
                        c1++
                IF v2[u]==0||v2[v]==0 THEN
                         WRITE(NO)
                c=0
                ELSE
                        IF v2[u]==0 THEN
                                v2[u]=1
                                c2++
                        IF v2[v]==0 THEN
                                v2[v]=1
                                c2++
                        IF v1[u]==0||v1[v]==0 THEN
                                WRITE(NO)
                                c=0
                IF c THEN   //c is used as a condition. If nonzero it is true, for 0 it is false
                        IF c1!=c2 THEN
                                WRITE(NO)
                        ELSE
                                WRITE(YES)
END;
```

Problem Statement

In competition, N participants are competing against each other for the top two spots. There are M-friendly relations between participants. You will be given two integers L and R in each friendship relation, indicating L and R are friends.

Note

If A & B are friends, B and C are friends, then A, B, C will have the same friend circle.
Two combinations are considered different if either first or second ranker is different.
The same friendship relation can occur multiple times in input; however, L will never be equal to R.

The jury has to decide the winners for the top two spots, but he does not want to select both participants from the same friend circle, so the competition seems fairer. Find the number of ways in which he can do so.
Input format.

The first line contains T denoting the number of test cases.

The first line of each test case contains two space-separated integers N and M, denoting the number of participants and the number of friendly relations.
Next, M lines of each test case contain two integers L and R, indicating a friendly relation between L and R.
Output format
For each test case, print a single line denoting the number of ways in which the jury can choose the top two participants.
Constraints
$1 \leq T \leq 20000$
$2 \leq N \leq 200000$
$1 \leq M \leq 500000$
$1 \leq L,R \leq N$
The Sum of N over all test cases does not exceed 200000 and M will not exceed 500000
Sample Input
2
3 1
1 3
4 2
1 2
3 4
Sample Output
4
8

4.3.6. Strongly Connected Component

In the mathematical theory of directed graphs, a graph is strongly connected if every vertex is reachable from every other vertex. The strongly connected components of an arbitrary directed graph form a partition into sub graphs that are strongly connected. SCC is an important application

of Depth First Search in which finds the components connected with each other using the DFS algorithm.

ALGORITHM: Strongly Connected Component (G)

BEGIN:

DFS(G) // In order to compute finishing time f[u] for each vertex

Compute G^T // reverse the direction of each edge

DFS(G^T) // Start DFS(u) whose finishing time is largest

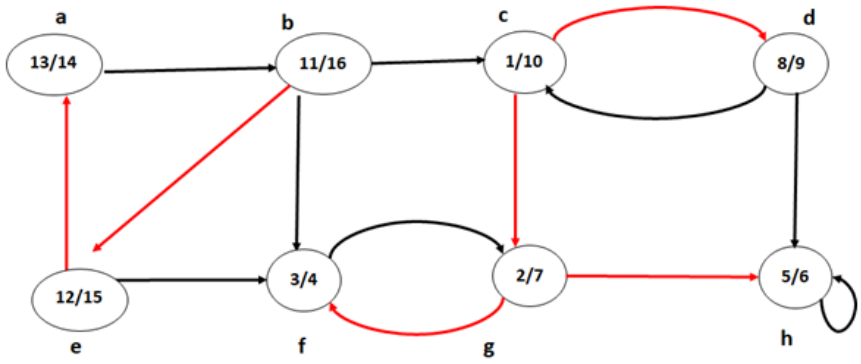
Output the vertices of each tree in depth first forest formed as a separate SCC

END;

Explanation

Step1-Let given Graph is G and apply DFS(G) and store the vertices in an array f[u] in increasing time

Adjacency List Of Given Graph



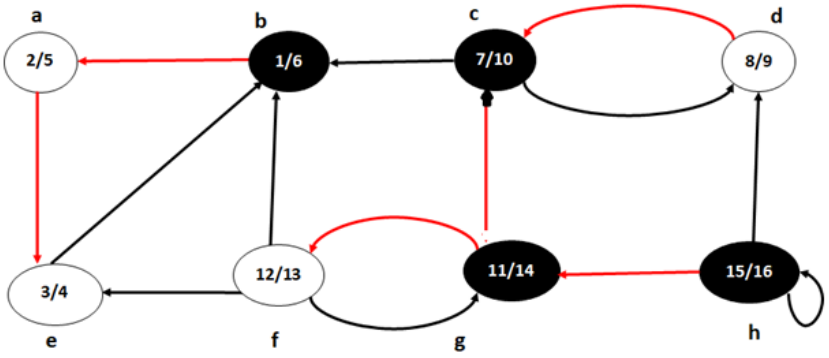
vertex	Adjacent vertex
a	b
b	c,f,e
c	d,g
d	c,h
e	a,f
f	g
g	f,h
h	-

Figure :

Array that stores vertices in ascending sequence of f[u]

f	h	g	d	c	a	e	b

Step2- Reverse the edge directions and recompute DFS(G^T) whose starting vertex is b(maximum finishing time)



Vertex	Adjacent vertex
a	e
b	
c	d
d	c
e	b
f	e,g
g	f
h	g

Figure :

Step3- from above step we get four tree in depth first forest formed namely {a,b, e}, {c,d} , {f,g} and {h} and these are four strongly connected components for given graph G. In order to edge connectivity for these components takes an example –

Let u = {abe} and v{cd}

In this we have path from u to v, i.e. b to c, so there is an edge from u to v.

If v = {fg} then there is also an edge from u to v because there is an edge or path from e to f

Let u = {fg} and v {cd} then there is no edge connectivity between u to v because there is no path from any vertex to any vertex.

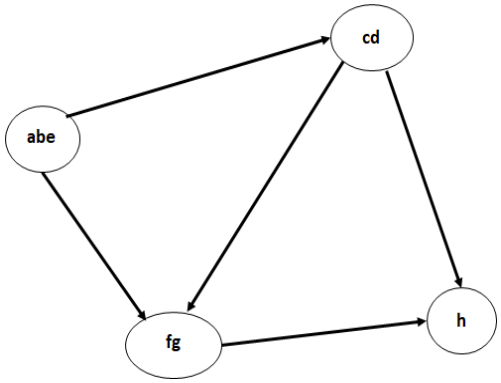
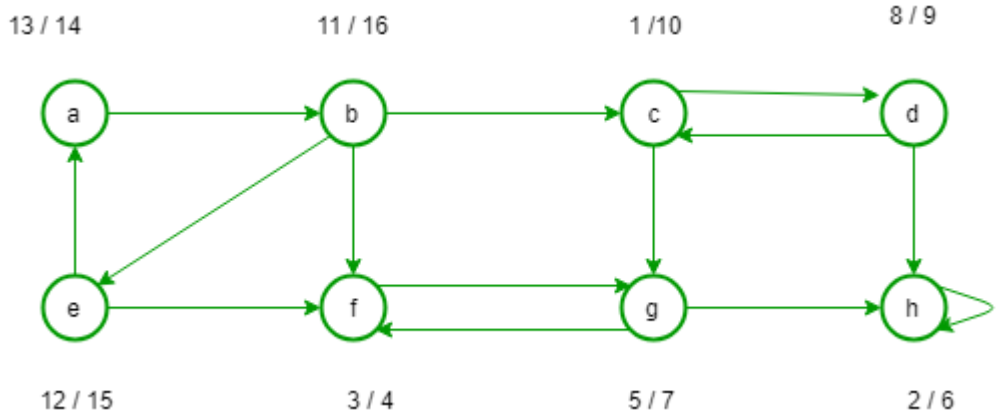


Figure:

Complexity-

The complexity of SCC of given directed graph G(V, E) is theta(V+E) using two DFS() one for G and another one for G^T

4.3.7. Multiple Choice Questions (GATE/Company Based)

	UGC NET CS 2015
1	In the following Graph, discovery time stamps and finishing time stamps of Depth First Search (DFS) are shown as x/y, where x is the discovery time stamp and y is the finishing time stamp. It shows which of the following depth-first forest? 
A	{a, b, e} {c, d, f, g, h}
B	{a, b, e} {c, d, h} {f, g}

C	{a, b, e} {f, g} {c, d} {h}
D	{a, b, c, d} {e, f, g} {h}
AN	{a, b, e} {c, d, f, g, h}

4.3.8. Competitive Coding Problem

GCD On Directed Graph

Problem Statement

You are given a directed graph with *n* nodes and *m* edges. The nodes are numbered from *1* to *n*, and node *i* is assigned a cost ci. The cost of a walk on the Graph is defined as the greatest common divisor of the costs of the vertices used in the walk. Formally, the cost of a walk $v_1, v_2, \cdots, v_k$ is given by $\gcd(cv_1, cv_2, \cdots, cv_k)$. Note that v_i's need not be distinct. Find the cost of the walk with minimum cost.

The walk might consist of a single vertex.

Input

The first line contains two integers, *n*, and *m*.

The next line contains *n* space-separated integers, ith of which is equal to ci

Each of the next *m* lines contains two integers, *u*, and *v*, denoting a directed edge from *u* to *v*.

Output

Print one integer containing the cost of the walk with minimum cost.

Constraints

$1 \le n, m, c_i \le 10^5$

Sample Input

3 2

4 6 8

1 2

2 3

Sample Output

2

4.3.9. Activity Network:

Topological Sort

Consider a situation in which a company plans to launch a product in the market. Several prerequisite activities need to be performed before the final launch. Some of those activities go in parallel, and some will precede the others. We can represent such scenarios as the Vertex and their completion using the directed edge.

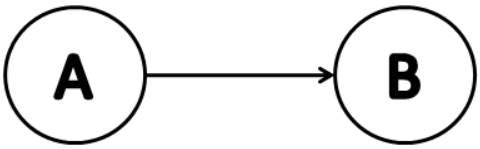


Figure:

There are 2 activities, A & B. Edge from A and B denote
A happens before B
B happens after A
Activity A precedes B
A < B
The Graph showing the representation of activities and their completion is known as Activity Network.

If A < B and B < C, transitively, we can say A < C (A happens before C). C < A here is not possible. This Meaning that such a Graph will never contain a cycle. The activity Network is a Directed Acyclic Graph (DAG) in shape. Activity network is also represented as partially ordered sets (POSET).

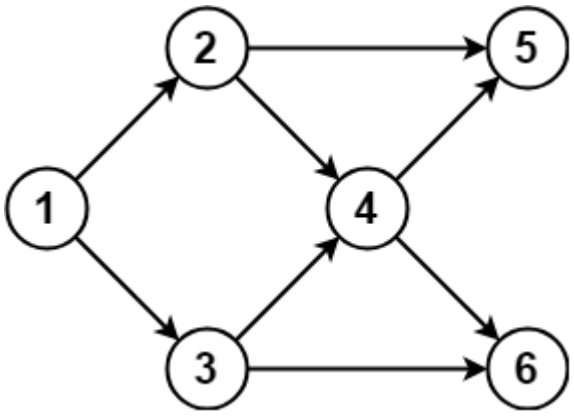


Figure:

We can utilize the POSETS/Activity Network to schedule a series of tasks, such as construction jobs, where we cannot start one task until its prerequisites are completed. We wish to organize the tasks into a linear order that allows us to complete them one at a time without violating any prerequisites.

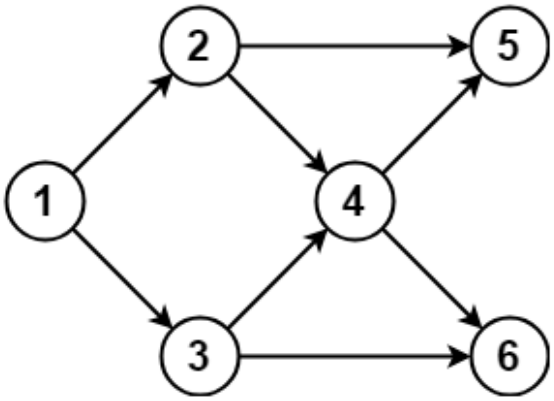
The process of laying out the vertices of a DAG in a linear order to meet the prerequisite rules is called a *topological sort*.

There are two methods by which the topological sort sequence can be found:
Using Queue and in-degree of nodes; and

Using DFS.

1. Using Queue and InDegree

Here we will use the In-degree of the nodes as the base for computing the Topological sort sequence. 0 In-degree of a node shows that the activity represented by this node is independent and can be started immediately. Such nodes are inserted in the Queue. We keep on exploring vertices from the first deleted node from the Queue and decrease the connected node's in-degree by 1. In this process, if 0 in-degree nodes is found, it is added to Queue, and the process repeats until the Queue is empty.



Adjacency List for the Graph	
Vertex	Connection
1	2,3
2	4,5
3	4,6
4	5,6
5	NULL
6	NULL

Step 1: Initialize Queue and insert all the vertices with Zero in-degree in it

Vertex	InDegree					
1	0					
2	1					
3	1					
4	2					
5	2					
6	2					
Queue	1					

Step 2:

Vertex	InDegree					
1	0					
2	1	0				
3	1	0				
4	2	2				
5	2	2				
6	2	2				
Queue		2	3			
	TS	1				

Step 3:

Vertex	InDegree					
1	0					
2	1	0				
3	1	0				
4	2	2	1			
5	2	2	1			
6	2	2	2			

Queue			3			
	TS	12				

Step 4:

Vertex	InDegree					
1	0					
2	1	0				
3	1	0				
4	2	2	1	0		
5	2	2	1	1		
6	2	2	2	1		

Queue				4		
	TS	123				

Step 5:

Vertex	InDegree					
1	0					
2	1	0				
3	1	0				
4	2	2	1	0		
5	2	2	1	1	0	
6	2	2	2	1	0	

Queue					5	6
	TS	1234				

Step 6:

Vertex	InDegree					
1	0					
2	1	0				
3	1	0				
4	2	2	1	0		
5	2	2	1	1	0	
6	2	2	2	1	0	0

Queue						6
	TS	12345				

Step 7:

Vertex	InDegree					
1	0					
2	1	0				
3	1	0				
4	2	2	1	0		
5	2	2	1	1	0	
6	2	2	2	1	0	0

Queue						
	TS	***123456***				

ALGORITHM TopologicalSort (G, Indeg [ ])

BEGIN:
Queue Q
Initialize (Q)
FOR All U ∈V [G] DO
IF Indeg [U] = = 0 THEN
EnQueue (Q, U)
WHILE! Empty (Q) DO
U = DeQueue (Q)
WRITE (U)
FOR All V ∈Adj [U] DO
Indeg [V] = Indeg [V] – 1
IF Indeg [V] = = 0 THEN
Enqueue (Q, U)
END;
Time Complexity: Θ(|V|+|E|)

Space Complexity: Θ(|V|)

2. Using DFS

For performing topological sort using the DFS method as a base, vertices are explored one by one and traversed in-depth recursively until no connections are explorable from the Vertex. The Vertex is then added to the TS Array once all its connections have been explored. Once all the vertices have been covered, TS Array will be printed in the opposite order giving the Topological Sort sequence.

ALGORITHM TSInitialization(G)

```
BEGIN:
TS [|V|]
Visited[|V|]
FOR i=1 TO |V| DO
        TS[i] = – 1
                Visited[i] = 0
        FOR each u ∈  V[G] DO
TopologicalSort(u, TS, Visited, |V|)
FOR i=|V| TO 1 STEP – 1 DO
        WRITE (TS[i])
END;

ALGORITHM TopologicalSort(u, TS[ ], Visited[ ], n)
BEGIN:
        Visited[u] = 1
        For each v ∈ Adj[u] DO
                IF Visited[v] == 0 THEN
                        TopologicalSort(v, TS, Visited, n)
        Append u to TS
END;
```

Time Complexity: Θ(|V|+|E|)

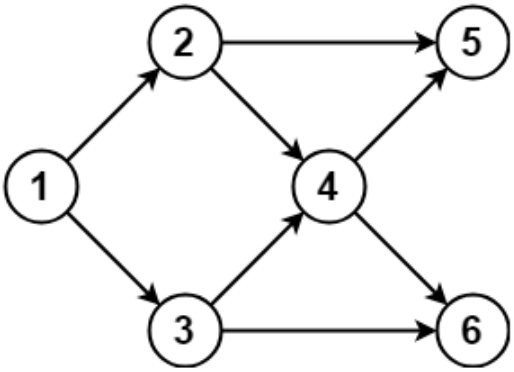
Space Complexity: Θ(|V|)

In the above algorithm, it is assumed that none of the Vertex has been visited initially. This is done by setting up the Vertex values as 0 in the Visited Array. Hence, the size of the Visited array is equal to the number of vertices in the Graph (|V|). We initialize all the elements in the Topological Sort sequence array TS as –1. As the exploration process for a vertex completes, it is added to this array. For each Vertex, the process opens up by calling the function named *TopologicalSort( )*. This

function has the parameters Vertex ‘v', Topological Sort sequence 'TS,' status of visited vertex ‘Visited’, and no. of vertices ‘n’.

The visited status is set as 1 for the given Vertex, and its connections are opened up for exploration. If the adjacent Vertex has not been visited yet, the Vertex is explored using the function *TopologicalSort( ).* Once all the connections are made for exploration, the given Vertex is added to TS Array. The process continues until all the vertices have been covered. TS Array will be printed in the opposite order giving the Topological Sort sequence.

Demonstration:



Adjacency List for the Graph	
Vertex	Connection
1	2,3
2	4,5
3	4,6
4	5,6
5	NULL
6	NULL

STEP 1

	1	2	3	4	5	6
Visited	0	0	0	0	0	0
TS	-1	-1	-1	-1	-1	-1

Initialization

STEP 2

	1	2	3	4	5	6
Visited	1	0	0	0	0	0
TS	-1	-1	-1	-1	-1	-1

First Vertex - 1 visited

STEP 3

	1	2	3	4	5	6
Visited	1	1	0	0	0	0
TS	-1	-1	-1	-1	-1	-1

Vertex Adjacent to 1 i.e. 2 is Visited

STEP 4

	1	2	3	4	5	6
Visited	1	1	0	1	0	0
TS	-1	-1	-1	-1	-1	-1

Vertex Adjacent to 2 i.e. 4 is Visited

STEP 5

1	2	3	4	5	6

	Visited	1	1	0	1	1	0
	TS	-1	-1	-1	-1	-1	-1

Vertex Adjacent to 4 i.e. 5 is Visited

STEP 6

		1	2	3	4	5	6
	Visited	1	1	0	1	1	0
	TS	5	-1	-1	-1	-1	-1

There is no connection from 5. Vertex 5 is added to TS Array

STEP 7

		1	2	3	4	5	6
	Visited	1	1	0	1	1	1
	TS	5	-1	-1	-1	-1	-1

Vertex Adjacent to 4 i.e. 6 is Visited

STEP 8

		1	2	3	4	5	6
	Visited	1	1	0	1	1	1
	TS	5	6	-1	-1	-1	-1

There is no connection from 6 Vertex 6 is added to TS Array

STEP 9

		1	2	3	4	5	6
	Visited	1	1	0	1	1	1
	TS	5	6	4	-1	-1	-1

There is no connection from 4 Vertex 4 is added to TS Array

STEP 10

		1	2	3	4	5	6
	Visited	1	1	0	1	1	1
	TS	5	6	4	2	-1	-1

There is no connection from 2 Vertex 2 is added to TS Array

STEP 11

		1	2	3	4	5	6
	Visited	1	1	1	1	1	1
	TS	5	6	4	2	-1	-1

Vertex Adjacent to 1 i.e. 3 is Visited

STEP 12

		1	2	3	4	5	6
	Visited	1	1	1	1	1	1
	TS	5	6	4	2	3	-1

There is no connection from 3 Vertex 3 is added to TS Array

STEP 13

	1	2	3	4	5	6
Visited	1	1	1	1	1	1
TS	5	6	4	2	3	1

There is no connection from 1

Vertex 1 is added to TS Array

TS array is displayed in opposite order

Topological Sort Sequence: 1, 3, 2, 4, 6, 5

4.3.10. Multiple Choice Questions:

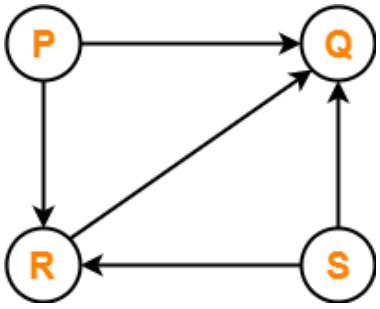
1	Topological sort of a Directed Acyclic graph is?
A	Always unique
B	Always Not unique
C	Sometimes unique and sometimes not unique
D	Always unique if Graph has even number of vertices
AN	C

2	A man wants to go to different places in the world. He has listed them down all. However, there are some places where he wants to visit before some other places. What application of Graph can he use to determine that?
A	BFS
B	DFS
C	Topological Sort
D	Dijkstra's Shortest Path Algorithm
AN	C

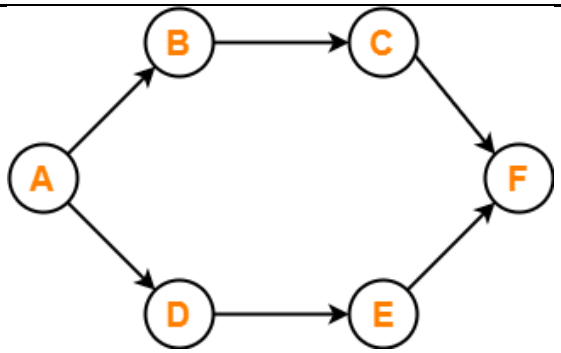
3	Which of the following is not an application of topological sorting?
A	Finding prerequisite of a task
B	Finding Deadlock in an Operating System
C	Finding Cycle in a graph
D	Ordered Statistics
AN	A

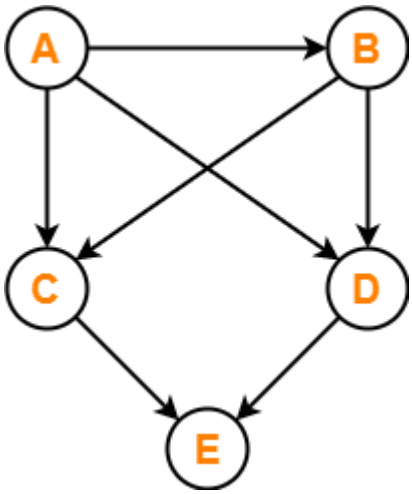
4	In most of the cases, topological sort starts from a node which has _____
A	Maximum Degree
B	Minimum Degree
C	Any degree
D	Zero Degree
AN	D
5	Most Efficient Time Complexity of Topological Sorting is? (V – number of vertices, E – number of edges)
A	O(V + E)
B	O(V)
C	O(E)
D	O(V * E)
AN	A

6	Topological sort is equivalent to which of the traversals in trees?
A	Pre-order traversal
B	Post-order traversal
C	In-order traversal
D	Level -order traversal
AN	B
7	Topological sort can be applied to which of the following graphs?
A	Undirected Cyclic Graphs
B	Directed Cyclic Graphs
C	Undirected Acyclic Graphs
D	Directed Acyclic Graphs
AN	D
8	When is the topological sort of a graph unique?
A	there exists a Hamiltonian path in the Graph
B	In the presence of multiple nodes with indegree 0
C	In the presence of a single node with indegree 0
D	In the presence of a single node with outdegree 0
AN	A

9	Consider the directed graph given below. Which of the following statements is true? 
A	The Graph does not have any topological ordering
B	Both PQRS and SRPQ are topological orderings
C	Both PSRQ and SPRQ are topological orderings.
D	PSRQ is the only topological ordering.
AN	A

10	Consider the following directed Graph-

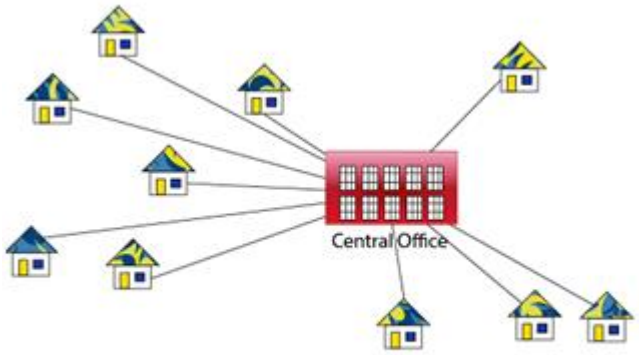
	 The number of different topological orderings of the vertices of the Graph is _____?
A	4
B	5
C	6
D	7
AN	6

11	Find the number of different topological orderings possible for the given Graph- 
A	2
B	3
C	4
D	5
AN	2

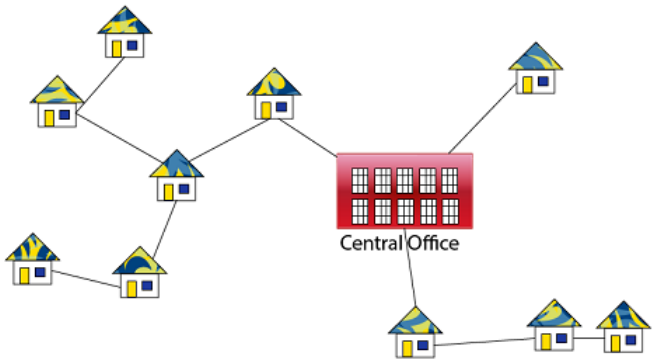
4.4. Spanning Tree:

4.4.1. Introduction to Spanning Tree:

Consider a situation where we can connect a network cable from the main office to various customers to provide a network connection.



A simple approach for wiring netwok cable



A more intellegent approach - minimizing the total cable length for wiring

Figure:

For an undirected graph G, A spanning tree T is a sub graph which includes all the vertices of graph G and having no cycle. A graph may have numerous spanning trees. There will no spanning tree if the graph is not connected.

The minimum spanning tree has various applications including network design, computer network, transportation network, electricity network, water supply network etc.

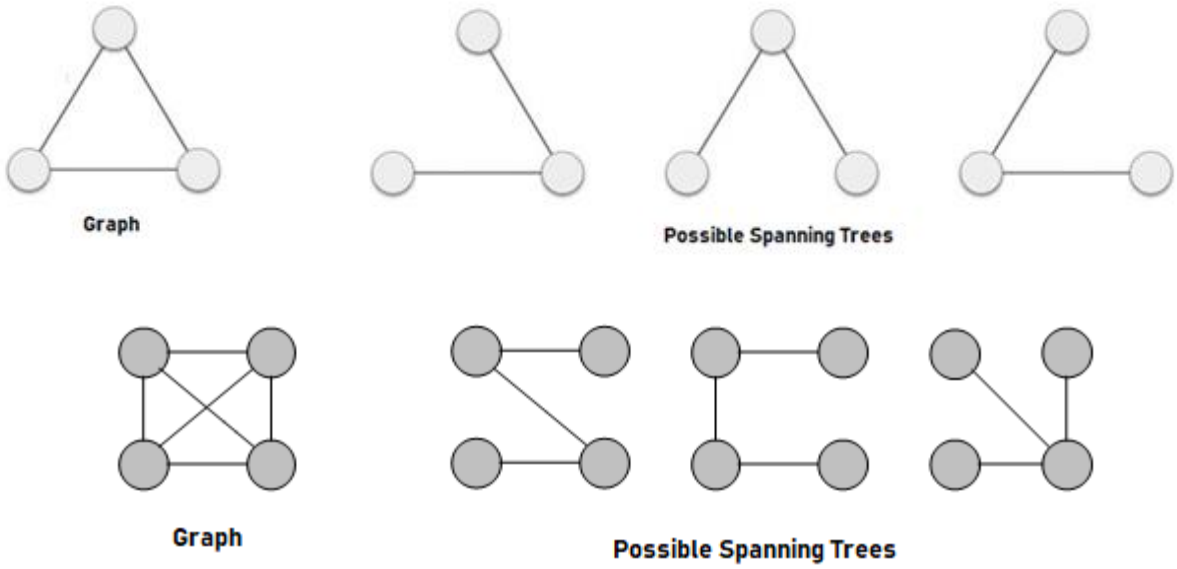


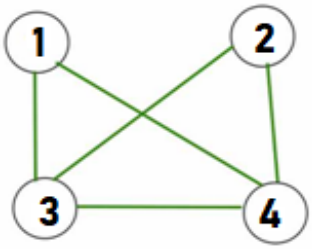
Figure:

The sum of the weights of all edges is weight of the spanning tree. Therefore, different spanning trees have different weights.

Let a complete graph G with N vertices. Then the total number of possible spanning trees will be $N^{(N-2)}$, , i.e., $|V|$ = N. in the graph. e.g., if graph contains 5 vertices then total number of spanning trees may be be $5^{(5-2)} = 125$

The following procedure is used to find the total number of spanning trees in an incomplete graph:-
For the given incomplete graph construct Adjacency Matrix.
The value of degree of nodes is put in all of the diagonal elements with. The degree of node 1 Element is placed at (1,1), and the degree of node 2 element is placed at (2,2). Similarly next all. All non-diagonal elements are replaced with -1.

For any element, determine co-factor. The obtained co-factor is equal to the total number of possible spanning-tree in graph.

Suppose a incomplete Graph	

The Adjacency Matrix for the given incomplete graph	$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$
The co-factor for (1, 1) of the resultant matrix after running STEP 2 and STEP 3 is eight. Hence total number of spanning trees = eight	$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 2 & 0 & -1 & -1 \\ 0 & 2 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix} \end{array}$

4.4.2 Minimum Spanning tree:

Consider a situation where a cable company wants to connect five cities to their network. What is the minimum length of cable through which these cities may be connected?

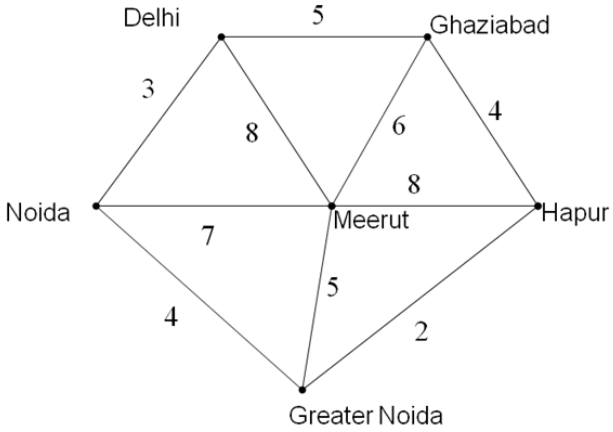


Figure:

The given problem is modelled as a network and now it is problem of finding the minimum spanning tree of given network.

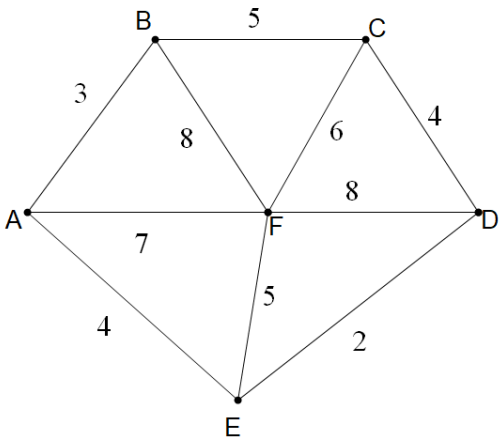
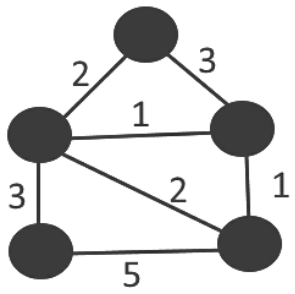


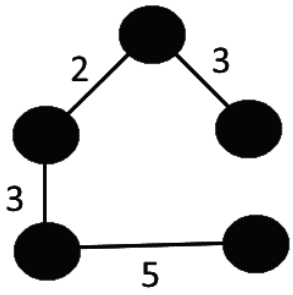
Figure:

The minimum spanning tree is the spanning tree having minimum weight. There may be various minimum spanning trees. In real life, this weight can be considered as distance, congestion, traffic stack etc.

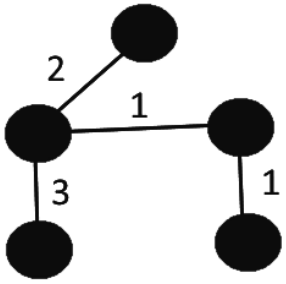
Suppose a connected, undirected graph G = (V,E), and weight W For each edge (u,v) in E, and weight of (u,v) w Ɛ W. Find an (acyclic) subset T of E that connects all vertices in V and sum of weight of all edges will be minimum. The subset T must be acyclic (no cycle), because the sum weight is minimized. Because, T is a tree, it is termed as spanning tree. Finding the tree T is called the minimum-spanning-tree problem.



Graph



Spanning Tree
Cost = 13



Minimum Spanning
Tree, Cost = 7

Design of networks directly involves minimum spanning. Some of the algorithms are traveling salesperson problem, minimum-cost weighted perfect matching , multi terminal minimum cut problem, Cluster Analysis, Handwriting recognition, Image segmentation etc.

MST in a directed graph

Let G = (V, E) be a weighted directed graph with weight matrix W, where W: E → R is weight function. A directed spanning tree (DST) of the given graph is a rooted sub graph T (tree) at r, where

r Ɛ V such that T contains a directed path from r to any other vertex in V AND the undirected version of T is a tree and. if all vertices in G are reachable from vertex r, then the graph contains a directed spanning tree rooted at r.

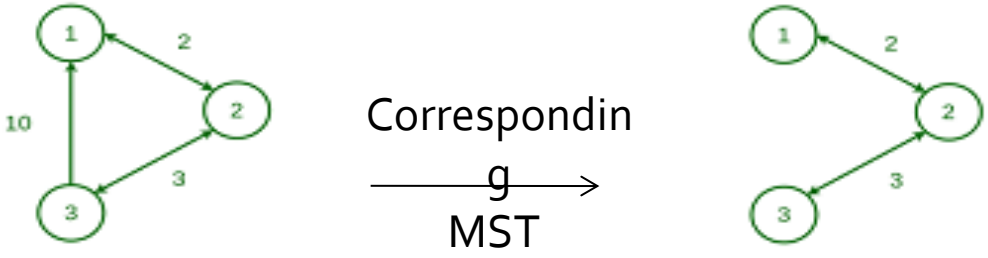
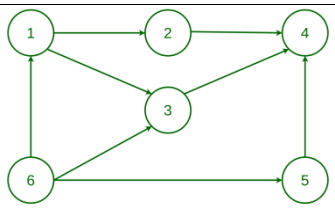
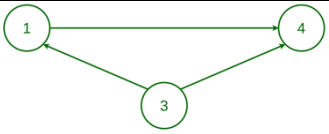


Figure:

It may be possible that every node is not reachable from every other node in a directed graph. But, the Prim's algorithm assumes that all vertices are connected. That is why Prim's algorithm fails.

Whereas in Kruskal's algorithm, we check that if the edges form a cycle with the growing spanning tree in each step. There are cases when there is no cycle between the vertices But Kruskal's algorithm fails to detect the cycles in a directed graph. This Algorithm doesn't consider some edges and assumes that it is forming a cycle. Due to this fact for directed graph Kruskal's Algorithm fails.

An optimum branching or a minimum-cost Arborescence in a directed graph, is taken as counterpart of a minimum spanning tree. The classical algorithm for solving this problem is the Chu-Liu/ Edmonds algorithm. There have been several optimized implementations of this algorithm over the years using better data structures.

	Node 4 is not reachable from any node. But prim's assumes that every node is reachable i.e., All vertices are connected (Prim's Fail in this example)
	The graph contains no cycle. Kruskal assumes that there is a cycle based on the union-find method of the disjoint set. (Kruskal'sfailsin this example)

4.4.3 Multiple-choice question:

1	Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. What is the minimum possible weight of a spanning tree T in this graph such that vertex 0 is a leaf node in the tree T? [GATE CS 2010]
a	7
b	8
c	9
d	10

AN	D

2	In the graph given in question (1) question, what is the minimum possible weight of a path P from vertex 1 to vertex 2 in this graph such that P contains at most 3 edges?
a	7
b	8
c	9
d	10
AN	B

3	An undirected graph G has n nodes. Its adjacency matrix is given by an n × n square matrix whose (i) diagonal elements are 0's and (ii) non-diagonal elements are 1's. Which one of the following is TRUE?
a	Graph G has no minimum spanning tree (MST
b	Graph G has a unique MST of cost n-1
c	Graph G has multiple distinct MSTs, each of cost n-1
d	Graph G has multiple spanning trees of different costs
AN	C

4	Let G be an undirected connected graph with a distinct edge weight. Let emax be the edge with maximum weight and emin the edge with minimum weight. Which of the following statements is false? [GATE CS 2000]
a	Every minimum spanning tree of G must contain emin
b	If Emax is in a minimum spanning tree, then its removal must disconnect G
c	No minimum spanning tree contains emax
d	G has a unique minimum spanning tree
AN	C

5	Consider a weighted complete graph G on the vertex set {v1,v2 ,v} such that the weight of the edge (v,,v) is 2|i-j|. The weight of a minimum spanning tree of G is: {GATE CS 2006}
a	n — 1
b	2n — 2
c	nC2
d	2
AN	B

6	Consider an undirected un weighted graph G. Let a breadth-first traversal of G be done starting from a node r. Let d(r,u) and d(r,v) be the lengths of the shortest paths from r to u and v respectively in G. If u is visited before v during the breadth-first traversal, which of the following statements is correct? [GATE-CS-2001]
a	d(r, u) < d(r, v)
b	d(r,u) > d(r,v)
c	d(r,u) <= (r,v)
d	None of the above
AN	C

7	Let G be connected undirected graph of 100 vertices and 300 edges. The weight of a minimum spanning tree of G is 500. When the weight of each edge of G is increased by five, the weight of a minimum spanning tree becomes _____ [GATE-CS-2015]
a	1000
b	995
c	2000
d	1995
AN	B

8	Let G be a weighted connected undirected graph with distinct positive edge weights. If every edge weight is increased by the same value, then which of the following statements is/are TRUE? P: Minimum spanning tree of G does not change Q: Shortest path between any pair of vertices does not change [GATE-CS-2015]
a	P only
b	Q only
c	Neither P nor Q
d	Both P and Q
AN	A

9	Let G be a complete undirected graph on 4 vertices, having 6 edges with weights being 1, 2, 3, 4, 5, and 6. The maximum possible weight that a minimum weight spanning tree of G can have is. [GATE-CS-2016]
a	6
b	7
c	8
d	9
AN	B

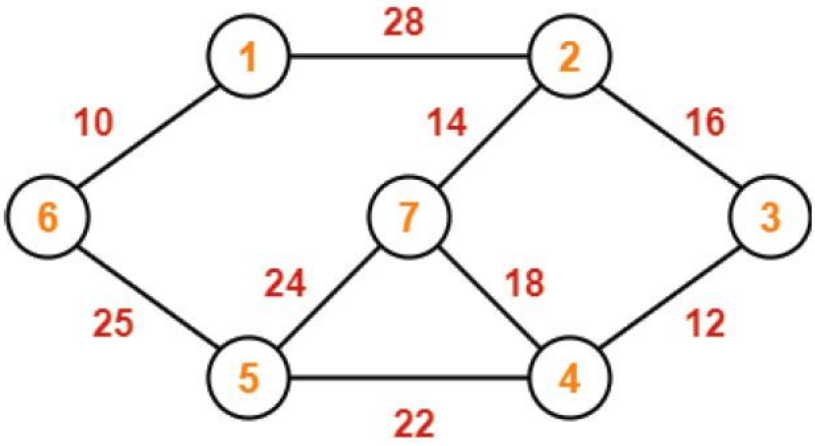
10	G = (V, E) is a simple undirected graph in which each edge has a distinct weight, and e is a particular edge of G. Which of the following statements about the minimum spanning trees (MSTs) of G is/are TRUE I. If e is the lightest edge of some cycle in G, then every MST of G includes e II. If e is the heaviest edge of some cycle in G, then every MST of G excludes e [GATE-CS-2016]
a	I only
b	II only
c	both I and II
d	neither I nor II
AN	B

4.4.4 Kruskal's algorithm for Minimum Spanning Tree

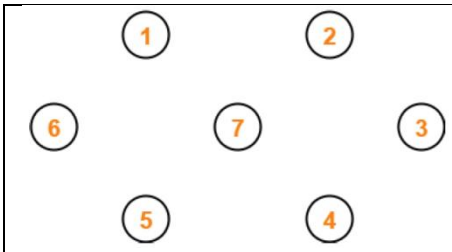
Most of the cable network companies (Landline cable/ TV cable/ Dish TV Network) use Kruskal's algorithm to save a huge amount of cable for connecting different offices/ houses. On our trip to some tourist locations, we plan to visit all the important places and sites, but we are short on time. Then Kruskal's algorithm gives us the way to visit all places in minimum time.

Sort all the edges from low weight to high weight.
Take the edge with the lowest weight and use it to connect the vertices of the graph. If adding an edge creates a cycle, reject that edge and go for the next least weight edge.

Keep adding edges (repeat step 2) until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

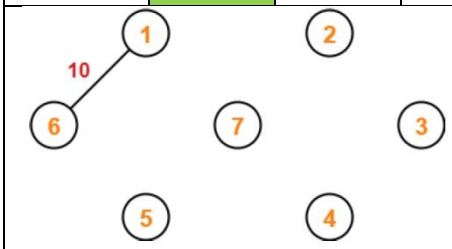


Step - 1									
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



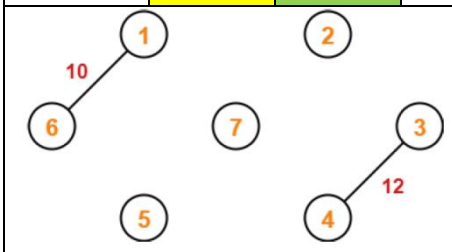
Step - 2

Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



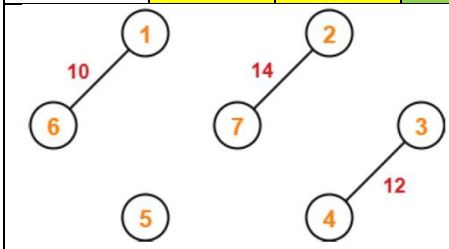
Step - 3

Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



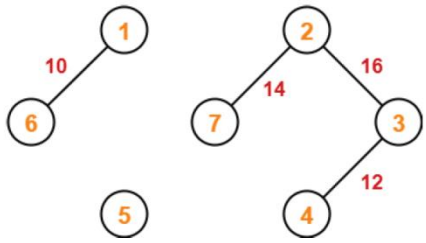
Step - 4

Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



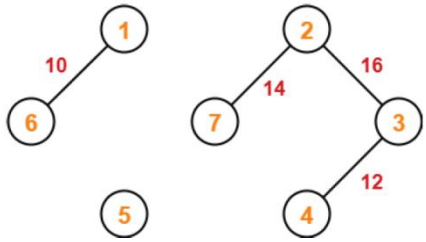
Step - 5

Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



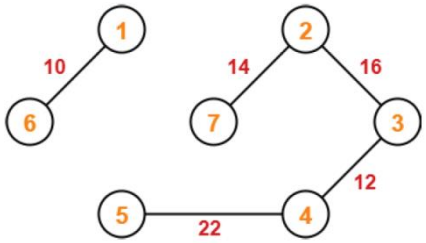
Step - 6

Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



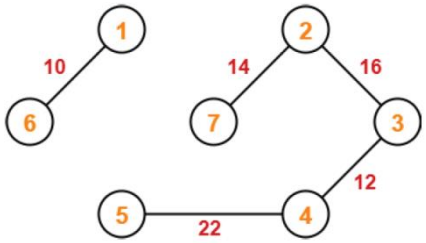
Step - 7

Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



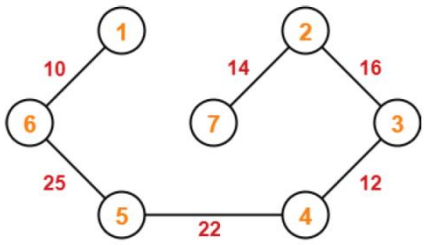
Step - 8

Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



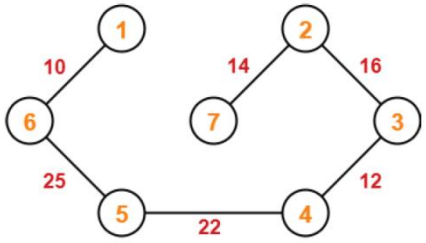
Step - 9

Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



Step - 10

Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)
Weight	10	12	14	16	18	22	24	25	28



Kruskal's algorithm (Step by step execution)

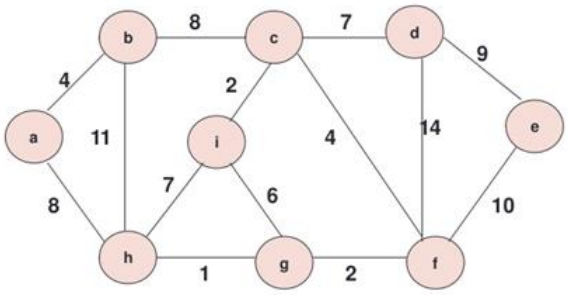
This algorithm finds a safe edge (u, v) of least weight among all the edges that connect any two trees in the growing forest. It is a greedy algorithm. It is like an algorithm to compute connected components. In Kruskal's algorithm, set A is a forest. Set A is always a subset of some minimum spanning tree. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components. It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in a tree of the current forest. It uses Find_Set(u) and UNION (u, v) procedure.

ALGORITHM MSTKruskal(G,W[ ][ ])

```
BEGIN:
E' = ∅
FOR each vertex v ∈ V [G] DO
        Make_Set(v)
Sort the edges of E into non-decreasing order by weight in W
FOR each (u, v) ∈ E DO
IF Find_Set(u) != Find_Set (v) THEN
                Add edge (u,v) in E'
                UNION (u, v)
RETURN E'
END;
```

Lines 1-3 initialize the set A as an empty set and create |V| trees, each containing one vertex. Line 4 sorts the edges in E into non-decreasing order by weight. Lines 5-8 (loop for each edge in E)

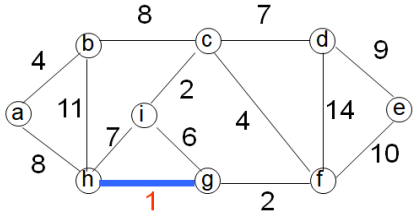
checks that if endpoints u and v belong to the same tree, then the edge (u, v) cannot be added to the forest, and the edge is discarded. Otherwise, the edge (u, v) is added to A in line 7, and the vertices in the two trees are merged in line 8.

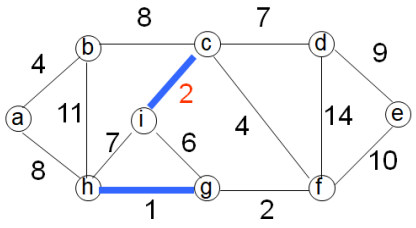


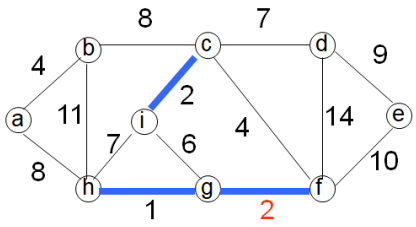
Applying and executing Line No - 1 on a given graph	E' = Ø
Initialize set A as an empty set.	E' = Ø or E'={ }

Applying and executing Line No - 2, 3, on given graph	FOR each vertex v ∈ V [G] DO Make_Set(v)
|V| trees each containing one vertex	{ a } { b } { c } { d } { e } { f } { g } { h } { i }

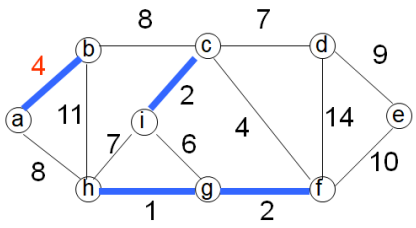
Applying and executing Line No - 4, on given graph	sort the edges of E into non-decreasing order by weight w													
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
Applying and executing Line No - 5 on a given graph	for each edge (u, v) ∈ E into non-decreasing order by weight w for the edge (g, h), u = g and v = h													
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
Applying and executing Line No - 6, on given graph	IF Find_Set(u) != Find_Set (v) THEN g and h belongs to a different set, edge (g, h) is included in A													
Applying and executing Line No - 7 on a given graph	Add edge (u,v) in E' A = { (g, h) }													
Applying and executing Line No - 8 on a given graph	UNION (u, v) { a } { b } { c } { d } { e } { f } { g, h } { i }													



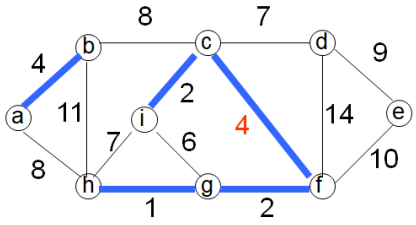
	for the edge (c, i), u = c and v = i													
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
c and i belongs to different set, edge (c, i) is included in A, union operation is performed														
A = { (g, h), (c, i) }														
{ a } { b } { c, i } { d } { e } { f } { g, h }														
														

	for the edge (f, g), u = f and v = g													
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
f and g belongs to different set, edge (f, g) is included in A, union operation is performed														
A = { (g, h), (c, i) , (f, g) }														
{ a } { b } { c, i } { d } { e } { f, g, h }														
														

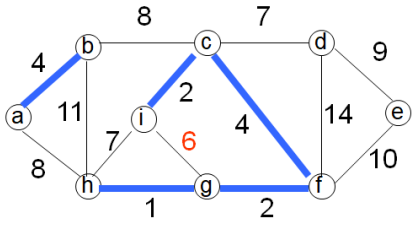
	for the edge (a, b), u = a and v = b												
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
a and b belongs to different set, edge (a, b) is included in A, union operation is performed														
A = { (g, h), (c, i), (f, g), (a, b) }														
{ a, b }{ c, i } { d } { e } { f, g, h }														



	for the edge (c, f), u = c and v = f												
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
c and f belongs to different set, edge (c, f) is included in A, union operation is performed														
A = { (g, h), (c, i), (f, g), (a, b), (c, f) }														
{ a, b } { d } { e } { c, f, g, h, i }														



	for the edge (g, i), u = g and v = i												
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
g and i belongs to Same set, so the given edge is not safe. It is not included in set A, no union as well														
A = { (g, h), (c, i), (f, g), (a, b), (c, f) }														
{ a, b } { d } { e } { c, f, g, h, i }														



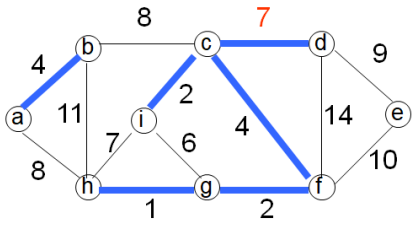
	for the edge (c, d), u = c and v = d												
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)

Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14

c and d belongs to different set, edge (c, d) is included in A, union operation is performed

A = { (g, h), (c, i), (f, g), (a, b), (c, f), (c, d) }

{ a, b } { e } { c, d, f, g, h, i }

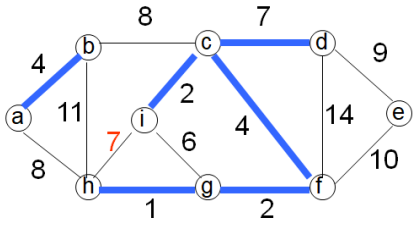


	for the edge (h, i), u = h and v = i													
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14

h and i belongs to Same set, so the given edge is not safe. It is not included in set A, no union as well

A = { (g, h), (c, i), (f, g), (a, b), (c, f), (c, d) }

{ a, b } { e } { c, d, f, g, h, i }

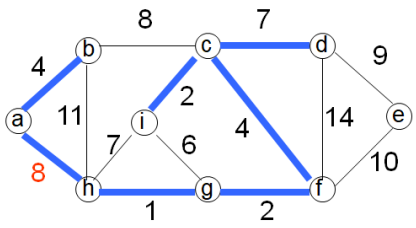


	for the edge (a, h), u = a and v = h													
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14

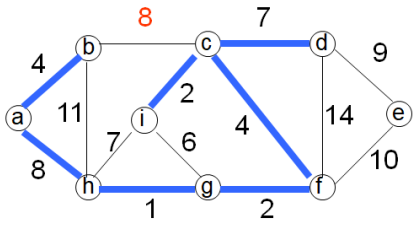
a and h belongs to different set, edge (a, h) is included in A, union operation is performed

A = { (g, h), (c, i), (f, g), (a, b), (c, f), (c, d), (a, h) }

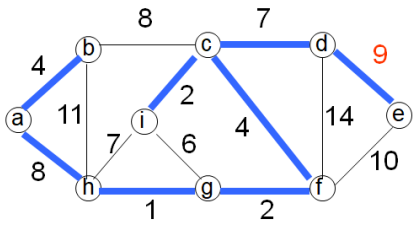
{ e } { a, b, c, d, f, g, h, i }



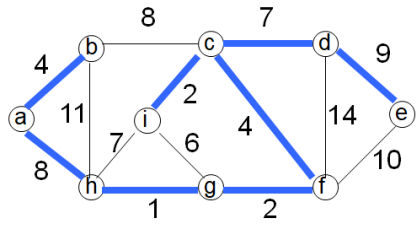
	for the edge (b, c), u = b and v = c													
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
b and c belongs to Same set, so the given edge is not safe. It is not included in set A, no union as well														
A = { (g, h), (c, i), (f, g), (a, b), (c, f), (c, d), (a, h) }														
{ e } { a, b, c, d, f, g, h, i }														



	for the edge (d, e), u = d and v = e													
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
d and e belongs to different set, edge (a, h) is included in A, union operation is performed														
A = { (g, h), (c, i), (f, g), (a, b), (c, f), (c, d), (a, h), (d, e) }														
{ a, b, c, d, e, f, g, h, i }														

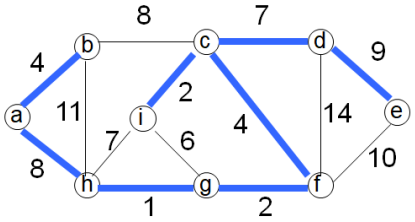


	for the all edge remaining {(e, f), (b, h), (d, f) },													
Edge	(g,h)	(c,i)	(f,g)	(a,b)	(c,f)	(g,i)	(c,d)	(h,i)	(a,h)	(b,c)	(d,e)	(e,f)	(b,h)	(d,f)
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14
All given edge are not safe. They are not included in set A, also no union operation.														
A = { (g, h), (c, i), (f, g), (a, b), (c, f), (c, d), (a, h), (d, e) }														

{ a, b, c, d, e, f, g, h, i }


Output:

The obtained Set A = { (g, h), (c, i), (f, g), (a, b), (c, f), (c, d), (a, h), (d, e) }.



The total weight of MST = 37

Kruskal's algorithm (algorithm analysis)

The running time of Kruskal's algorithm for a graph G = (V, E) depends on the implementation of the disjoint-set data structure. We shall assume the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics since it is the asymptotically fastest implementation known. The running time of the combined union-by-rank and path compression heuristic is O(m α (n)) from disjoint-set operations on n elements.

Initializing the set A in line 1 takes O(1) time.
The cost of the |V| MAKE-SET operations in lines 2-3 is O(V).
The time to sort the edges in line 4 is O(E log E).

The for loop of lines 5-8 performs O(E) FIND-SET and UNION operations on the disjoint-set forest. Along with the |V| MAKE-SET operations, these take a total of O((V + E) α(V)) time, where α is the very slowly growing function.

Because G is assumed to be connected, we have |E| ≥ |V| - 1, and so the disjoint-set operations take O(E α(V)) time. Moreover, since α(|V|) = O(log V) = O(LG E), the total running time of Kruskal's algorithm is O(E log E). Observing that |E| < |V|2, we have LG |E| = O(log V), and so we can restate the running time of
Kruskal's algorithm as O(E lg V).

Another alternative:

Checking of vertices (whether connected or not) could be done using DFS, which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of where is the number of vertices is the number of edges. Time Complexity of DFS is also O(V+E), where V is vertices and E is edges. So the best solution is "Disjoint Sets."

The edges are maintained as min-heap. The next edge can be obtained in O(logE) time if the graph has E edges. Reconstruction of heap takes O(E) time. So, Kruskal's Algorithm takes O(ElogE) time. The value of E can be at most O(V2). So, O(logV) and O(logE) are same. The worst-case time complexity of Kruskal's Algorithm = O(ElogV) or O(ElogE). If the edges are already sorted, then there is no need to construct a min-heap. So, deletion from min-heap time is saved. In this case, the time complexity of Kruskal's Algorithm = O(E + V)

4.4.5 Multiple-choice question

1	Kruskal's algorithm is used to
a	find minimum spanning tree
b	find the single-source shortest path
c	find all pair shortest path algorithm
d	traverse the graph
AN	A

2	Kruskal's algorithm is a _____
a	divide and conquer algorithm
b	dynamic programming algorithm
c	greedy algorithm
d	approximation algorithm
AN	C

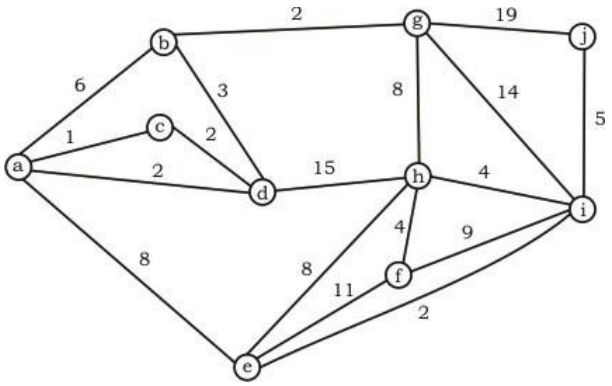
3	What is the time complexity of Kruskal's algorithm?
a	O(log V)
b	O(E log V)
c	O(E2)
d	O(V log E)
AN	B

4	Which of the following is true?
a	Prim's algorithm can also be used for disconnected graphs
b	Kruskal's algorithm can also run on the disconnected graphs
c	Prim's algorithm is simpler than Kruskal's algorithm
d	In Kruskal's sort edges are added to MST in decreasing order of their weights
AN	B

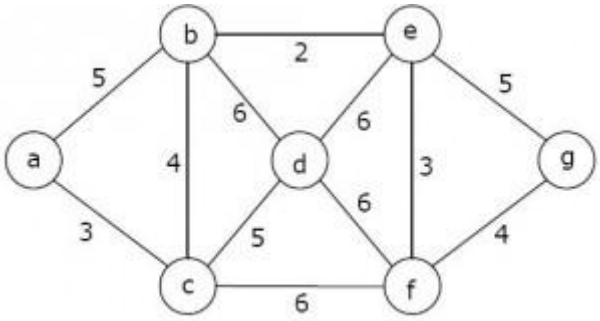
5	Which of the following is false about Kruskal's algorithm?
a	It is a greedy algorithm
b	It constructs MST by selecting edges in increasing order of their weights
c	It can accept cycles in the MST
d	It uses a union-find data structure
AN	C

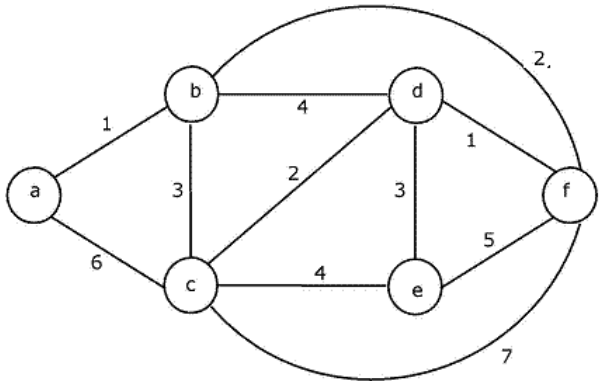
6	If Kruskal's algorithm is used for finding a minimum spanning tree of a weighted graph G with n vertices and m edges and edge weights are already given in a sorted list, then, What will be the time complexity to compute the minimum cost spanning tree given that union and find operations take amortized O(1)
a	O(m logn)
b	O(n)
c	O(m)
d	O(n log)
AN	c

7	Consider the following statements. S1. Kruskal's algorithm might produce a non-minimal spanning tree. S2. Kruskal's algorithm can efficiently be implemented using the disjoint-set data structure.
a	S1 is true but S2 is false
b	Both S1 and S2 are false
c	Both S1 and S2 are true
d	S2 is true but S1 is false
AN	D

8	What is the weight of a minimum spanning tree of the following graph? [GATE-CS-2003] 
a	29
b	31
c	38

d	41
AN	B

9	Consider the following graph: Which one of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm? [GATE-CS-2009] 
A	(b,e)(e,f)(a,c)(b,c)(f,g)(c,d)
B	(b,e)(e,f)(a,c)(f,g)(b,c)(c,d)
C	(b,e)(a,c)(e,f)(b,c)(f,g)(c,d)
D	(b,e)(e,f)(b,c)(a,c)(f,g)(c,d)
AN	D

10	Consider the following graph. Which one of the following cannot be the sequence of edges added, in that order, to a minimum spanning tree using Kruskal's algorithm? [GATE-CS-2006] 
A	(a—b),(d—f),(b—f),(d—c),(d—e)
B	(a—b),(d—f),(d—c),(b—f),(d—e)
C	(d—f),(a—b),(d—c),(b—f),(d—e)
D	(d—f),(a—b),(b—f),(d—e),(d—c)
AN	D

4.4.6 Prim's Algorithm

Prim's algorithm is used to find the minimum spanning tree. It takes a graph as input and gives a minimum spanning tree as output. In prims, the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Prims follow the greedy strategy since the tree is

augmented at each step with an edge that contributes the minimum amount possible to the tree's weight.

Procedure

Step 1: first initialize the starting node of the graph.

Step 2: initialize the priority queue Q and assign key-value ∞ to each vertex except root or source node whose key value is 0.

Step 3: check all adjacent nodes belongs to Q of the current node and compare the new cost with the existing cost. Select minimum cost edge among all edges connected with adjacent nodes. Remove current node from Q.

Step 4: make the current node to the minimum cost adjacent node.

Step 5: Repeat steps 3 and 4.

ALGORITHM MST Prims (G, w[ ][ ], r) //Graph G, w is weight matrix and r is the starting node

BEGIN:

FOR each vertex u ϵ V[G] DO // u is current vertex and V[G] is all vertices belongs to graph G.

key[u] = ∞ // initially key value of all vertices are infinite

∏[u] = NIL // ∏[u] means parent of u and root node have no parent so value is NIL

Key[r]=0 // value of starting node is 0

Initialize(PQ) // PQ is priority queue

FOR all u ϵ V[G] DO

EnQueue(PQ, u)

WHILE !Empty(PQ) DO // all nodes should be traversed

u = DeQueue(PQ) // Return the vertex with minimum key value

FOR each v ϵ adj[u] DO // v is adjacent node of u

if v ϵ PQ and w(u,v) < key[v] THEN // if v present in Priority Queue

∏[v] = u // parent of v is u

key [v] = w(u,v) // update key value of v

END;

Complexity:

We can divide the algorithm into three parts

Part 1 Extraction of the vertex from the priority queue process takes O(log V) time.

Part 2 Decreasing the key of adjacent vertices takes O(log V) time.

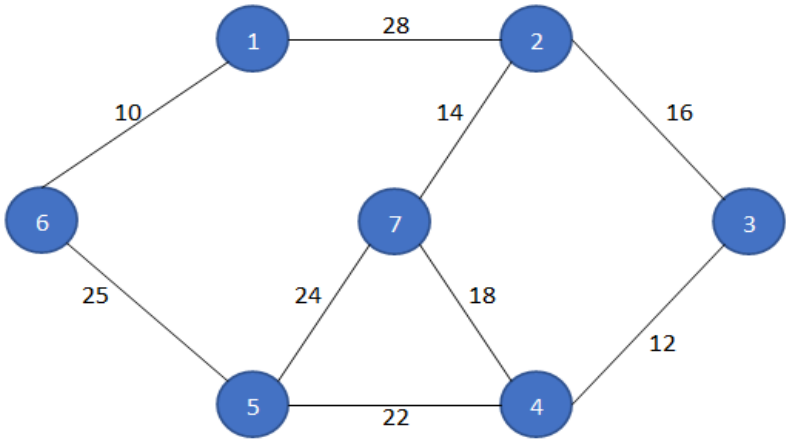
Part 3 updating the key values takes constant time.

So the total running time complexity = O(V log V + E log V)

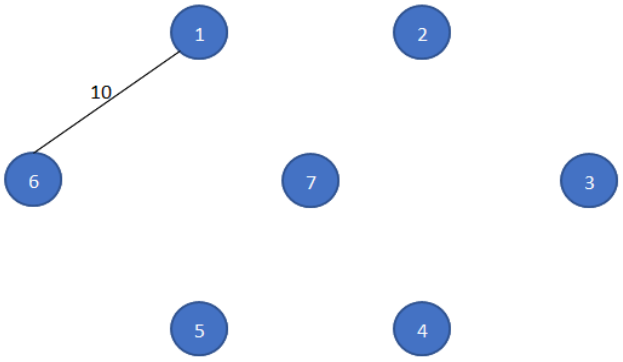
= O(E log V) E is always greater than V.

Example 1:

Find the minimum spanning tree for the given graph.

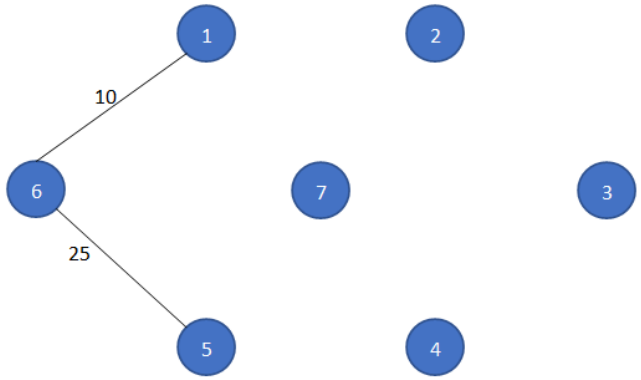


1	2	3	4	5	6	7	extracted vertex	adjacent vertex	edges	selected vertex
0	∞	∞	∞	∞	∞	∞	1	2	1,2 = 28	6
								6	1,6 = 10	



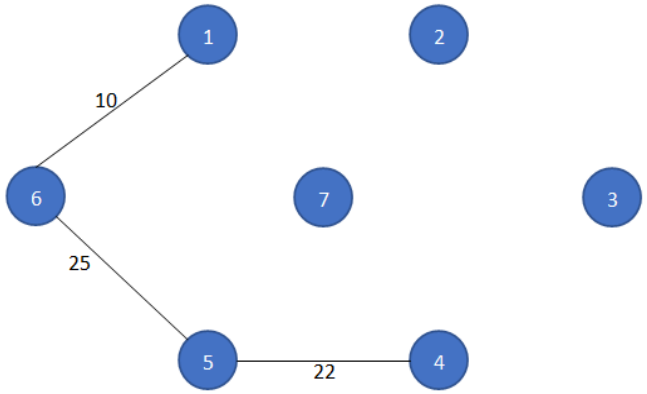
1	2	3	4	5	6	7	extracted vertex	adjacent vertex	edges	selected vertex
0	∞	∞	∞	∞	∞	∞	1	2	1,2 = 28	6
								6	1,6 = 10	
X	28	∞	∞	∞	10	∞	6	1	NA	5
								5	6,5 = 25	

1 is not in Priority Queue



1	2	3	4	5	6	7	extracted vertex	adjacent vertex	edges	selected vertex
0	∞	∞	∞	∞	∞	∞	1	2	1,2 = 28	6
								6	1,6 = 10	
X	28	∞	∞	∞	10	∞	6	1	NA	5
								5	6,5 = 25	
X	28	∞	∞	25	X	∞	5	4	5,4 = 22	4
								7	5,7 = 24	

1 is not in Priority Queue

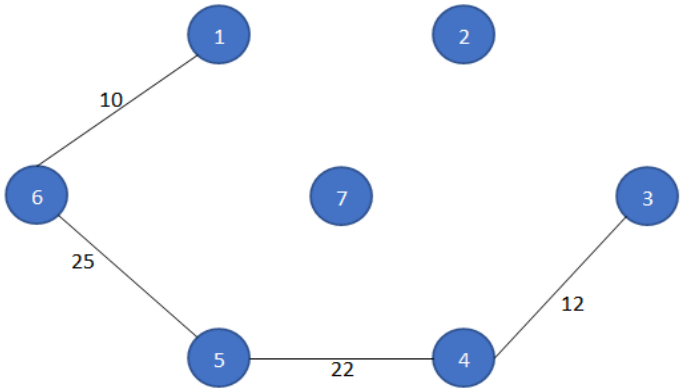


1	2	3	4	5	6	7	extracted vertex	adjacent vertex	edges	selected vertex
0	∞	∞	∞	∞	∞	∞	1	2	1,2 = 28	6
								6	1,6 = 10	
X	28	∞	∞	∞	10	∞	6	1	NA	5
								5	6,5 = 25	
X	28	∞	∞	25	X	∞	5	4	5,4 = 22	4
								7	5,7 = 24	
X	28	∞	22	X	X	24	4	3	4,3 = 12	3

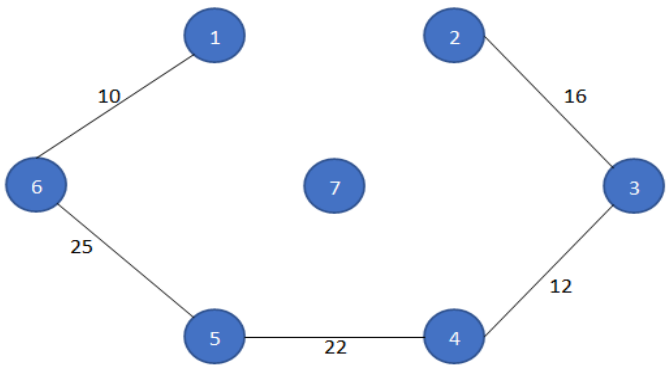
1 is not in Priority Queue

								5	NA	
								7	4,7 = 18	

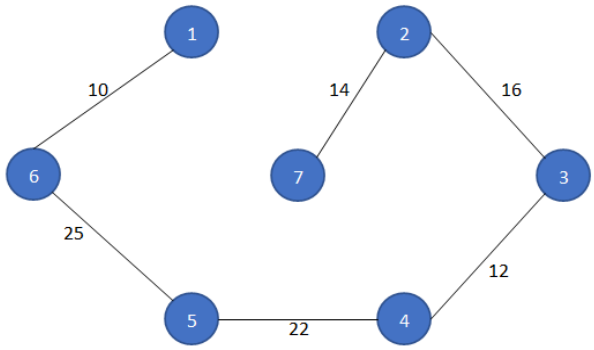
5 is not in Priority Queue



1	2	3	4	5	6	7	extracted vertex	adjacent vertex	edges	selected vertex	
0	∞	∞	∞	∞	∞	∞	1	2	1,2=28	6	
								6	1,6=10		
X	28	∞	∞	∞	10	∞	6	1	NA	5	1 is not in Priority Queue
								5	6,5=25		
X	28	∞	∞	25	X	∞	5	4	5,4=22	4	
								7	5,7=24		
X	28	∞	22	X	X	24	4	3	4,3=12	3	
								5	NA		5 is not in Priority Queue
								7	4,7=18		
X	28	12	X	X	X	18	3	2	3,2=16	2	
								4	NA		4 is not in Priority Queue



1	2	3	4	5	6	7	extracted vertex	adjacent vertex	edges	selected vertex	
0	∞	∞	∞	∞	∞	∞	1	2	1,2=28	6	
								6	1,6=10		
X	28	∞	∞	∞	10	∞	6	1	NA	5	1 is not in Priority Queue
								5	6,5=25		
X	28	∞	∞	25	X	∞	5	4	5,4=22	4	
								7	5,7=24		
X	28	∞	22	X	X	24	4	3	4,3=12	3	
								5	NA		5 is not in Priority Queue
								7	4,7=18		
X	28	12	X	X	X	18	3	2	3,2=16	2	
								4	NA		4 is not in Priority Queue
X	16	X	X	X	X	18	2	1	NA	7	1 is not in Priority Queue
								3	NA		3 is not in Priority Queue
								7	2,7=14		



1	2	3	4	5	6	7	extracted vertex	adjacent vertex	edges	selected vertex	
0	∞	∞	∞	∞	∞	∞	1	2	1,2=28	6	
								6	1,6=10		
X	28	∞	∞	∞	10	∞	6	1	NA	5	1 is not in Priority Queue
								5	6,5=25		
X	28	∞	∞	25	X	∞	5	4	5,4=22	4	
								7	5,7=24		
X	28	∞	22	X	X	24	4	3	4,3=12	3	
								5	NA		5 is not in Priority Queue
								7	4,7=18		
X	28	12	X	X	X	18	3	2	3,2=16	2	
								4	NA		4 is not in Priority Queue
X	16	X	X	X	X	18	2	1	NA	7	1 is not in Priority Queue

								3	NA		3 is not in Priority Queue
								7	2,7=14		
X	X	X	X	X	X	18	7	2	NA	7	2 is not in Priority Queue
								4	NA		4 is not in Priority Queue
								5	NA		5 is not in Priority Queue

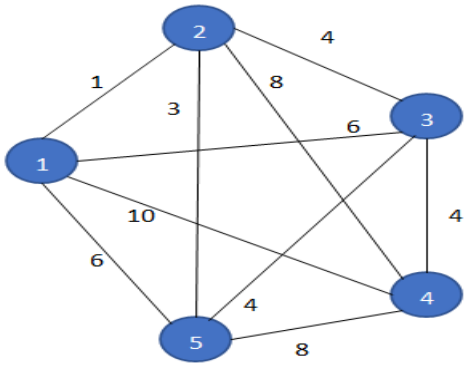
No updation is required because Q = Ø.

Total cost of the given spanning tree is = 10 + 25 + 22 + 12 + 16 + 14

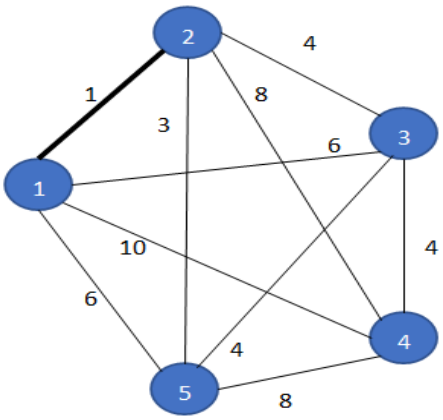
= 99

Example 2:

Find the minimum spanning tree for the given graph.

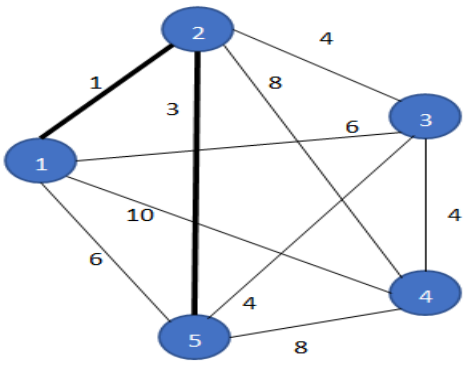


1	2	3	4	5	extracted vertex	adjacent vertex	edges	selected vertex
0	∞	∞	∞	∞	1	2	1,2=1	2
						3	1,3=6	
						4	1,4=10	
						5	1,5=6	

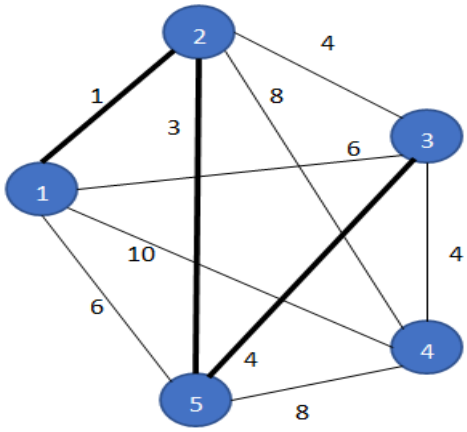


1	2	3	4	5	extracted vertex	adjacent vertex	edges	selected vertex
0	∞	∞	∞	∞	1	2	1,2=1	2
						3	1,3=6	

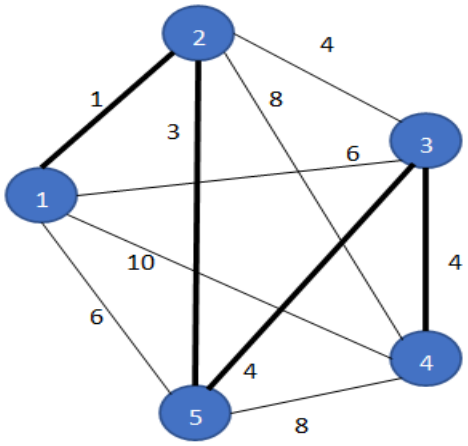
						4	1,4=10		
						5	1,5=6		
X	1	6	10	6	2	1	NA	5	1 is not in Priority Queue
						3	2,3=4		
						4	2,4=8		
						5	2,5=3		



1	2	3	4	5	extracted vertex	adjacent vertex	edges	selected vertex	
0	∞	∞	∞	∞	1	2	1,2=1	2	
						3	1,3=6		
						4	1,4=10		
						5	1,5=6		
X	1	6	10	6	2	1	NA	5	1 is not in Priority Queue
						3	2,3=4		
						4	2,4=8		
						5	2,5=3		
X	X	4	8	3	5	1	NA	3	1 is not in Priority Queue
						2	NA		2 is not in Priority Queue
						3	5,3=4		
						4	5,4=8		



1	2	3	4	5	extracted vertex	adjacent vertex	edges	selected vertex	
0	∞	∞	∞	∞	1	2	1,2=1	2	
						3	1,3=6		
						4	1,4=10		
						5	1,5=6		
X	1	6	10	6	2	1	NA	5	1 is not in Priority Queue
						3	2,3=4		
						4	2,4=8		
						5	2,5=3		
X	X	4	8	3	5	1	NA	3	1 is not in Priority Queue
						2	NA		2 is not in Priority Queue
						3	5,3=4		
						4	5,4=8		
X	X	4	8	X	3	1	NA	4	1 is not in Priority Queue
						2	NA		2 is not in Priority Queue
						4	3,4=4		
						5	NA		4 is not in Priority Queue



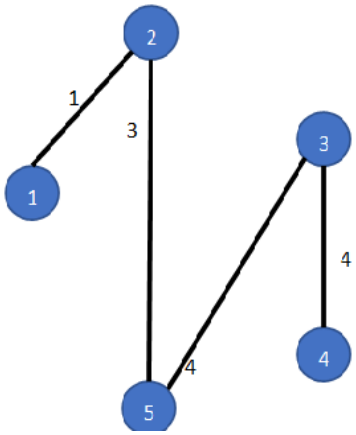
1	2	3	4	5	extracted vertex	adjacent vertex	edges	selected vertex	
0	∞	∞	∞	∞	1	2	1,2=1	2	
						3	1,3=6		
						4	1,4=10		
						5	1,5=6		
X	1	6	10	6	2	1	NA	5	1 is not in Priority Queue
						3	2,3=4		
						4	2,4=8		
						5	2,5=3		
X	X	4	8	3	5	1	NA	3	1 is not in Priority Queue
						2	NA		2 is not in Priority Queue
						3	5,3=4		
						4	5,4=8		
X	X	4	8	X	3	1	NA	4	1 is not in Priority Queue
						2	NA		2 is not in Priority Queue
						4	3,4=4		
						5	NA		4 is not in Priority Queue
X	X	X	4	X	4	1	NA		1 is not in Priority Queue
						2	NA		2 is not in Priority Queue

			3	NA	
			5	NA	

3 is not in Priority Queue

5 is not in Priority Queue

A possible minimum spanning tree is

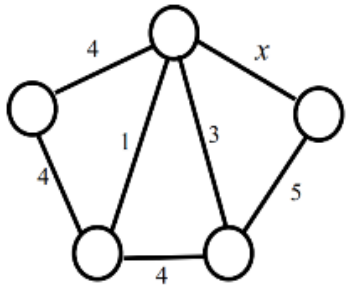


Total cost= 1 + 3 + 4 + 4

= 12

4.4.7 Multiple Choice questions:

	GATE 2019 Marks 2
1	Let *G* be any connected, weighted, undirected graph. I. *G* has a unique minimum spanning tree if no two edges of *G* have the same weight. II. *G* has a unique minimum spanning tree if, for every cut of *G*, there is a unique minimum-weight edge crossing the cut. Which of the above two statements is/are TRUE?
A	I only
B	II only
C	Both I and II
D	Neither I nor II
AN	Both I and II

	GATE 2018 Marks 2
2	Consider the following undirected graph G: 

	Choose a value for x that will maximize the number of minimum weight spanning trees (MSTs) of G. The number of MWSTs of G for this value of *x* is _____.
AN	4

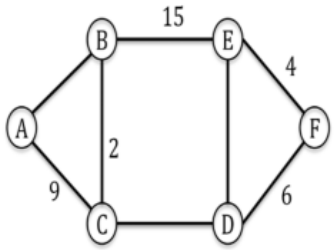
	GATE 2017 Marks 2
3	Let *G= (V,E)* be any connected undirected edge-weighted graph. The weight of the edges in *E* is positive and distinct. Consider the following statements: (I) Minimum Spanning Tree of *G* is always unique. (II) the shortest path between any two vertices of *G* is always unique. Which of the above statements is/are necessarily true?
A	I only
B	II only
C	Both I and II
D	Neither I nor II
AN	I only

	GATE 2016 Marks 1
4	Let *G* be a weighted connected undirected graph with distinct positive edge weights. If every edge weight is increased by the same value, then which of the following statements is/are TRUE? P: Minimum spanning tree of *G* does not change Q: Shortest path between any pair of vertices does not change
A	P only
B	Q only
C	Both P and Q
D	Neither P nor Q
AN	P only

	GATE 2016 Marks 2
5	Let *G* be a complete undirected graph on 4 vertices, having 6 edges with weights being 1, 2, 3, 4, 5, and 6. The maximum possible weight that a minimum weight spanning tree of *G* can have is _____.
AN	7.0

	GATE 2016 Marks 2
6	*G = (V,E)* is a simple undirected graph in which each edge has a distinct weight, and *e* is a particular edge of *G*. Which of the following statements about the minimum spanning trees (MSTs) of *G* is/are TRUE? I. If *e* is the lightest edge of some cycle in *G*, then every MST of *G* includes *e* II. If *e* is the heaviest edge of some cycle in *G*, then every MST of *G* excludes *e*

A	I only
B	II only
C	Both I and II
D	Neither I nor II
AN	II only

	GATE 2015 Marks 2
7	The graph shown below has 8 edges with distinct integer edge weights. The minimum spanning tree (MST) is of weight 36 and contains the edges: {(A, C), (B, C), (B, E), (E, F), (D, F)}. The edge weights of only those edges which are in the MST are given in the figure shown below. The minimum possible sum of weights of all 8 edges of this graph is _____. 
AN	69

	GATE 2015 Marks 2
8	Let *G* be a connected undirected graph of 100 vertices and 300 edges. The weight of a minimum spanning tree of *G* is 500. When the weight of each edge of *G* is increased by five, the weight of a minimum spanning tree becomes_____.
AN	995

Animation Link: https://www.youtube.com/watch?v=g1EyWlt3TbM

4.4.8 Boruvka's Algorithm for Minimum Cost Spanning Tree

Boruvka's algorithm is a hybrid of Kruskal's and Prim's algorithm. Borůvka's algorithm is a greedy algorithm for finding a minimum spanning tree in a graph or a minimum spanning forest in the case of a graph that is not connected.

The algorithm begins by finding the minimum-weight edge incident to each vertex of the graph and adding all of those edges to the forest. Then, it repeats a similar process of finding the minimum-weight edge from each tree constructed so far to a different tree and adding all of those edges to the forest.

Each repetition of this process reduces the number of trees, within each connected component of the graph, to at most half of this former value, so after logarithmically many repetitions, the process finishes. When it does, the set of edges it has added forms the minimum spanning forest. This algorithm is frequently called Collin's algorithm and is used in parallel computing.

Algorithm:

Step 1: Initialize all nodes as individual components.

Step 2: Initialize the minimum spanning tree S as an empty set that will contain the solution.

Step 3: If there is more than one component:

Find the minimum-weight edge that connects this component to any other component.

If this edge isn't in the minimum spanning tree S, we add it.

Step 4: If there is only one component left, we have reached the end of the tree.

ALGORITHM MSTBoruvka(F,W[ ][ ])

Input: A graph G whose edges have distinct weights.

Output: F is the minimum spanning forest of G.

BEGIN:

Initialize a forest F to be a set of one-vertex trees, one for each vertex of the graph.

Completed = false

WHILE not Completed DO

Find the connected components of F and label each vertex of G by its component

Initialize the minimum weight edge for each component to "None"

FOR each edge (u, v) of G DO

If u and v have different component labels THEN

If u, v is cheaper than the minimum weight edge for the component of u THEN

Set u, v as the minimum weight edge for the component of u

If u, v is cheaper than the minimum weight edge for the component of v THEN

Set u, v as the minimum weight edge for the component of v

Completed = true

FOR each component whose minimum weight edge is not "None" DO

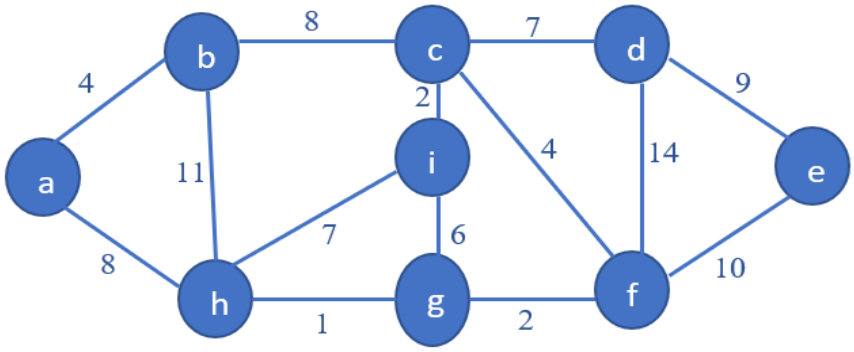
Add its cheapest edge to Forest F

Completed = false

END;

Time Complexity: Each execution of the while loop lowers the number of trees by at least a factor of 2, which means we run the while loop O (log V) times. This, together with the $O(E)$ time for covering all the edges to form a connected component, gives a total running time of $O(E \log V)$.Time Complexity of Boruvka's algorithm is O (E log V) which is the same as Kruskal's and Prim's algorithms.

Example 1: Given a Graph G= (V, E). Apply Boruvka's Algorithm for computing the minimum spanning tree.



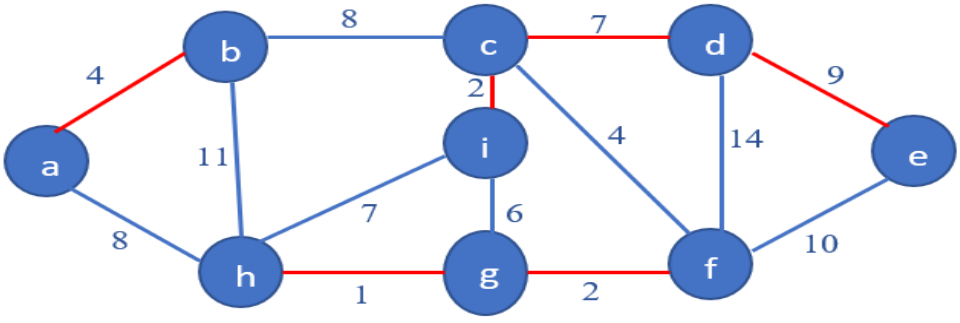
Solution 1:

Step 1: Initially Set S is empty. Every vertex is a single component.

{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

Step 2: For every vertex component, find the minimum weight edge that connects to another vertex.

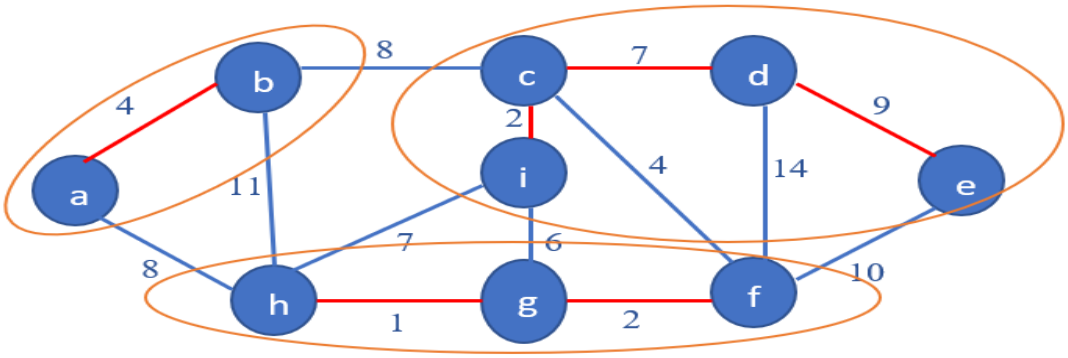
Vertex Component	Minimum Weight Edge that connects to another vertex component	Weight of edge
{a}	a-b	4
{b}	a-b	4
{c}	c-i	2
{d}	c-d	7
{e}	d-e	9
{f}	f-g	2
{g}	g-h	1
{h}	g-h	1
{i}	c-i	8



Minimum weight edges are highlighted with red color.

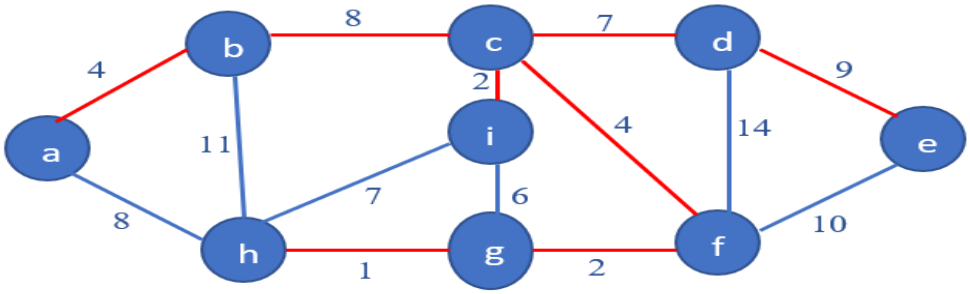
Step 3: Red edges in the graph represent edges that bind together its closest vertex component

Now we have three vertex components: {a, b}, {c, d, e, i}, {f, g, h}



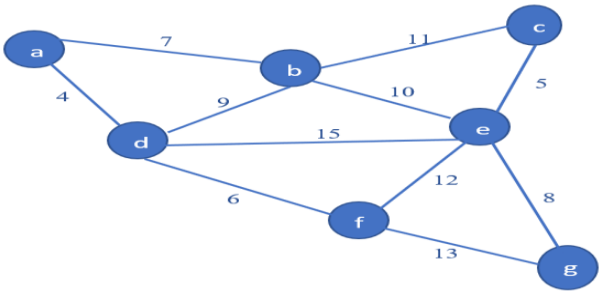
Step 4:Now Repeat the step for every vertex component, i.e., to find the minimum weight edge that connects to the vertex component.

Vertex Component	Minimum Weight Edge that connects to another vertex component	Weight of edge
{a, b}	b-c or a-h	8
{c, d, e, i}	c-f	4
{f, g, h}	c-f	4



Step 5: Now it is one vertex component that contains all the edges: {a, b, c, d, e, f, g, h, i}
So, will STOP and return the MST.

Example 2: Apply Boruvka's Algorithm to the given graph to find the minimum spanning tree.

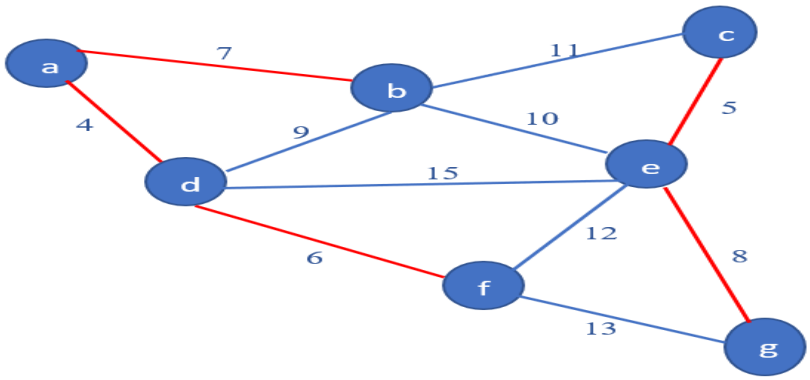


Step 1: Initially Set S is empty. Every vertex is a single component. {a} , {b}, {c}, {d}, {e}, {f}, {g}

Step 2: For every vertex component, find the minimum weight edge that connects to another vertex.

Vertex Component	Minimum Weight Edge that connects to another vertex component	Weight of edge
{a}	a-d	4
{b}	a-b	7

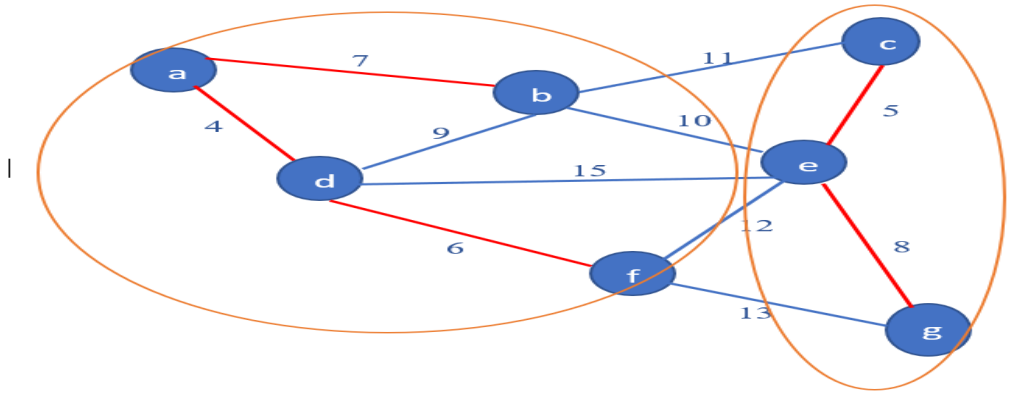
{c}		c-e	5
{d}	a-d		4
{e}	c-e		5
{f}	d-f		6
{g}	e-g		8



Minimum weight edges are highlighted with red color.

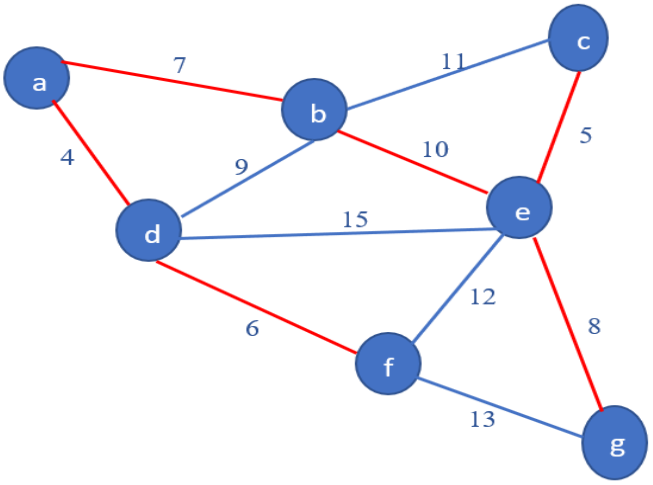
Step 3: These red edges in the graph represent the edges that bind together its closest vertex component.

Now we have two vertex components: {a, b, d, f} and {c, e, g}



Step 4: Now Repeat the step for every vertex component, i.e., to find the minimum weight edge that connects to the vertex component.

Vertex Component	Minimum Weight Edge that connects to another vertex component	Weight of edge
{a, b, d, f}	b-e	10
{c, e, g}	b-e	10



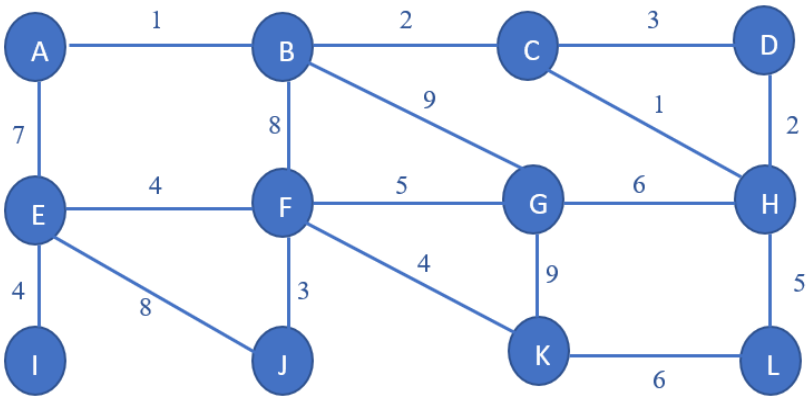
Edge bd cannot be selected because both vertices b and d are in the same component.

Step 5: Now it is one vertex component that contains all the edges: {a, b, c, d, e, f, g,}

So, It will STOP and return the MST.

Example 3:

Apply Boruvka's Algorithm to the given graph to find the minimum spanning tree.



Solution:

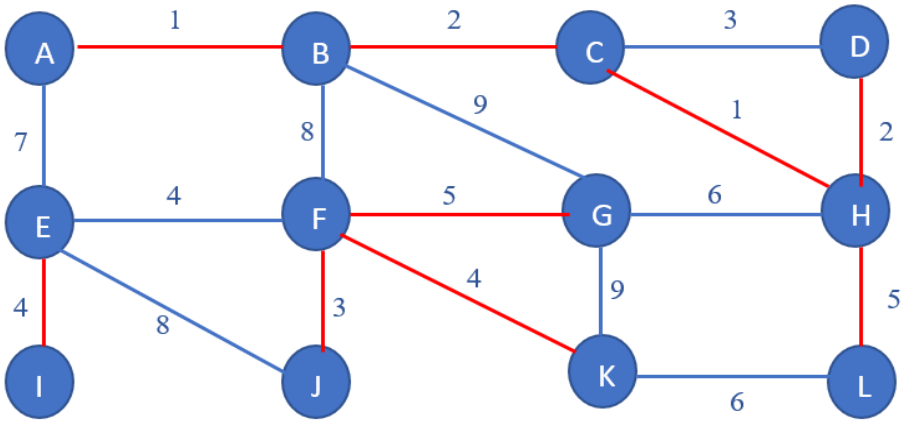
Step 1: For this graph, we have list of components are : A,B,C,D,E,F,G,H,I,J,K,L.

Step 2: Highlight the minimum weight outgoing edge for each node in your list.

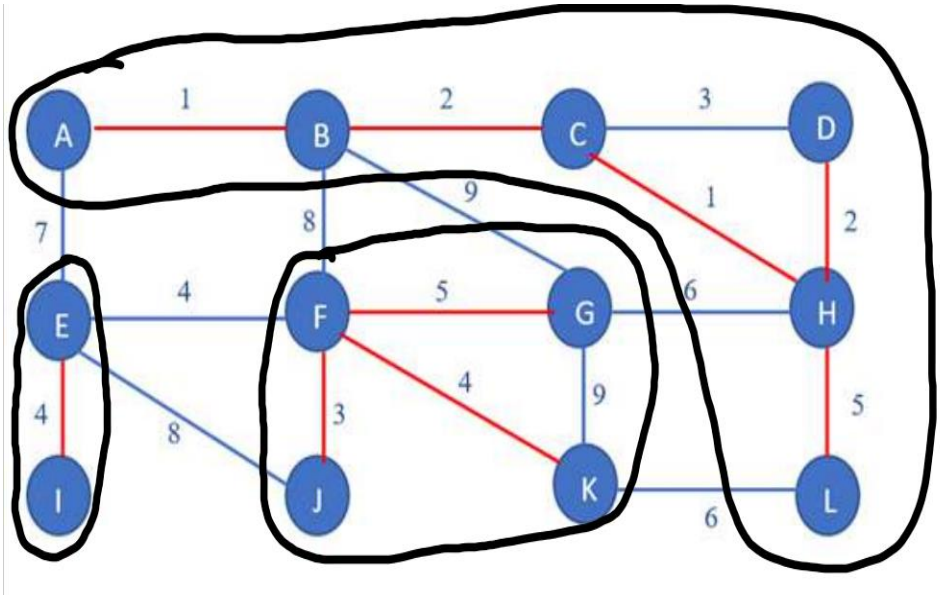
At this stage, only highlight the minimum weight edge for each node.

If there is a tie (node E has two edges with a weight of 4), assign a lower weight to one of the edges. This is arbitrary but must stay consistent throughout the process. For this example, I'll make the edge to the left the lowest weight.

Always highlight the minimum weight edge, *even if it has been highlighted before*. Do not choose the edge with the next lowest weight!



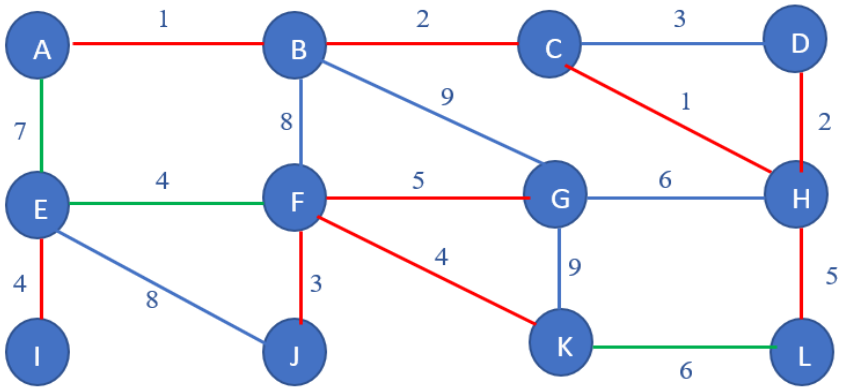
Step 3: Highlight the separate tree clusters. These are the sets of connected nodes, which we'll call *components*.



Step 4: Repeat the algorithm for each component. This time, for each node, choose the minimum edge outside of the component. Here we have 3 components (ABCDHL) (EI) (FGJK).

For example, ABCDHL is one component. For node A, the cheapest edge outside the component is node 7 (because node 1 is connected inside the component). Now, after connecting the ABCDHL component with the EI component with the minimum weight edge. After this connection, we are left with only two components one is ABCDHLEI and FGJK. Now we will simply connect both components with the minimum weight edge from vertex E to Vertex f.

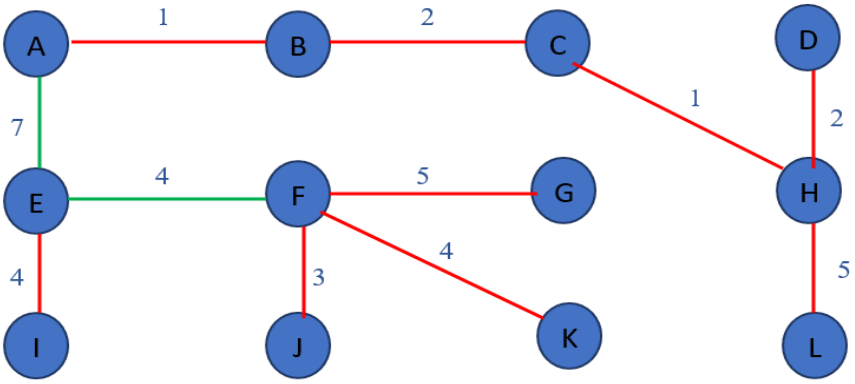
Make sure you are choosing the minimum weight edge *outside* of the component. I am highlighting green color.



Step 5: After this connection, we are left with only two components one is ABCDHLEI and FGJK. Now we will simply connect both components with the minimum weight edge from vertex E to Vertex f.

If your nodes are all connected in a single component, you're done. If not, repeat step 4.

Simply remove Edge (E, H) (B, G) (B, F) (G, H) and (K, L). Hence, it does not form any cycle.



Animations of Boruvka's Algorithm:

https://www.youtube.com/watch?v=jsmMtJpPnhU

https://www.youtube.com/watch?v=X5zEIg_Zuao

Applications of Boruvka's Algorithm:

Developing an efficient electricity network.

Developing the efficient transportation networks

4.4.9 Objective Type Questions:(GATE/PSUs and Company Specific)

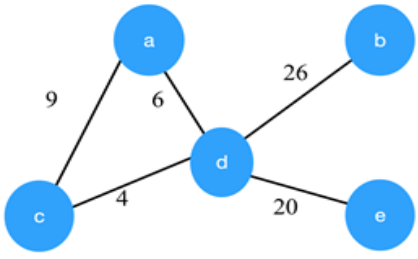
1	Consider a complete graph G with 4 vertices. Graph G has ____ spanning trees.
A	15
B	8
C	16
D	13
ANS	C

2	The travelling salesman problem can be solved using _____

A	Spanning Tree
B	Minimum Spanning Tree
C	Bellman-Ford
D	DFS Traversal
ANS	B

3	Consider an undirected graph G with vertices { A, B, C, D, E}. In graph G, every edge has a distinct weight. Edge CD is edge with minimum weight, and edge AB is edge with maximum weight. Then, which of the following is false?
A	Every minimum spanning tree of G must contain CD
B	If AB is in a minimum spanning tree, then its removal must disconnect G
C	No minimum spanning tree contains AB
D	G has a unique minimum spanning tree
ANS	C

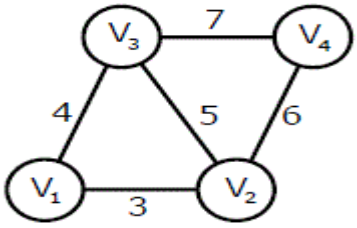
4	Consider graph M with 3 vertices. Its adjacency matrix is shown below. Which of the following is true? $M = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$
A	Graph M has no minimum spanning tree
B	Graph M has a unique minimum spanning trees of cost 2
C	Graph M has 3 distinct minimum spanning trees, each of cost
D	Graph M has 3 spanning trees of different costs
ANS	C

5	Consider the graph shown below. Which of the following are the edges in the MST of the given graph? 
A	(a-c)(c-d)(d-b)(d-b)
B	(c-a)(a-d)(d-b)(d-e)
C	(a-d)(d-c)(d-b)(d-e)
D	(c-a)(a-d)(d-c)(d-b)(d-e)
ANS	C

6	Which of the following is not the algorithm to find the minimum spanning tree of the given graph?
A	Boruvka's Algorithm
B	Prim's Algorithm
C	Kruskal's Algorithm
D	Bellman-Ford Algorithm
ANS	D

7	Which of the following is false?
A	The spanning trees do not have any cycles
B	MST have n – 1 edges if the graph has n edges
C	Edge e belonging to a cut of the graph if has the weight smaller than any other edge in the same cut, then the edge e is present in all the MSTs of the graph.
D	Removing one edge from the spanning tree will not make the graph disconnected.
ANS	D

8	The length of the path from v5 to v6 in the MST of the previous question with n = 10 is
A	11
B	25
C	31
D	41
ANS	C

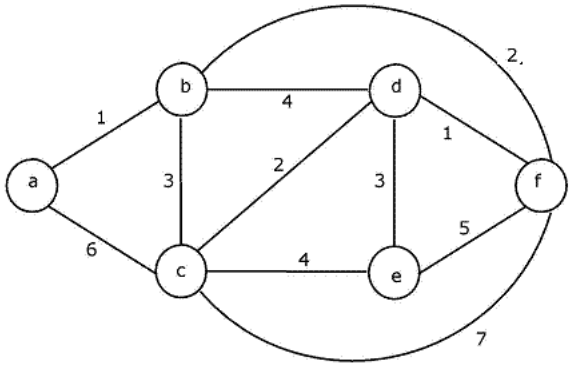
9	An undirected graph G V, E) contains n (n> 2) nodes named v1 , v2 ,….vn. Two nodes vi ,vj are connected if and only if 0 < |i – j| <= 2. Each edge (vi, vj) is assigned a weight i + j. A sample graph with n = 4 is shown below. What will be the cost of the minimum spanning tree (MST) of such a graph with n nodes?  (GATE 2011)
A	1/12(11n^2 – 5n)
B	n^2 – n + 1
C	6n-11
D	2n+1
ANS	B

10	Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. What is the minimum possible weight of a spanning tree T in this graph such that vertex 0 is a leaf node in the tree T? $W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$ (GATE CS 2010)
A	7
B	8
C	9
D	10
ANS	D

11	In the graph given in the above question, what is the minimum possible weight of a path P from vertex 1 to vertex 2 in this graph such that P contains at most 3 edges?
A	7
B	8
C	9
D	10
ANS	B

12	An undirected graph G has n nodes. Its adjacency matrix is given by an n × n square matrix whose (i) diagonal elements are 0's and (ii) non-diagonal elements are 1's. Which one of the following is TRUE?
A	Graph G has no minimum spanning tree (MST)
B	Graph G has a unique MST of cost n-1
C	Graph G has multiple distinct MSTs, each of cost n-1
D	Graph G has multiple spanning trees of different costs
ANS	C

13	Consider the following graph:

	 Which one of the following cannot be the sequence of edges added, in that order, to a minimum spanning tree using Kruskal's algorithm?
A	(a—b),(d—f),(b—f),(d—c),(d—e)
B	(a—b),(d—f),(d—c),(b—f),(d—e)
C	(d—f),(a—b),(d—c),(b—f),(d—e)
D	(d—f),(a—b),(b—f),(d—e),(d—c)
ANS	D

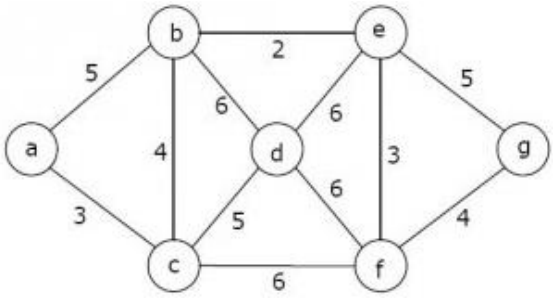
14	Let G be an undirected connected graph with a distinct edge weight. Let e-max be the edge with maximum weight and e-min the edge with minimum weight. Which of the following statements is false? (GATE CS 2000)
A	Every minimum spanning tree of G must contain e-min
B	If e-max is in a minimum spanning tree, then its removal must disconnect G
C	No minimum spanning tree contains e-max
D	G has a unique minimum spanning tree
ANS	C

15	Consider a weighted complete graph G on the vertex set {v1,v2 ,v} such that the weight of the edge (v,v) is 2|i-j|. The weight of a minimum spanning tree of G is:(GATE CS 2006)
A	n-1
B	n-2
C	nC2
D	2
ANS	B

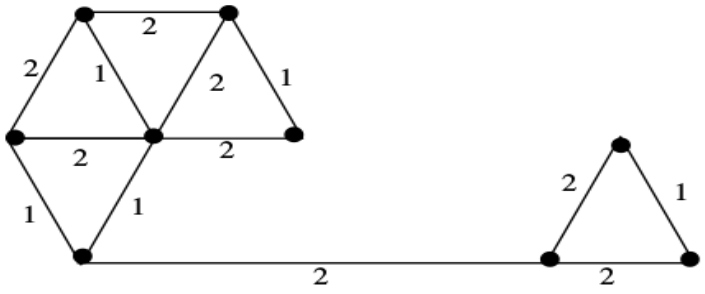
16	Let G be a weighted graph with edge weights greater than one, and Gabe the graph constructed by squaring the weights of edges in G. Let T and T' be the minimum spanning trees of G and G', respectively, with total weights t and t'. Which of the following statements is TRUE? (GATE CS 2012)
A	T' = T with total weight t' = t^2
B	T' = T with total weight t' < t^2

C	T' != T but total weight t' = t^2
D	None of the above
ANS	D

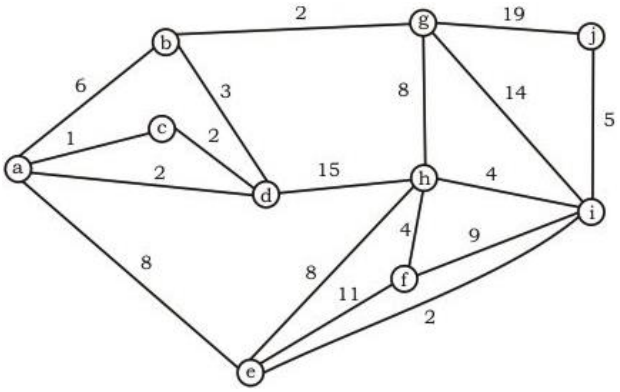
17	What is the largest integer m such that every simple connected graph with n vertices and n edges contains at least m different spanning trees?
A	1
B	2
C	3
D	N
ANS	C

18	Let G be connected undirected graph of 100 vertices and 300 edges. The weight of a minimum spanning tree of G is 500. When the weight of each edge of G is increased by five, the weight of a minimum spanning tree becomes _____. GATE 2015
A	1000
B	995
C	2000
D	1995
ANS	B
19	Consider the following graph:  Which one of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm? (GATE 2009)
A	(b,e)(e,f)(a,c)(b,c)(f,g)(c,d)
B	(b,e)(e,f)(a,c)(f,g)(b,c)(c,d)
C	(b,e)(a,c)(e,f)(b,c)(f,g)(c,d)
D	(b,e)(e,f)(b,c)(a,c)(f,g)(c,d)
ANS	D

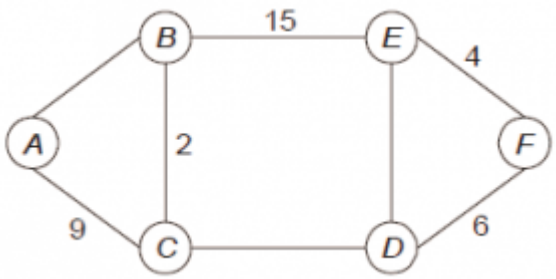
20	The number of distinct minimum spanning trees for the weighted graph below is

	 (GATE 2014)
A	4
B	5
C	6
D	7
ANS	C

21	Let s and t be two vertices in an undirected graph G + (V, E) having distinct positive edge weights. Let [X, Y] be a partition of V such that s ∈ X and t ∈ Y. Consider the edge e having the minimum weight amongst all those edges that have one vertex in X and one vertex in Y. The edge e must definitely belong to (GATE 2005)
A	the minimum weighted spanning tree of G
B	the weighted shortest path from s to t
C	each path from s to t
D	the weighted longest path from s to t
ANS	A

22	What is the weight of a minimum spanning tree of the following graph?  GATE 2003
A	29
B	31
C	38
D	41
ANS	B

23	Consider an undirected unweighted graph G. Let a breadth-first traversal of G be done starting from a node r. Let d(r,u) and d(r,v) be the lengths of the shortest paths from r to u and v respectively in G. If u is visited before v during the breadth-first traversal, which of the following statements is correct? GATE 2001
A	d(r, u) < d(r, v)
B	d(r, u) > d(r, v)
C	d(r, u) <= d(r, v)
D	None of these
ANS	C

24	The graph shown below 8 edges with distinct integer edge weights. The minimum spanning tree (MST) is of weight 36 and contains the edges: {(A, C), (B, C), (B, E), (E, F), (D, F)}. The edge weights of only those edges which are in the MST are given in the figure shown below. The minimum possible sum of weights of all 8 edges of this graph is _____  GATE 2015
A	66
B	69
C	68
D	70
ANS	B

25	Let G be a weighted connected undirected graph with distinct positive edge weights. If every edge weight is increased by the same value, then which of the following statements is/are TRUE? P: Minimum spanning tree of G does not change Q: Shortest path between any pair of vertices does not change. GATE 2015
A	P only
B	Q only
C	Neither P and Q
D	Both P and Q
ANS	A

26	Let G be a complete undirected graph on 4 vertices, having 6 edges with weights being 1, 2, 3, 4, 5, and 6. The maximum possible weight that a minimum weight spanning tree of G can have is. GATE 2016
A	6
B	7
C	8
D	9
ANS	B

27	If Kruskal's algorithm is used for finding a minimum spanning tree of a weighted graph G with n vertices and m edges and edge weights are already given in a sorted list, then, What will be the time complexity to compute the minimum cost spanning tree given that union and find operations take amortized O(1). GATE 2018
A	O(m log n)
B	O(n)
C	O(m)
D	O(n log m)
ANS	C

4.5. Shortest Path

4.5.1. Introduction to Shortest Path:

In order to find a path between two vertices in such a manner that the sum of the weights of edges lies in the path is minimum. We can find the shortest path for the directed, undirected or mixed graph.

Analogy

Road map between two cities. Source city in Greater Noida and Destination city is Meerut. We have a different path between these two cities that can better be understood by Google Maps.


Here we have three images of google Maps; each image contains a different path and different distances. The first image has a distance 84.4 km, the second image has a distance 78 km, and the third image shows the train path between greater Noida to Meerut. Google map is the best example to find the shortest path between the source location to the destination location. These locations can be considered as nodes or vertices in the graph, and roads or paths between cities can be considered as edges between nodes or vertices of the graph.

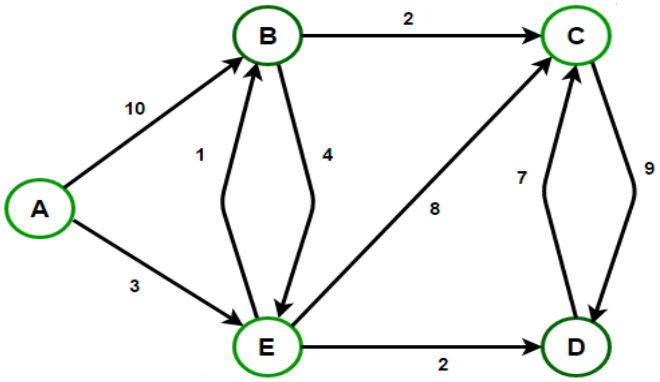
4.5.2. Types of Shortest Path:

Single source shortest path
Find a path between one node to all other nodes.
Dijkstra Algorithm
Bellman-Ford Algorithm
All Pairs Shortest Path
Find a path between all pairs of nodes. All nodes are considered as a source node and destination node.

Floyd-Warshall algorithm
Johnson's algorithm

4.5.3. Single-Source Shortest Path

The Single source shortest path algorithm is used to find the shortest path between a particular node to the other nodes. In a single-source shortest path, first initialize a source node and find the minimum distance between the source node to all other nodes of graph G.
Consider a situation in which a person is in city A and wants to go to city D; there are many paths possible between A to D.



We have different path with different cost

A → B → C → D COST= 10 + 2 + 9 = 21
A → B → E → D COST= 10 + 4 + 2 = 16
A → B → E → C → D COST= 10 + 4 + 8 + 9 = 31
A → E → B → C → D COST= 3 + 1 + 2 + 9 = 15
A → E → C → D COST= 3 + 8 + 9 = 20
A → E → D COST= 3 + 2 = 5

We can see that there are many paths possible between city A to city D. This is very complex to find all possible path between one node to all other nodes, here we find only one combination from A to D. if we find A to B, A to C, A to D and A to E, then there are many possible paths. After finding all the paths, we have to find a minimum path between one city to all other cities. To resolve this problem, we have two algorithms that solves different types of problems of the single-source shortest path-

Dijkastra's Algorithm

Undirected graph or directed graph
Non-negative weights
Bellman-ford Algorithm
Directed graph
Non-negative weights are allowed
Both the algorithms initialize the single source and update the weight. To initialize the source node

ALGORITHM Initialize_single_source(G,s)
BEGIN:
FOR each vertex v ϵ V[G] DO // loop executes for all vertices of graph
d[v] = ∞ // Initially weight of all vertices is assigned ∞. We consider that there // is no path between any vertices. Update weight of vertices after // traversal the path between two vertices.
∏[v] = NIL // Initially we consider that there is no path so there is no predecessor // of any vertex.
d[s] = 0 // Assign 0 weight to the starting/ source vertex.
END;
Relax algorithm is called for updating the weight of the vertex.
ALGORITHM Relax(u, v, w)
BEGIN:
IF d[v] > d[u] + w(u, v) THEN //existing weight (d[v]) is greater than current weight(weight // at vertex u + weight of edge between u & v)
d[v] = d[u] + w(u, v) // if condition true then update weight of vertex v else leave // as it is
∏[v] = u // u is predecessor of v
END;

4.5.4. Dijkstra's Algorithm

Dijkstra's algorithm is a single source shortest path algorithm in which we find the shortest path between the source node to other nodes. It takes a weighted graph as input and gives the shortest path between the source node to other nodes. We can apply Dijkstra's algorithm only when the graph is directed, weighted, and weights are non-negative. If weights are negative, then we can apply bellman-ford, which we will discuss later.

Procedure
Step1:First we initialize the single source.
Step 2: Set weight of source vertex as 0, and weight of other vertices as ∞.
Step 3: Set a priority Queue for all vertices on the basis of weight.
Step 4:Extract the min from the priority Queue.
Step 5: Find the adjacent node update the weight of the adjacent node if the net weight is less than the existing weight else retain existing weight.
Step 6:Update priority queue.
Step 7:Repeat steps 4 to 6 until the priority queue empty.

ALGORITHM Dijkastra(G, w[ ][ ], s)
BEGIN:
Initialize_single_source(G, s) // Initialize the source/starting vertex of the graph
S = Ø // S contain the traversed node initially empty
Initialize(PQ) //PQ is a Priority Queue

```
FOR all u ϵ V[G] DO
        EnQueue(PQ, u)          // Priority Queue initially contain all the vertices of graph
WHILE !Empty(PQ) DO             // loop executes until Q empty (all vertices traversed)
    u = DeQueue(PQ)             // Extract the minimum from the priority queue for traversal
    S = S U {u}                 // Union traversed vertex with S.
    FOR each vertex v ϵAdj[u] DO// loop executes for the all-adjacent vertices of u
        Relax(u, v, w)          // Relax function update the value of adjacent vertices of u.
END;
```

Complexity

The complexity of Dijkstra's algorithm is dependent upon the representation of the graph. There are two methods to represent the graph

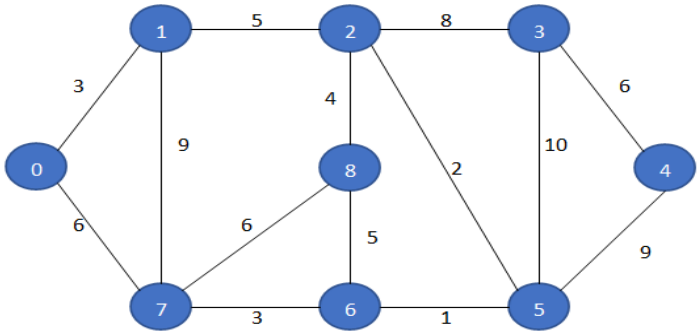
Array representation

Adjacency list representation

If we represent a graph using an array, then the time complexity of Dijkstra's algorithm is $O(V^2)$. If we represent the graph using the Adjacency list, Time Complexity can be reduced to O(E log V) with the help of binary heap.

Example

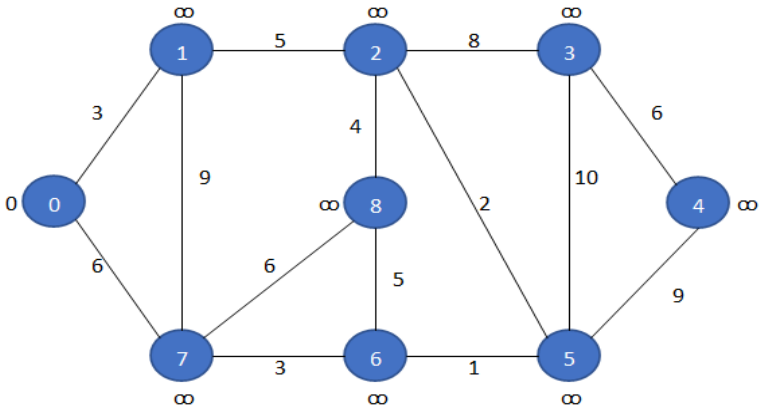
Find the shortest path from node 0 to other nodes.



Step 1 weight of Source node = 0

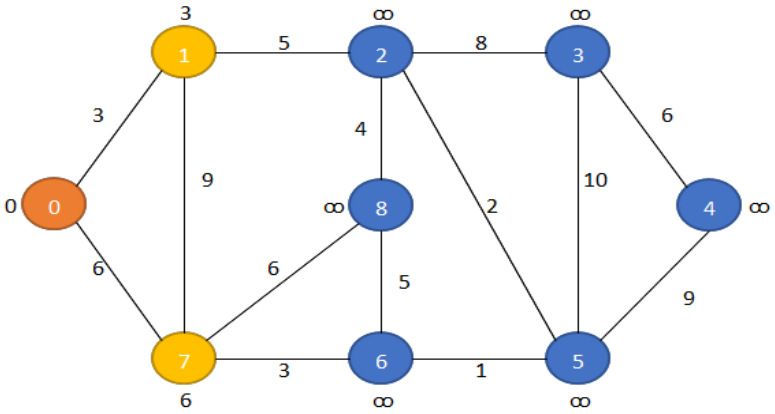
Weight of other nodes = ∞

0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-



Step 2Extract min node that is node 0 and updates weight of adjacent vertices that is 1 and 6 with respective weight 3 and 6.

0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-
0	3	∞	∞	∞	∞	∞	6	∞	0	1,7



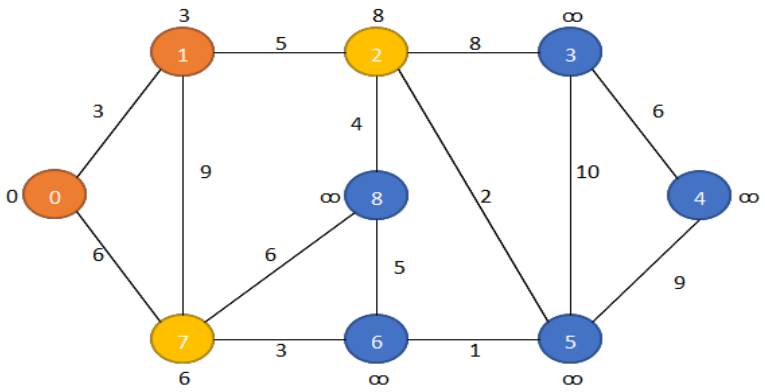
Step 3 Extract min = 1, adjacent vertices = 2, 7. Update only node 2 but not 7 because the existing cost of 7 is less than the current cost.

Existing weight of node 7 = 6

current weight= weight of node 1 + weight (1, 7)

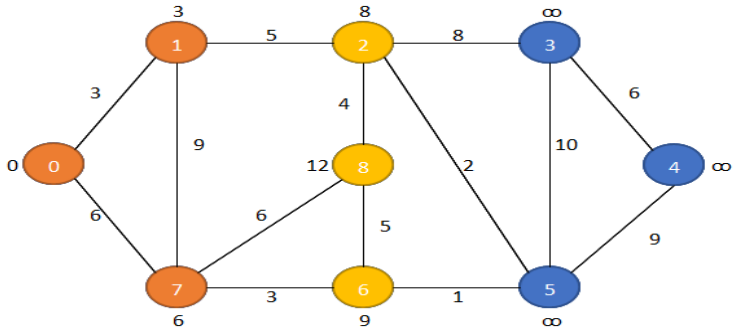
current weight = 3 + 9 = 12

0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-
0	3	∞	∞	∞	∞	∞	6	∞	0	1,7
	3	8	∞	∞	∞	∞	6	∞	1	2



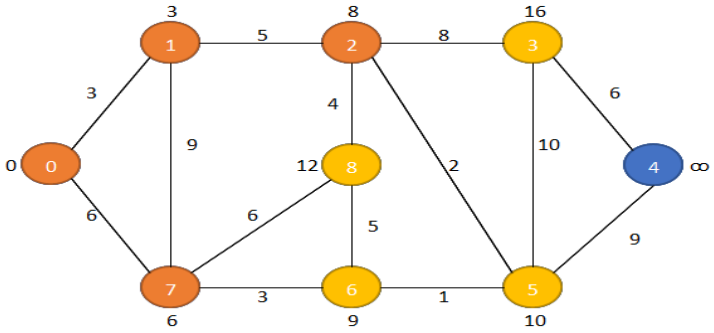
Step 4Similarly update the weight of nodes 6 and 8.

0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-
0	3	∞	∞	∞	∞	∞	6	∞	0	1,7
	3	8	∞	∞	∞	∞	6	∞	1	2
		8	∞	∞	∞	9	6	12	7	6,8



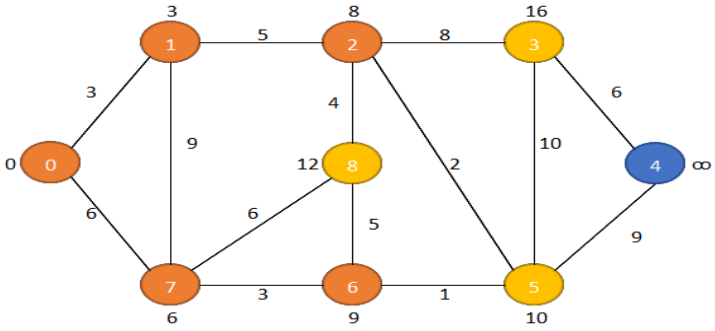
Step 5 Similarly update the weight of nodes 3 and 5.

0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-
0	3	∞	∞	∞	∞	∞	6	∞	0	1,7
	3	8	∞	∞	∞	∞	6	∞	1	2
		8	∞	∞	∞	9	6	12	7	6,8
		8	16	∞	10	9		12	2	3,5



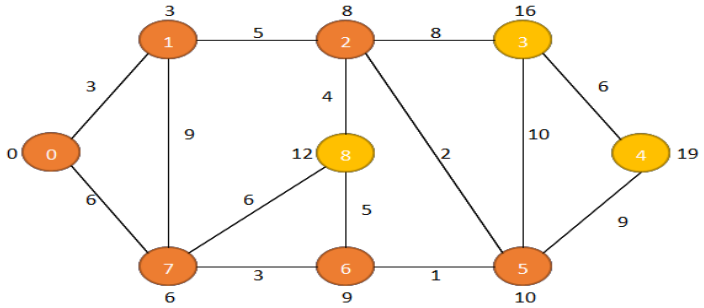
Step 6No update

0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-
0	3	∞	∞	∞	∞	∞	6	∞	0	1,7
	3	8	∞	∞	∞	∞	6	∞	1	2
		8	∞	∞	∞	9	6	12	7	6,8
		8	16	∞	10	9		12	2	3,5
			16	∞	10	9		12	6	-



Step 7Similarly update weight of node 4

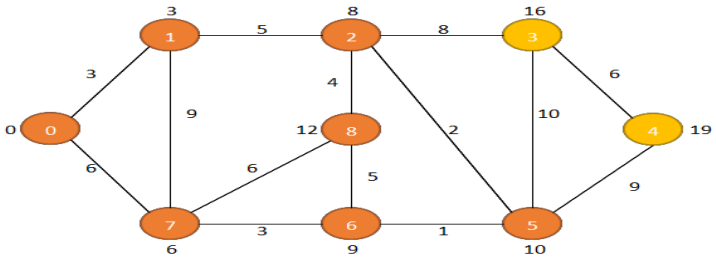
0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-
0	3	∞	∞	∞	∞	∞	6	∞	0	1,7
	3	8	∞	∞	∞	∞	6	∞	1	2
		8	∞	∞	∞	9	6	12	7	6,8
		8	16	∞	10	9		12	2	3,5
			16	∞	10	9		12	6	-
			16	19	10			12	5	4



Step 8No Update

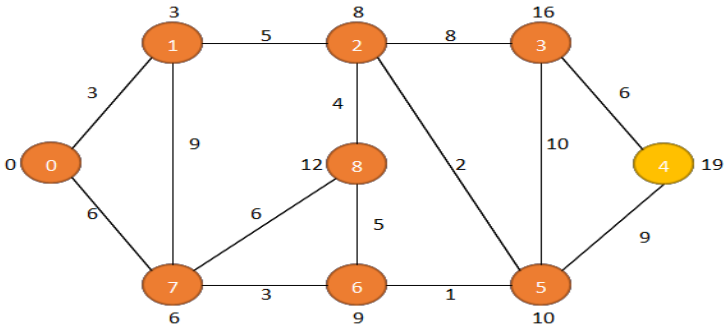
0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-
0	3	∞	∞	∞	∞	∞	6	∞	0	1,7
	3	8	∞	∞	∞	∞	6	∞	1	2

		8	∞	∞	∞	9	6	12	7	6,8
		8	16	∞	10	9		12	2	3,5
			16	∞	10	9		12	6	-
			16	19	10			12	5	4
			16	19				12	8	-



Step 9No Update

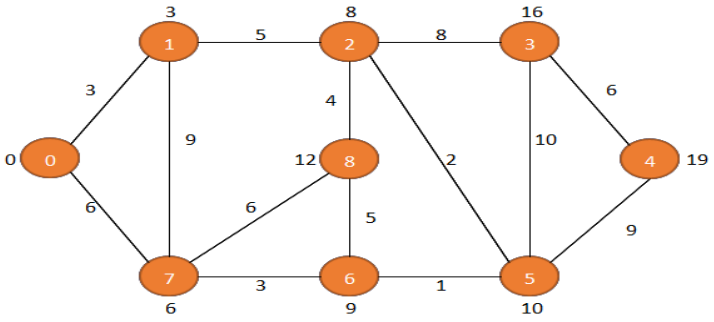
0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-
0	3	∞	∞	∞	∞	∞	6	∞	0	1,7
	3	8	∞	∞	∞	∞	6	∞	1	2
		8	∞	∞	∞	9	6	12	7	6,8
		8	16	∞	10	9		12	2	3,5
			16	∞	10	9		12	6	-
			16	19	10			12	5	4
			16	19				12	8	-
			16	19					3	-



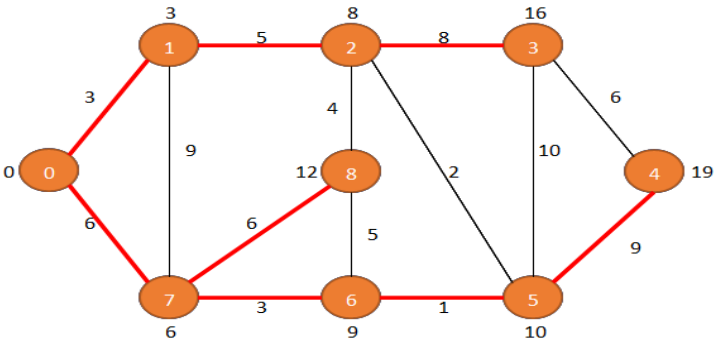
Step 10 Last Node Traversed

0	1	2	3	4	5	6	7	8	Current Node	Node value updated
0	∞	∞	∞	∞	∞	∞	∞	∞	-	-
0	3	∞	∞	∞	∞	∞	6	∞	0	1,7
	3	8	∞	∞	∞	∞	6	∞	1	2
		8	∞	∞	∞	9	6	12	7	6,8
		8	16	∞	10	9		12	2	3,5
			16	∞	10	9		12	6	-

			16	19	10			12	5	4
			16	19				12	8	-
			16	19					3	-
				19					4	-

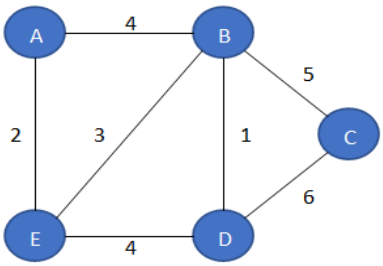


Step 11Final shortest path between source node to other nodes.

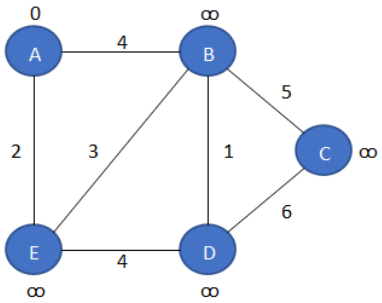


Example 2

Find the single source shortest path using Dijkstra's Algorithm if the source node is A.

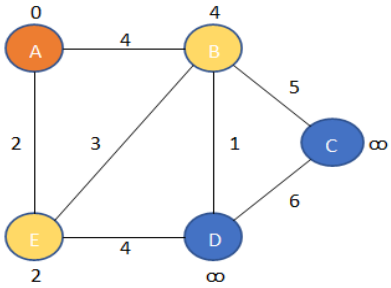


Step 1:



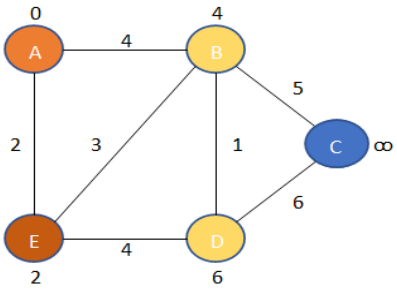
A	B	C	D	E	Current Node	Node value updated
0	∞	∞	∞	∞	-	-

Step 2:



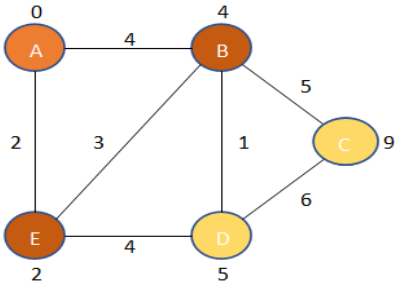
A	B	C	D	E	Current Node	Node value updated
0	∞	∞	∞	∞	-	-
0	4	∞	∞	2	A	B,E

Step 3:



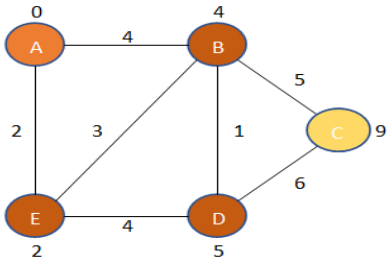
A	B	C	D	E	Current Node	Node value updated
0	∞	∞	∞	∞	-	-
0	4	∞	∞	2	A	B,E
	4	∞	6	2	E	D

Step 4:



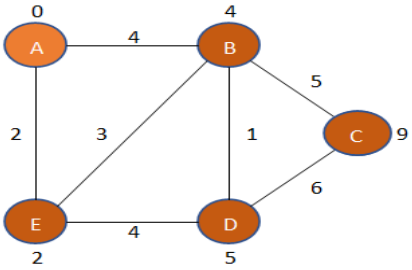
A	B	C	D	E	Current Node	Node value updated
0	∞	∞	∞	∞	-	-
0	4	∞	∞	2	A	B,E
	4	∞	6	2	E	D
	4	∞	6		B	C,D

Step 5:



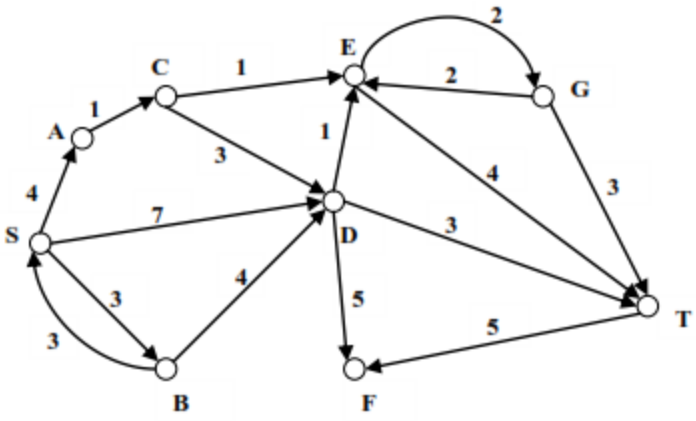
A	B	C	D	E	Current Node	Node value updated
0	∞	∞	∞	∞	-	-
0	4	∞	∞	2	A	B,E
	4	∞	6	2	E	D
	4	∞	6		B	C,D
		9	5		D	-

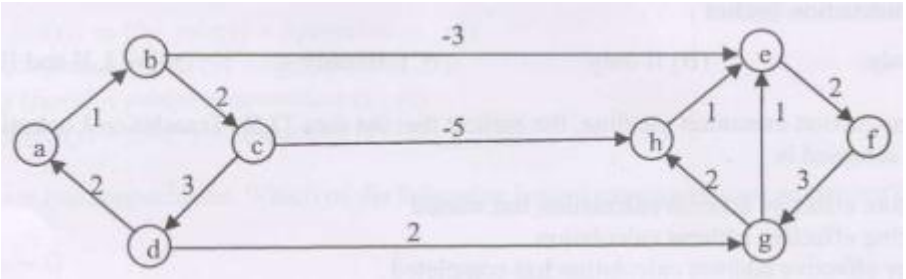
Step 6:



A	B	C	D	E	Current Node	Node value updated
0	∞	∞	∞	∞	-	-
0	4	∞	∞	2	A	B,E
	4	∞	6	2	E	D
	4	∞	6		B	C,D
		9	5		D	-
		9			C	-

4.5.5. Multiple Choice Questions:

	GATE 2012 Marks 2
1	Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex *v* is updated only when a strictly shorter path to *v* is discovered. 
A	SDT
B	SBDT
C	SACDT
D	SACET
AN	SACET

	GATE 2008 Marks 2
2	Dijkstra's single-source shortest path algorithm, when run from vertex *a* in the above graph, computes the correct shortest path distance to 
A	Only vertex a
B	Only vertices a, e, f, g, h
C	Only vertices a, b, c, d
D	All the vertices
AN	All the vertices

	GATE 2007 Marks 2
3	In an unweighted, undirected connected graph, the shortest path from a node S to every other node is computed most efficiently, in terms of *time complexity*, by

A	Dijkstra’s algorithm starting from S.
B	Warshall’s algorithm
C	Performing a DFS starting from S.
D	Performing a BFS starting from S.
AN	Performing a BFS starting from S.

4	Dijkstra’s Algorithm is used to solve _____ problems.
A	All pair shortest path
B	Single source shortest path
C	Network Flow
D	Sorting
AN	Single source shortest path

5	What is the time complexity of Dijkstra's algorithm?
A	$O(N^2)$
B	$O(N^3)$
C	O(N)
D	O(logN)
AN	$O(N^2)$

4.5.6. Bellman Ford Algorithm:

Dijkstra algorithm is "more efficient" than the Bellman-Ford algorithm then why do we use the Bellman-Ford algorithm?

Dijkstra’s algorithm is an efficient single source shortest path algorithm. It works better than Bellman-ford algorithm when the edge weights are positive. But if we have negative weights in graph than Dijkstra may or may not provide a solution. So there is no guarantee for a solution. In this scenario we use Bellman-ford algorithm it not only will provides the solution for negative-weighted edges, it will also tell that there exists a negative-weighted cycle in the graph or not.
Why would one ever have edges with negative weights in real life?

Negative weight edges might seem useless at first sight but through them we can easily explains various phenomena like cashflow, or the heat absorbed/ released process in chemical reactions, or others chemical reactions like change in temperature (solid to liquid and liquid to solid) etc. For example, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption. If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

Bellman-Ford algorithm

In many applications we want to find the single source shortest path problem where there may be weights that are negative. For a given directed graph G= (V, E) where V denotes the number of vertices and E represents the edges of graph, the weight function w: E → R and s is the source vertex of the graph. The output of the bellman-ford algorithm is a Boolean value (true/ false) that states that whether there is a negative cycle in the graph or not. The true indicates that there is a negative cycle in the graph and there is no solution for that graph. Otherwise, it will produce the shortest path with their weights.

This algorithm also makes use of the relaxation function progressively to reduce the estimate d[v] on weight in shortest path from source to every vertex v till if gets the actual shortest path weights.

So for each edge u-v

If dist[v] > dist[u]+ weights [u,v],

then dist[v] ← dist[u]+ weights [u,v]

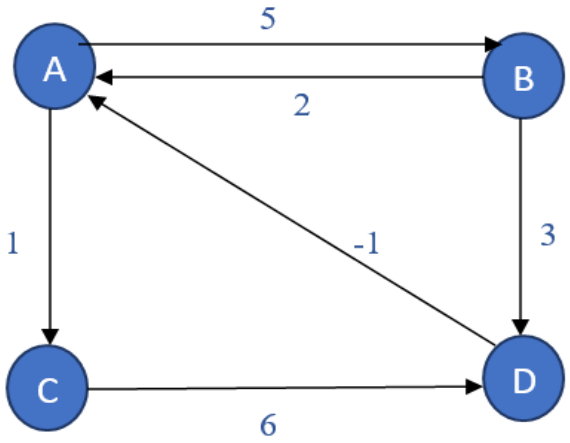
π [v]← u.

```
BELLMAN -FORD (G, w, s)
 1. INITIALIZE - SINGLE - SOURCE (G, s)
 2. for i ← 1 to |V[G]| - 1                    //O(V) time
 3.       do for each edge (u, v) ∈ E [G]      //O(E) time
 4.             do RELAX (u, v, w)             // O(1) time
 5. for each edge (u, v) ∈ E [G]
 6.       do if d [v] > d [u] + w (u, v)
 7.             then return FALSE.
 8. return TRUE.
```

Complexity analysis:

In the above algorithm G is the graph given having a edge set and vertices set, w is the weight matrix and s is the source node of the graph. Line number 1 is the initialization step. In this we are assigning infinity to every node except the source node, in source node s we are assigning so time taken will be O(V). In line number 2 we are running a loop from 1 to V[G]-1 because we want to relax the edges from 1 to V-1 times so this outer loop will run for (V-1) times. The inner loop in line number 3 will run for (E) times as we running it to the number of edges. The line 4 will run as implementation requires only array and we are just decreasing the value so it will take O(1) time. So line number 2- 4 will take (V-1)*(E) time i.e. O(V*E). Line number 5- 8 checks for each edge from the edge set so it will run for O(E) times, and we are just checking whether any vertex whose weight can be reduced, if it can be reduced then there will be a negative cycle otherwise, we can have the solution for the graph. So the complexity of the bellman ford algorithm is O(V*E)

Example 1:



Step 1 weight of Source node = 0
Weight of other nodes = ∞

Distance Matrix:

A	B	C	D
0	∞	∞	∞

Parent Matrix:

A	B	C	D
0	NIL	NIL	NIL

STEP NO. 2:

Let all the edges be processed in following order (A, B), (A, C), (B, A), (B, D), (C, D) and (D, A).

For i= V[G]-1 // i= 4-1 =3

i=1

Apply Relax Algorithm on each edge.
Relax(u,v,w)
If dist[v] > dist[u]+ weights [u,v],
then dist[v] ← dist[u]+ weights [u,v]
π [v]← u.

Now for edge A→ B

If dist[B] > dist[A] + weight[A,B] // if ∞ > 0 + 5 yes
Then dist[B] ← dist[A] + weight [A, B] then dist[B] ← 0+5 =5
π [B]← A π [B]← A

Similarly for all the edges apply relax algorithm.

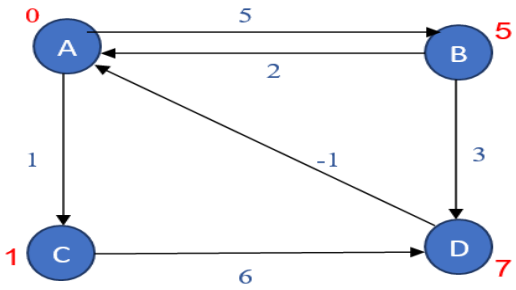
Edges	Weights	i=1
A→ B	5	dist[B] = 5
A→ C	1	dist[C] = 1
B→A	2	dist[A] = 0
B→D	3	dist[D] = 8
C→D	6	dist[D] = 7
D→A	-1	dist[A] = 0

Distance Matrix:

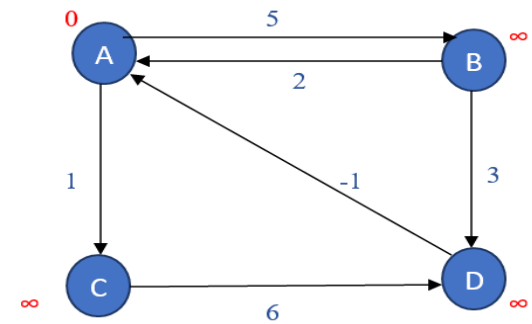
	A	B	C	D
i=0	0	∞	∞	∞
i = 1	0	5	1	7

Parent Matrix:

	A	B	C	D
i=0	NIL	NIL	NIL	NIL
i = 1	NIL	A	A	C



STEP No. 3 : Now i = 2. Apply relax algorithm on every edge.

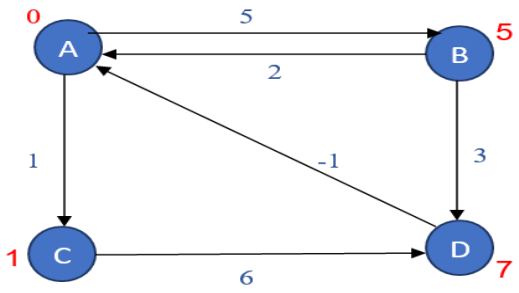


Edges	Weights	i=1	i=2

A→ B	5	dist[B] = 5	dist[B] = 5
A→ C	1	dist[C] = 1	dist[C] = 1
B→A	2	dist[A] = 0	dist[A] = 0
B→D	3	dist[D] = 8	dist[D] = 7
C→D	6	dist[D] = 7	dist[D] = 7
D→A	-1	dist[A] = 0	dist[A] = 0

Distance Matrix:

	A	B	C	D
i=0	0	∞	∞	∞
i = 1	0	5	1	7
i = 2	0	5	1	7



Parent Matrix:

	A	B	C	D
i=0	NIL	NIL	NIL	NIL
i = 1	NIL	A	A	C
i=2	NIL	A	A	C

STEP No. 4 : Now i = 3. Apply relax algorithm on every edge.

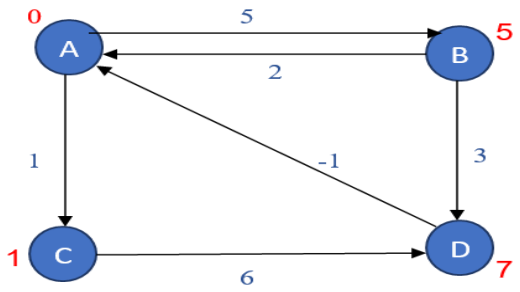
Edges	Weights	i=1	i=2	i=3
A→ B	5	dist[B] = 5	dist[B] = 5	dist[B] = 5
A→ C	1	dist[C] = 1	dist[C] = 1	dist[C] = 1
B→A	2	dist[A] = 0	dist[A] = 0	dist[A] = 0
B→D	3	dist[D] = 8	dist[D] = 7	dist[D] = 7
C→D	6	dist[D] = 7	dist[D] = 7	dist[D] = 7
D→A	-1	dist[A] = 0	dist[A] = 0	dist[A] = 0

Distance Matrix:

	A	B	C	D
i=0	0	∞	∞	∞

i = 1	0	5	1	7
i = 2	0	5	1	7
i = 3	0	5	1	7

Parent Matrix:



	A	B	C	D
i=0	NIL	NIL	NIL	NIL
i = 1	NIL	A	A	C
i=2	NIL	A	A	C
i=3	NIL	A	A	C

STEP No. 5 : Now run last iteration to check if there exits a negative weight cycle or not.

Check the condition If dist[v] > dist[u]+ weights [u,v],

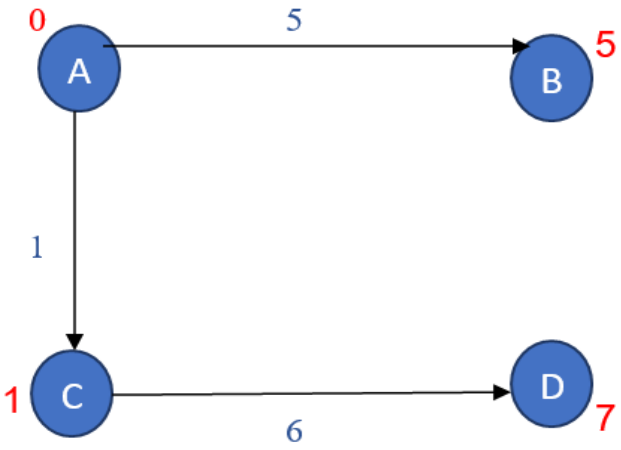
Then return false // It means if condition is true then it contains the negative weight cycle.

Now check for all the edges:

Edges	Weights	If dist[v] > dist[u] + weight[u,v]
A→ B	5	NO
A→ C	1	NO
B→A	2	NO
B→D	3	NO
C→D	6	NO
D→A	-1	NO

Now condition is false of each edge so this graph does not contain any cycle.

STEP NO.6 : So final graph would be



A→ B = 5

A→ C = 1

A→ D= A→ C→ D = 1+6 =7

Example No. 2: Consider the following the graph

Step 1 weight of Source node = 0

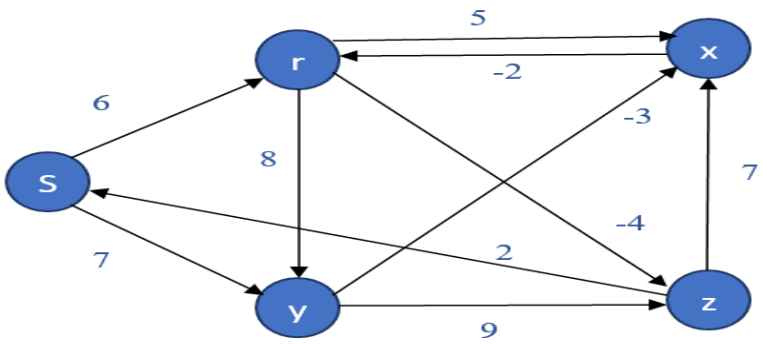
Weight of other nodes = ∞

Step 1 weight of Source node = 0

Weight of other nodes = ∞

Step 1 weight of Source node = 0

Weight of other nodes = ∞



STEP NO. 1: weight of Source node = 0

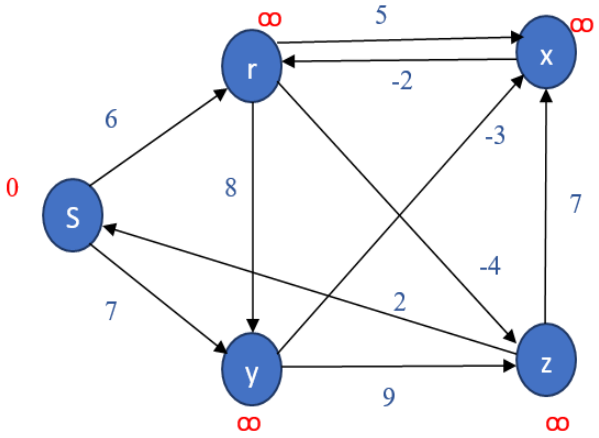
Weight of other nodes = ∞

Distance Matrix:

s	r	x	y	Z
0	∞	∞	∞	∞

Parent Matrix:

s	r	x	y	Z
NIL	NIL	NIL	NIL	NIL



STEP NO. 2:

Let all the edges be processed in following order (s, r), (s, y), (r, y), (y, z), (z, x), (r, z), (z, s), (r, x) and (x, r), (y,x).

For i= V[G]-1 // i= 5-1 =4

i=1

Apply Relax Algorithm on each edge.
Relax(u,v,w)
If dist[v] > dist[u]+ weights [u,v],
then dist[v] ← dist[u]+ weights [u,v]
π [v]← u.

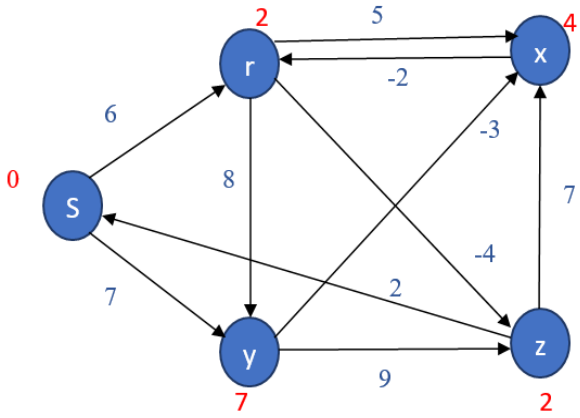
For all edges apply relax algorithm.

Edges	Weights	i=1
s→r	6	dist[r] = 6
s→y	7	dist[y] = 7
r→y	8	dist[y] = 7
y→z	9	dist[z] = 16
z→x	7	dist[x] = 23

r→z	-4	dist[z] = 2
z→s	2	dist[s] = 0
r→x	5	dist[x] = 11
x→r	-2	dist[r] = 2
y→x	-3	dist[x]= 4

Distance Matrix:

	s	r	x	y	z
i=0	0	∞	∞	∞	∞
i=1	0	2	4	7	2



Parent Matrix:

	s	r	x	y	z
i=0	NIL	NIL	NIL	NIL	NIL
i=1	NIL	x	y	s	r

STEP No. 3 : Now i = 2. Apply relax algorithm on every edge.

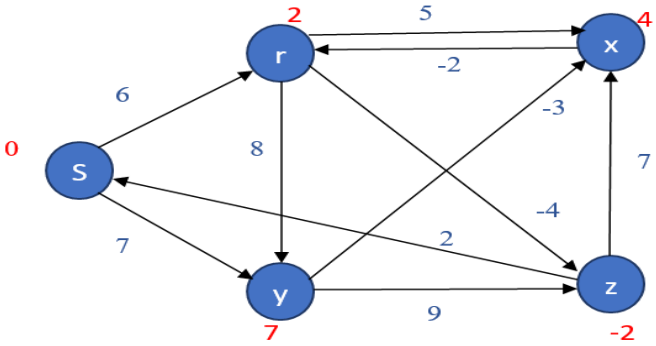
Edges	Weights	i=1	i=2
s→r	6	dist[r] = 6	dist[r] = 2
s→y	7	dist[y] = 7	dist[y] = 7
r→y	8	dist[y] = 7	dist[y] = 7
y→z	9	dist[z] = 16	dist[z] = 2
z→x	7	dist[x] = 25	dist[x] = 4
r→z	-4	dist[z] = 2	dist[z] = -2
z→s	2	dist[s] = 0	dist[s] = 0
r→x	5	dist[x] = 11	dist[x] = 4
x→r	-2	dist[r] = 2	dist[r] = 2
y→x	-3	dist[x]= 4	dist[x]= 4

Distance Matrix:

	s	r	X	y	z
i=0	0	∞	∞	∞	∞
i=1	0	2	4	7	2
i=2	0	2	4	7	-2

Parent Matrix:

	s	r	X	y	z
i=0	NIL	NIL	NIL	NIL	NIL
i=1	NIL	x	y	s	r
i=2	NIL	x	y	s	r



STEP No. 4 : Now i = 3. Apply relax algorithm on every edge.

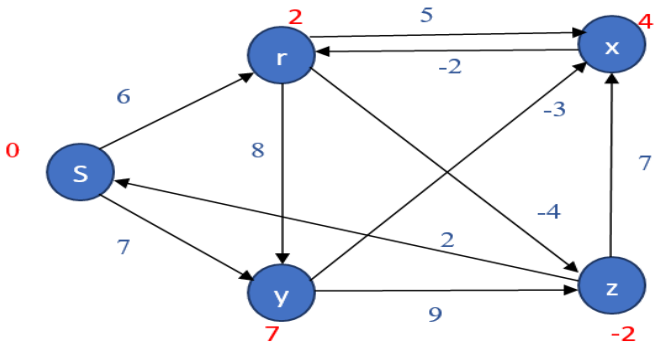
Edges	Weights	i=1	i=2	i=3
s→r	6	dist[r] = 6	dist[r] = 2	dist[r] = 6
s→y	7	dist[y] = 7	dist[y] = 7	dist[y] = 7
r→y	8	dist[y] = 7	dist[y] = 7	dist[y] = 7
y→z	9	dist[z] = 16	dist[z] = 2	dist[z] = -2
z→x	7	dist[x] = 25	dist[x] = 4	dist[x] = 4
r→z	-4	dist[z] = 2	dist[z] = -2	dist[z] = -2
z→s	2	dist[s] = 0	dist[s] = 0	dist[s] = 0
r→x	5	dist[x] = 11	dist[x] = 4	dist[x] = 4
x→r	-2	dist[r] = 2	dist[r] = 2	dist[r] = 2
y→x	-3	dist[x]= 4	dist[x]= 4	dist[x]= 4

Distance Matrix:

	s	r	x	y	z
i=0	0	∞	∞	∞	∞
i=1	0	2	4	7	2
i=2	0	2	4	7	-2
i=3	0	2	4	7	-2

Parent Matrix:

	s	r	x	y	z
i=0	NIL	NIL	NIL	NIL	NIL
i=1	NIL	x	y	s	r
i=2	NIL	x	y	s	r
i=3	NIL	x	y	s	r

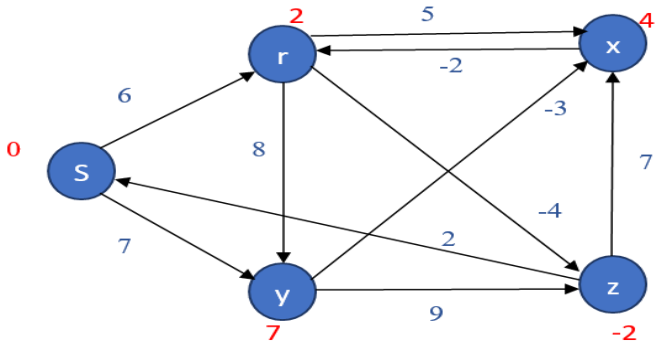


STEP No. 5 : Now i = 4. Apply relax algorithm on every edge.

Edges	Weights	i=1	i=2	i=3	i=4
s→r	6	dist[r] = 6	dist[r] = 2	dist[r] = 6	dist[r] = 6
s→y	7	dist[y] = 7	dist[y] = 7	dist[y] = 7	dist[y] = 7
r→y	8	dist[y] = 7	dist[y] = 7	dist[y] = 7	dist[y] = 7
y→z	9	dist[z] = 16	dist[z] = 2	dist[z] = -2	dist[z] = -2
z→x	7	dist[x] = 25	dist[x] = 4	dist[x] = 4	dist[x] = 4
r→z	-4	dist[z] = 2	dist[z] = -2	dist[z] = -2	dist[z] = -2
z→s	2	dist[s] = 0	dist[s] = 0	dist[s] = 0	dist[s] = 0
r→x	5	dist[x] = 11	dist[x] = 4	dist[x] = 4	dist[x] = 4
x→r	-2	dist[r] = 2	dist[r] = 2	dist[r] = 2	dist[r] = 2
y→x	-3	dist[x]= 4	dist[x]= 4	dist[x]= 4	dist[x]= 4

Distance Matrix:

	s	R	x	y	z
i=0	0	∞	∞	∞	∞
i=1	0	2	4	7	2
i=2	0	2	4	7	-2
i=3	0	2	4	7	-2
i=4	0	2	4	7	-2



Parent Matrix:

	s	r	x	y	z
i=0	NIL	NIL	NIL	NIL	NIL
i=1	NIL	x	y	s	r
i=2	NIL	x	y	s	r
i=3	NIL	x	y	s	r
i=4	NIL	x	y	s	r

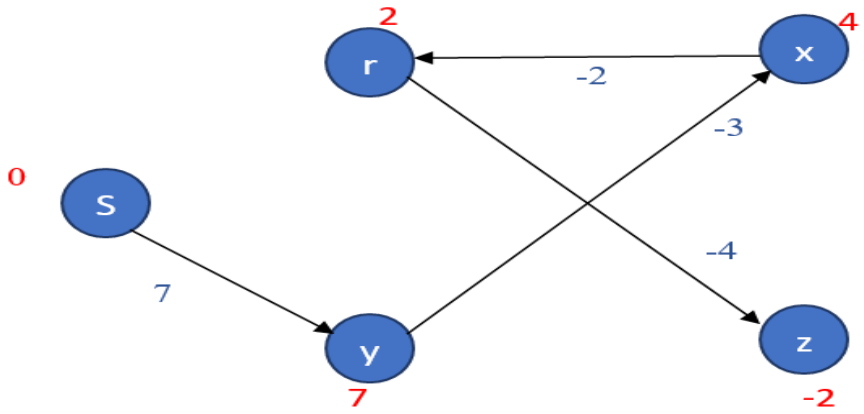
STEP No. 6 : Now run last iteration to check if there exits a negative weight cycle or not.
Check the condition If dist[v] > dist[u]+ weights [u,v],
Then return false // It means if condition is true then it contains the negative weight cycle.

Now check for all the edges:

Edges	Weights	If dist[v] > dist[u] + weight[u,v]
s→r	6	NO
s→y	7	NO
r→y	8	NO
y→z	9	NO
z→x	7	NO
r→z	-4	NO

z→s	2	NO
r→x	5	NO
x→r	-2	NO
y→x	-3	NO

STEP NO.7 : So final graph would be



s→r = s→y→x→r =2
s→y = 7
s→x = s→y→x = 4
s→z= s→y→x→r→z= -2

4.5.7. Introduction to All Pair Shortest Path:

Transitive Closure

Suppose we are at airport x and want to reach another airport, y, z, a, b, etc. If we can reach airport y from x, there will exist a path between them or say airport y is reachable through airport x. Similarly, if we can reach airport z through y or through some other airports, then there would be a path between airport x and y, and we can say airport z is reachable through airport x.

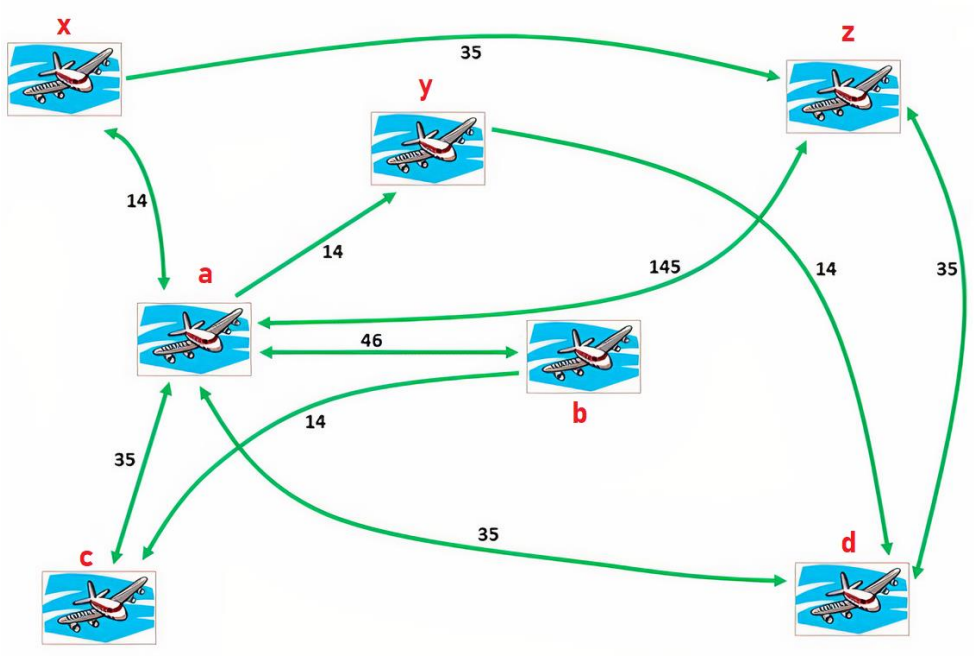


Figure: Different airports

Definition

Transitive Closure in a graph shows the path between the nodes. In other words, if there exists a path between node x and node y, then there should be a corresponding edge(s) between these in Graph.

The final transitive Closure $T_{ij}^{(k)}$ of a graph is a Boolean matrix, where if there is a path between two vertices, it is indicated by 1 in the Matrix, otherwise 0. Here, k is the set of all intermediate vertices from {1, 2, 3, …, k}.
The recursive definition of transitive Closure of a graph $T_{ij}^{(k)}$ is given as below:

For k=0 that is, $T_{ij}^{(0)} = \begin{cases} 0, \text{ if } i \neq j \text{ and } (i,j) \notin E \\ 1, \text{ if } i = j \text{ or } (i,j) \in E \end{cases}$ Here E = (i, j) that is edge between 2 vertices

Note:- Here, we are considering that there should exist a path when the minimum number of nodes is two unless there exists a self-loop on the node. Let us understand this with some examples of directed Graph.

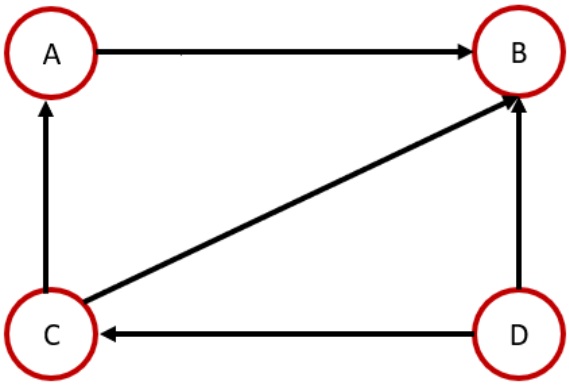


Figure: - Example of Directed Graph

Step 1 In the above Graph, when we convert it into adjacency matrix, the equivalent matrix will be shown below. The algorithm runs exactly till it reaches the number of nodes present in Graph. Here it will run four times.

A1 matrix shows all path having no intermediate nodes between any two nodes

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

Figure:- Computing A^1 Matrix

Now, we will compute all nodes that can be reached via some nodes using the matrix multiplication method. Here, the above path matrix shows all possible edges of length one. In other words, it shows all that path where there exists a direct edge.

STEP 2 Now, we will find all path matrices of length 2. For finding such a Matrix, we will multiply A^1 x A^1. The output of this Matrix consists of those nodes which can be reached via using one intermediate node.

A^2 =

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

X

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

A^2 matrix is calculated by multiplying A^1 X A^1

Figure:- Computing A^2 Matrix

A^2 =

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

A^2 matrix shows all paths having one intermediate node between any two nodes.

Figure:- Final A^2 Matrix

The above Matrix, so the path matrix of all paths of length 2 exists in the original Matrix.

Step 3 Now, we will find all such path matrices whose length will be 3. For finding such a Matrix, we will multiply A^2 x A^1. The output of this Matrix consists of those nodes which can be reached via using two intermediate nodes.

A^3 =

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

X

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

A^3 matrix is calculated by multiplying A^2 X A^1

Figure:- Computing A^3 Matrix

A^3=

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

A^3 matrix shows all paths having three intermediate nodes between any two nodes.

Figure:- Final A^3 Matrix

Step 4 The last step will be to find the path matrix of length 4 in the given Matrix. For finding such a Matrix, we will multiply A^3 x A^1. The output of this Matrix consists of those nodes which can be reached via using three intermediate nodes.

A^4 =

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

X

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

A^4 matrix shows all path having three intermediate nodes between any two nodes.

Figure:- Computing A^4 Matrix

A^4 =

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

A^4 matrix is calculated by multiplying A^3 X A^1

Figure:- Final A^4 Matrix

Step 5 To find the path matrix, the initial phase algorithm runs according to the number of nodes in the Graph. In order to calculate the resultant path matrix, if there are V1, V2, V3, …, Vn nodes than

Path matrix is = $A^1 + A^2 + A^3 + \ldots + A^n$

In the above example, to show all the path length in this Matrix, we need to add all the Matrix we have calculated. The final $A^1 + A^2 + A^3 + A^4$, we get

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

Figure:- Path Matrix obtained

Step 6 The final transitive Closure can be calculated using the following formula.
If a[i][j] != 0 in path matrix than 1 else 0.

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

Figure:- Resultant Transitive Closure

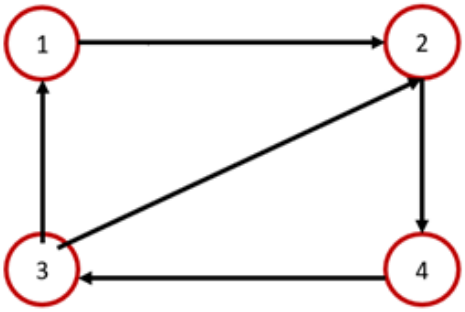
ALGORITHM TransitiveClosure(A[ ], N)

```
BEGIN:
M[N][N] = {0}
X[N][N] = {0}
For I = 1 to N-1 DO
        X = matrix multiply (X, A)
        M = M + X
For I = 1 to N DO
For J = 1 to N DO
                If M[i][j]!= 0 THEN
                        M[i][j] = 1
RETURN M
End;
```

Complexity: Transitive Closure has a solution via graph traversal as well for each Vertex in the Graph. If there is reachability for a vertex, then in the Matrix, the element is filled with 1. If the Graph is represented with an adjacency matrix, the cost is $\Theta(n^3)$, where *n* specifies the number of vertices/nodes in the given Graph.

Example 2:-

The second example also helps us in understanding how to calculate transitive Closure using matrix multiplication. Here, also we have taken the example of a directed Graph.



Step 1 We will compute all nodes that can be reached via the matrix multiplication method via some nodes. Here, the above path matrix shows all possible edges of length one. In other words, it shows the entire path where there exists a direct edge.

A1 =

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

A1 matrix shows all path having no intermediate nodes between any two nodes

Figure:- Computing A1 matrix

Step 2 Now, we will find all path matrices of length 2. For finding such a Matrix, we will multiply A^1 x A^1. The output of this Matrix consists of those nodes which can be reached via using one intermediate node.

A2 =

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

X

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

A2 matrix is calculated by multiplying A^1 X A^1.

Figure:- Computing A^2 Matrix

A2 =

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

A2 matrix shows all path having one intermediate node between any two nodes

Figure:- Final A^2 Matrix

Step 3 Now, we will find all such path matrices whose length will be 3. For finding such a Matrix, we will multiply A^2 x A^1. The output of this Matrix consists of those nodes which can be reached via using two intermediate nodes.

A3 =

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

X

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

A3 matrix is calculated by multiplying A^2 X A^1

Figure:- Computing A^3 Matrix

A3 =

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

A3 matrix shows all path having two intermediate nodes between any two nodes

Figure:- Final A^3 Matrix

Step 4 The last step will be to find the path matrix of length 4 in the given Matrix. For finding such a Matrix, we will multiply A^3 x A^1. The output of this Matrix consists of those nodes which can be reached via using three intermediate nodes.

A4 =

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

X

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

A4 matrix is calculated by multiplying A^3 X A^1

Figure:- Computing A^4 Matrix

A4 =

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

A4 matrix shows all path having three intermediate nodes between any two nodes

Figure:- Final A^4 Matrix

Step 5 Calculate Path Matrix

In the above example, to show all the path length in this Matrix, we need to add the entire Matrix we have calculated. The final $A^1 + A^2 + A^3 + A^4$, we get

P = A1+A2+A3+A4

A4 =

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

Figure:- Path Matrix obtained

Step 6 Print Transitive Closures

The final transitive closure can be calculated using the following formula

If a[1][j] = != 0 then put a[i][j] equals to 1 else put a[i][j] equals to 0.

Here, a[1][1] is 1 a non-zero value, so it will be 1 as shown below in the final Matrix.

A4 =

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

If a[i][j] != 0 then a[i][j] = 1

	1	2	3	4
1	1			
2				
3				
4				

A[1][1] != 0 ➡ A[1][1] = 1

Figure:- Printing value of a[1][1]

Here, a[1][2] is 1 a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

If a[i][j] != 0 then a[i][j] = 1

	1	2	3	4
1	1	1		
2				
3				
4				

A[1][2] != 0 ➡ A[1][2] = 1

Figure:- Printing value of a[1][2]

Here, a[1][3] is 1 a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

If a[i][j] != 0 then a[i][j] = 1

	1	2	3	4
1	1	1	1	
2				
3				
4				

A[1][3] != 0 ➡ A[1][3] = 1

Figure:- Printing value of a[1][3]

Here, a[1][4] is 1 a non-zero value, so in final Matrix, it will be 1 as shown below.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2				
3				
4				

If a[i][j] != 0 then a[i][j] = 1

A[1][4] != 0 ➡ A[1][4] = 1

Figure:- Printing value of a[1][4]

Here, a[2][1] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1			
3				
4				

If a[i][j] != 0 then a[i][j] = 1

A[2][1] != 0 ➡ A[2][1] = 1

Figure:- Printing a value of a[2][1]

Here, a[2][2] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1		
3				
4				

If a[i][j] != 0 then a[i][j] = 1

A[2][2] != 0 ➡ A[2][2] = 1

Figure:- Printing a value of a[2][2]

Here, a[2][3] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1	1	
3				
4				

If a[i][j] != 0 then a[i][j] = 1

A[2][3] != 0 ➡ A[2][3] = 1

Figure:- Printing a value of a[2][3]

Here, a[2][4] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3				
4				

If a[i][j] != 0 then a[i][j] = 1 A[2][4] = 0 A[2][4] = 1

Figure:- Printing a value of A[2][4]

Here, A[3][1] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1			
4				

If a[i][j] != 0 then a[i][j] = 1 A[3][1] != 0 A[3][1] = 1

Figure:- Printing a value of A[3][1]

Here, a[3][2] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1		
4				

If a[i][j] != 0 then a[i][j] = 1 A[3][2] != 0 A[3][2] = 1

Figure:- Printing a value of a[3][2]

Here, a[3][3] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1	1	
4				

If a[i][j] != 0 then a[i][j] = 1 A[3][3] != 0 A[3][3] = 1

Figure:- Printing a value of a[3][3]

Here, a[3][4] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4				

If a[i][j] != 0 then a[i][j] = 1 A[3][4] != 0 A[3][4] = 1

Figure:- Printing a value of a[3][4]

Here, a[4][1] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1			

If a[i][j] != 0 then a[i][j] = 1 A[4][1] != 0 A[4][1] = 1

Figure:- Printing a value of a[4][1]

Here, a[4][2] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1		

If a[i][j] != 0 then a[i][j] = 1 A[4][2] != 0 A[4][2] = 1

Figure:- Printing a value of a[4][2]

Here, a[4][3] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	

If a[i][j] != 0 then a[i][j] = 1 A[4][3] != 0 A[4][3] = 1

Figure:- Printing a value of a[4][3]

Here, a[4][4] is a non-zero value, so it will be 1 as shown below in the final Matrix.

	1	2	3	4
1	1	2	1	1
2	1	2	1	2
3	2	3	2	2
4	1	2	2	2

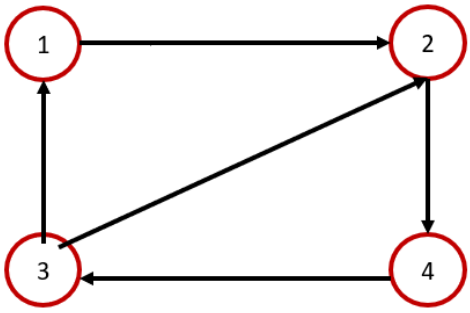
	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1

If a[i][j] != 0 then a[i][j] = 1

A[4][4] != 0 ➡ A[4][4] = 1

Figure:- Printing a value of a[4][4]

The final transitive Closure for the Graph given below is given as



	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1

Figure: Final Transitive Closure of the above Graph

4.5.8. Transitive Closure (Method 2):

A Reachability Matrix represents the transitive Closure of a directed graph. Reachability Matrix here means to have a path to reach from a vertex i to another vertex j in a given Graph which is G = (V, E). The path to reach from Vertex i to j must be between all pairs of Vertices (i, j).

The recursive definition of transitive Closure of a graph $T_{ij}^{(k)}$ is given as below:

For k=0 that is, $T_{ij}^{(0)} =$ $\begin{cases} 0, \text{ if } i \,!= j \text{ and } (i, j) \notin E \\ 1, \text{ if } i = j \text{ or } (i, j) \in E \end{cases}$ Here E = (i, j) that is edge between 2 vertices

For $k >= 1$ $T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee (T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)})$

The Algorithm for the above equation is as follows:

ALGORITHM TransitiveClosure(G)

```
BEGIN:
        n = | V[G] |
        FOR i = 1 to n DO
            FOR j = 1 to n DO
                IF ( (i = j) or ( i, j) ∈ E [G]) THEN
                        Then T(i,j)(0) = 1
                ELSE T(i,j)(0) = 0
        FOR k = 1 to n DO // number of nodes
        FOR i = 1 to n DO // number of row
```

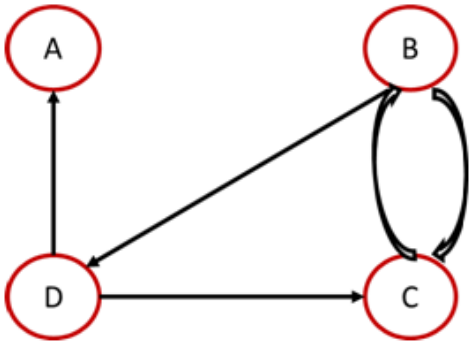
FOR j = 1 to n DO // number of columns

$$T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee (T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)})$$

RETURN T^n

END;

The above algorithm can be explained by a step by step approach for the given below Graph:



$T^{(0)}$ =

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

$T^{(1)}$ =

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

Step 2: If calculated by the formula $T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee (T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)})$
if i= 1 and j= 2 and k = 1 Then $T_{12}^{(1)} = (T_{12}^{(0)} \vee (T_{11}^{(0)} \wedge T_{12}^{(0)})$
That is , = $0 \vee (1 \wedge 0) = 0$,
In the same way using the above formula it can be calculated for all rows and all column values. Also k = 1 indicates that any path reachable between two vertices using intermediate vertex A.

$T^{(2)}$ =

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

Step 3: If calculated by the formula $T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee (T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)})$
if i= 3 and j= 4 and k = 2 Then $T_{34}^{(2)} = (T_{34}^{(1)} \vee (T_{32}^{(1)} \wedge T_{24}^{(1)})$
That is = $0 \vee (1 \wedge 1) = 1$
Also k = 2 indicates that any path reachable between two vertices using intermediate vertex B. we can see that a path from C to B and B to D exists now.

$T^{(3)}$ =

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

Step 4: If calculated by the formula $T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee (T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)})$
if i= 4 and j= 2 and k = 3 Then $T_{42}^{(3)} = (T_{42}^{(2)} \vee (T_{43}^{(2)} \wedge T_{32}^{(1)})$
That is = = $0 \vee (1 \wedge 1) = 1$
Also k = 3 indicates that any path reachable between two vertices using intermediate vertex C. we can see that a path from D to C and C to B exists now.

$T^{(4)}$ =

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

Step 5: If calculated by the formula $T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee (T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)})$
if i= 2 and j= 1 and k = 4 Then $T_{21}^{(4)} = (T_{21}^{(3)} \vee (T_{24}^{(3)} \wedge T_{41}^{(3)})$
That is = = $0 \vee (1 \wedge 1) = 1$
Also k = 4 indicates that any path reachable between two vertices using intermediate vertex D. we can see that a path from B to D and D to A exists now also path from C to B, B to D and D to A exists.

The above $T^{(4)}$ Matrix is the Final Transitive Closure of the above-shown Graph, showing no reachability from Node A to Node B, C, and D after all intermediate steps.

Time Complexity: The time -complexity of the above solution is $\Theta(n^3)$ as three for loops exist.

4.5.9. Multiple Choice Questions:

1.	Find the zero-one Matrix of the transitive Closure of the relation given by the Matrix A? (UGC NET 2019) $A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$
A	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
B	$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$
C	$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$
D	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$
AN	A

2.	What will be the complexity for finding transitive Closure for dense Matrix?
A	$O(V^2)$
B	$O(V^4)$
C	$O(V^3)$
D	None of these
AN	C
3.	What will be the complexity for finding transitive Closure for sparse Matrix?
A	$O(V^2 + E)$
B	$O(V^4 + E)$
C	$O(V^3 + E)$
D	None of these
AN	A
4.	What will be the complexity for finding transitive Closure for sparse Matrix?
A	$O(V^2 + E)$
B	$O(V^4 + E)$
C	$O(V^3 + E)$
D	None of these
AN	A
5.	Which of the following algorithm is used to find transitive Closure?
A	Prims Algorithm
B	Kruskal Algorithm
C	Floyd Warshall Algorithm
D	None of these
AN	C

6.	Find the A^2 Matrix of the transitive Closure of the relation given by the Matrix A? $A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$
A	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
B	$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$

C	$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$
D	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$
AN	A

Coding Question:

In order to reduce the complexity of Transitive Closure, can we implement it with DFS? After implementing, write its complexity?

4.5.10. All Pair Shortest Path Problem: Floyd Warshall

Given a Graph G =(V,E), we have to find the shortest path from every vertex u to every vertex v given in a graph is called All Pair Shortest Path Problem.

Basically, the shortest path from every vertex u to every vertex v can be obtained using the single-source shortest path algorithms by running the loop |V| times, by making each vertex as a source once.

If the weights associated with the edges are positive then, we can simply apply Dijkstra's Algorithm. If the weights associated with the edges are negative then, Dijkstra's algorithm will no longer work. Here, we must run the slower Bellman-Ford Algorithm once from each vertex. The resulting running time is O (V^2E), which on a dense graph will result in O (V^4).

Here, we will see the more optimized approach in comparison to Single source shortest path approaches for computing the shortest path from every vertex u to every vertex v. We will also explore the relation of all pair shortest path problem to matrix multiplication.
As in a Single Source Shortest Path Algorithms, we assume an adjacency list representation of the graph. But in all pair shortest path algorithms, we assume the adjacency matrix representation of the graph.
For a given directed graph G= (V, E), First, we need to prepare the adjacency matrix of a given graph. Weight adjacency matrix of n* n vertices is prepared for representing the edge weights of an n vertex directed graph G= (V, E). Weight adjacency matrix is defined as:

$$W_{ij} = \begin{cases} 0 & \text{If } i = j \\ \text{Weight of the directed edge from i to j} & \text{If not equal to j and (i, j) belongs to E} \\ \Theta & \text{If not equal to j and (i, j) does not belongs to E} \end{cases}$$

Negative-weight edges are allowed, but we assume for the time being that the input graph contains no negative-weight cycles.

Floyd Warshall's Algorithm:

Floyd Warshall's Algorithm is for solving the all-Pairs Shortest Path problem. The problem is to find the shortest distances between every pair of vertices in a given edge-weighted directed graph.

The strategy followed by Floyd Warshall's algorithm is dynamic programming. The resulting algorithm, known as the Floyd-Warshall algorithm, runs in $\Theta(V^3)$ time. As before, negative-weight edges may be present, but we assume that there are no negative weight cycles.

Step 1: Characterizing the Structure of Shortest Path:

In this, we will use different categorizations to structure the shortest path in comparison to matrix multiplication based on all pairs algorithms. The algorithm considers the "intermediate" vertices of the shortest path, where an intermediate vertex of a simple path $p = v_1, v_2,...,v_l$ is any vertex of p other than v_1 or v_l, that is, any vertex in the set $\{v_2, v_3,..., v_{l-1}\}$.

The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2,..., k - 1}. The relationship depends on whether or not k is an intermediate vertex of path p. There will be two cases used to define whether k is an intermediate vertex or not.

Case 1: If K is not an intermediate vertex: If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2, ..., k-1}.

Thus, the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2, ..., k-1} is also the shortest path from i to j with all intermediate vertices in the set {1, 2, ..., k}.

Case 2: If K is an intermediate vertex: If k is an intermediate vertex of path p, then we break path P, from i to j with the help of k. p1 is the shortest path from i to k with all intermediate vertices in the set {1, 2,..., k}. Because vertex k is not an intermediate vertex of path p1, we see that p1 is a shortest path from i to k with all intermediate vertices in the set {1, 2,..., k - 1}. Similarly, p2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set {1, 2, ..., k-1}.

Step 2: A recursive solution to the all-pairs shortest-paths problem:

Let d_{ij}^k be the weight of the shortest path from vertex *i* to vertex *j* for which all intermediate vertices are in the set {1, 2,...,*k*}. When *k* = 0, a path from vertex *i* to vertex *j* with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^0 = w_{ij}$

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Step 3: Computing the shortest path weights in a bottom-up fashion:

In this bottom-up procedure is used to compute the shortest path weights for the increasing values of k. Initially, when k=0, there is no intermediate vertex; simply write the weights associated on the edges, which may be 0, Θ, and weight on the directed edge.
Once k increases from 0, each vertex one by one will behave as an intermediate vertex and update the shortest path weights accordingly.

ALGORITHM FloydWarshall (W[ ][ ])

BEGIN:

1. n = rows [W]
2. $D^0 = W$
3. FOR k = 1 TO n DO
4. FOR i = 1 to n DO
5. FOR j = 1 to n DO
6. $d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7. RETURN $D^{(n)}$

END;

Complexity Analysis of Floyd Warshall Algorithm:

1. n = rows [W] ……O(1)
2. $D^0 = W$ ……O(1)
3. FOR k = 1 TO n DO ……O(n)
4. FOR i = 1 to n DO ……O(n)
5. FOR j = 1 to n DO ……O(n)
6. $d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ ……O(1)
7. RETURN $D^{(n)}$ ……O(1)

Complexity of the Floyd-Warshall algorithm is computed by the three nesting for loops of lines 3-6. Each execution of line 6 takes O (1) time. The algorithm thus runs in time $\theta(n^3)$

Step 4: Constructing the Shortest Path:

There are a variety of different methods for constructing shortest paths in Floyd-Warshall's algorithm. One way is to compute the matrix D of shortest-path weights and then construct the predecessor matrix Π from the D matrix. This method can be implemented to run in O (n^3) time.

Given the predecessor matrix Π, the PRINT-ALL-PAIRS SHORTEST-PATH procedure can be used to print the vertices on a given shortest path.

We can compute the predecessor matrix Π "on-line" just as the Floyd-Warshall algorithm computes the matrices $D^{(k)}$. Specifically, we compute a sequence of matrices Π(0), Π(1),..., Π(n), where Π = Π(n) and Π_{ij}^K is defined to be the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set {1, 2,..., k}.

Recursive Formulation:

Case 1: when K =0, there is no intermediate vertex:

$\Pi_{ij}^{(k)}$= Nil If i=j or $W_{ij} = \Theta$

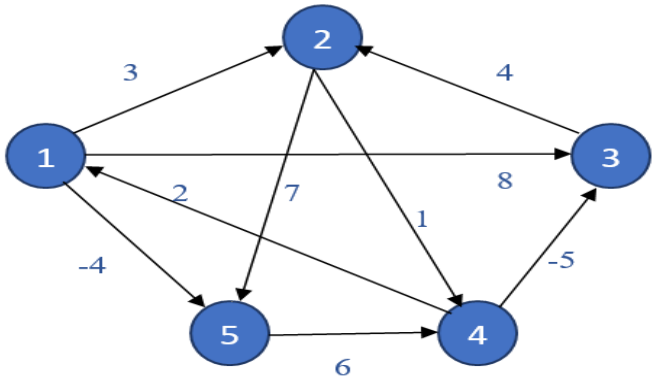
i If i not equal to j and $W_{ij} = \Theta$

Case 2: When K is greater than equal to 1, that is, there exists an intermediate vertex.

$\Pi_{ij}^{(K)} = \Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)} <= d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

$\Pi_{kj}^{(K-1)}$If $d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

Example 1: Apply Floyd Warshall's Algorithm on the given directed graph. Compute the shortest path weights D^K as well as Π^Kpredecessor matrix for the given graph.



Solution 1: First, we will represent the given graph in the form of a matrix. Compute D^0 and Π^0 using the recursive formulation to computing the shortest path weights and predecessor node.

Start by Calculating D Matrix: From the given directed graph, First of all, we will prepare the weighted adjacency matrix highlighting no intermediate vertex between the vertex i and j. Simply assign weights to the matrix under these three categories: a) If there is a directed edge from vertex i to j, then simply assign the weights associated with that particular edge. b) If there is no directed edge from vertex i to j, then simply assign the largest value that is ∞. c) If there is a self-loop, then simply assign 0.

Preparing π^0 from D^0: Without Intermediate vertex: There will be two cases:

Assign NIL, where there is no directed edge as well as a self-loop. That is, we simply assign NIL to all the places in π^0matrix where, in D^0 matrix there is∞ and 0.

Assign row value i, where there is a directed edge from vertex i to vertex j. Here we simply return i, as i is the predecessor of vertex j.

D^0	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	–5	0	∞
5	∞	∞	∞	6	0

π^0	1	2	3	4	5
1	NIL	1	1	NIL	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	NIL	NIL
4	4	NIL	4	NIL	NIL
5	NIL	NIL	NIL	5	NIL

Step 1: When K=1 (Node '1' is intermediate node): To calculate shortest path weights:

$d_{ij}^{(k)} \leftarrow \min (d_{ij}^{(k-1)}, dik^{(k-1)} + d_{kj}^{(k-1)})$

$d_{12}^{(1)} = \min(d_{12}^{(0)}, d_{11}^{(0)} + d_{12}^{(0)})$

$d_{12}^{(1)} = \min (3, 0+3) = 3$

$d_{13}^{(1)} = \min(d_{13}^{(0)}, d_{11}^{(0)} + d_{13}^{(0)})$

$d_{13}^{(1)} = \min (8, 0+8) = 8$

$d_{14}^{(1)} = \min(d_{14}^{(0)}, d_{11}^{(0)} + d_{14}^{(0)})$

$d_{15}^{(1)} = \min(d_{15}^{(0)}, d_{11}^{(0)} + d_{15}^{(0)})$

$d_{14}^{(1)} = \min(\infty, 0+\infty) = \infty$

$d_{15}^{(1)} = \min(-4, 0+(-4)) = -4$

$d_{21}^{(1)} = \min(d_{21}^{(0)}, d_{21}^{(0)} + d_{11}^{(0)})$
$d_{21}^{(1)} = \min(\infty, \infty + 0) = \infty$

$d_{23}^{(1)} = \min(d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)})$
$d_{23}^{(1)} = \min(\infty, \infty+8) = \infty$

$d_{24}^{(1)} = \min(d_{24}^{(0)}, d_{21}^{(0)} + d_{14}^{(0)})$
$d_{24}^{(1)} = \min(1, \infty+\infty) = 1$

$d_{25}^{(1)} = \min(d_{25}^{(0)}, d_{21}^{(0)} + d_{15}^{(0)})$
$d_{25}^{(1)} = \min(7, \infty+(-4)) = 7$

$d_{31}^{(1)} = \min(d_{31}^{(0)}, d_{31}^{(0)} + d_{11}^{(0)})$
$d_{31}^{(1)} = \min(\infty, \infty + 0) = \infty$

$d_{32}^{(1)} = \min(d_{32}^{(0)}, d_{31}^{(0)} + d_{12}^{(0)})$
$d_{32}^{(1)} = \min(4, \infty+3) = 4$

$d_{34}^{(1)} = \min(d_{34}^{(0)}, d_{31}^{(0)} + d_{14}^{(0)})$
$d_{34}^{(1)} = \min(-5, \infty+\infty) = -5$

$d_{35}^{(1)} = \min(d_{35}^{(0)}, d_{31}^{(0)} + d_{15}^{(0)})$
$d_{35}^{(1)} = \min(\infty, \infty+(-4)) = \infty$

$d_{41}^{(1)} = \min(d_{41}^{(0)}, d_{41}^{(0)} + d_{11}^{(0)})$
$d_{41}^{(1)} = \min(2, 2 + 0) = 2$

$d_{42}^{(1)} = \min(d_{42}^{(0)}, d_{41}^{(0)} + d_{12}^{(0)})$
$d_{42}^{(1)} = \min(\infty, 2 +3) = 5$

$d_{43}^{(1)} = \min(d_{43}^{(0)}, d_{41}^{(0)} + d_{13}^{(0)})$
$d_{43}^{(1)} = \min(-5, 2+8) = -5$

$d_{45}^{(1)} = \min(d_{45}^{(0)}, d_{41}^{(0)} + d_{15}^{(0)})$
$d_{45}^{(1)} = \min(\infty, 2+(-4)) = -2$

$d_{51}^{(1)} = \min(d_{51}^{(0)}, d_{51}^{(0)} + d_{11}^{(0)})$
$d_{51}^{(1)} = \min(\infty, \infty + 0) = \infty$

$d_{52}^{(1)} = \min(d_{52}^{(0)}, d_{51}^{(0)} + d_{12}^{(0)})$
$d_{52}^{(1)} = \min(\infty, \infty +3) = \infty$

$d_{53}^{(1)} = \min(d_{53}^{(0)}, d_{51}^{(0)} + d_{13}^{(0)})$
$d_{53}^{(1)} = \min(\infty, \infty +8) = \infty$

$d_{54}^{(1)} = \min(d_{54}^{(0)}, d_{51}^{(0)} + d_{14}^{(0)})$
$d_{54}^{(1)} = \min(6, \infty+(-4)) = 6$

Preparing π^1 from D^1: 1 as Intermediate vertex: There will be two cases:
Simply copy all the values of π^0 to π^1 where there is no change in D^1 matrix from D^0. Basically, this will satisfy that condition of recursive formulation in which
$\Pi_{ij}^{(K)} = \Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)} <= d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

If there is the changeinD^1 matrix from D^0 then we simply calculate the predecessor node of vertex j with the help of the following condition of recursive formulation:
$\Pi_{ij}^{(K)} = \Pi_{kj}^{(K-1)}$ If $d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

D^1	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	–5	0	–2
5	∞	∞	∞	6	0

π^1	1	2	3	4	5
1	NIL	1	1	NIL	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	NIL	NIL
4	4	1	4	NIL	1
5	NIL	NIL	NIL	5	NIL

Step 2: When K=2 (Node '2' is intermediate node)

$d_{ij}^{(k)} \leftarrow \min (d_{ij}^{(k-1)}, dik^{(k-1)} + d_{kj}^{(k-1)})$

$d_{12}^{(2)} = \min(d_{12}^{(1)} , d_{12}^{(1)} + d_{21}^{(1)})$

$d_{12}^{(2)} = \min (3, 3+ \infty) = 3$

$d_{13}^{(2)} = \min(d_{13}^{(1)} , d_{12}^{(1)} + d_{23}^{(1)})$

$d_{13}^{(2)} = \min (8, 2+\infty) = 8$

$d_{14}^{(2)} = \min(d_{14}^{(1)} , d_{12}^{(1)} + d_{24}^{(1)})$

$d_{14}^{(2)} = \min (\infty, 3+1) = 4$

$d_{15}^{(2)} = \min(d_{15}^{(1)} , d_{12}^{(1)} + d_{25}^{(1)})$

$d_{15}^{(2)} = \min (-4, 3+7) = -4$

$d_{21}^{(2)} = \min(d_{21}^{(1)} , d_{22}^{(1)} + d_{21}^{(1)})$

$d_{21}^{(2)} = \min (\infty, 0+ \infty) = \infty$

$d_{23}^{(2)} = \min(d_{23}^{(1)} , d_{22}^{(1)} + d_{23}^{(1)})$

$d_{23}^{(2)} = \min (\infty, 0+\infty) = \infty$

$d_{24}^{(2)} = \min(d_{24}^{(1)} , d_{22}^{(1)} + d_{24}^{(1)})$

$d_{24}^{(2)} = \min (1, 0+1) = 1$

$d_{25}^{(2)} = \min(d_{25}^{(1)} , d_{22}^{(1)} + d_{25}^{(1)})$

$d_{25}^{(2)} = \min (7 , 0+7) = 7$

$d_{31}^{(2)} = \min(d_{31}^{(1)} , d_{32}^{(1)} + d_{21}^{(1)})$

$d_{31}^{(2)} = \min (\infty, 4+ \infty) = \infty$

$d_{32}^{(2)} = \min(d_{32}^{(1)} , d_{32}^{(1)} + d_{22}^{(1)})$

$d_{32}^{(2)} = \min (4, 4+0) = 4$

$d_{34}^{(2)} = \min(d_{34}^{(1)} , d_{32}^{(1)} + d_{24}^{(1)})$

$d_{34}^{(2)} = \min (\infty, 4+1) = 5$

$d_{35}^{(2)} = \min(d_{35}^{(1)} , d_{32}^{(1)} + d_{25}^{(1)})$

$d_{35}^{(2)} = \min (\infty , 4+7) = 11$

$d_{41}^{(2)} = \min(d_{41}^{(1)} , d_{42}^{(1)} + d_{21}^{(1)})$

$d_{41}^{(2)} = \min (2, 5+ \infty) = 2$

$d_{42}^{(2)} = \min(d_{42}^{(1)} , d_{42}^{(1)} + d_{22}^{(1)})$

$d_{42}^{(2)} = \min (5, 5+0) = 5$

$d_{43}^{(2)} = \min(d_{43}^{(1)} , d_{42}^{(1)} + d_{23}^{(1)})$

$d_{43}^{(2)} = \min (10, 5+\infty) = 10$

$d_{45}^{(2)} = \min(d_{45}^{(1)} , d_{42}^{(1)} + d_{25}^{(1)})$

$d_{45}^{(2)} = \min (-2 , 5+7) = -2$

$d_{51}^{(2)} = \min(d_{51}^{(1)} , d_{52}^{(1)} + d_{21}^{(1)})$

$d_{51}^{(2)} = \min (\infty, \infty+ \infty) = \infty$

$d_{52}^{(2)} = \min(d_{52}^{(1)} , d_{52}^{(1)} + d_{22}^{(1)})$

$d_{52}^{(2)} = \min (\infty, \infty+0) = \infty$

$d_{53}^{(2)} = \min(d_{53}^{(1)} , d_{52}^{(1)} + d_{23}^{(1)})$

$d_{53}^{(2)} = \min (\infty, \infty+\infty) = \infty$

$d_{54}^{(2)} = \min(d_{54}^{(1)} , d_{52}^{(1)} + d_{24}^{(1)})$

$d_{54}^{(2)} = \min (6 , \infty+1) = 6$

Preparing π^2 from D^2: 2 as Intermediate vertex: There will be two cases:

Simply copy all the values of $\pi^1$to $\pi^2$where there is no change in D^2 matrix from D^1. Basically, this will satisfy that condition of recursive formulation in which

$\Pi_{ij}^{(K)} = \Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)} <= d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

If there is the changeinD^2 matrix from D^1 then we simply calculate the predecessor node of vertex j with the help of following condition of recursive formulation:

$\Pi_{ij}^{(K)} = \Pi_{kj}^{(K-1)}$ If $d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

D^2	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	–5	0	–2
5	∞	∞	∞	6	0

π^2	1	2	3	4	5
1	NIL	1	1	2	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	2	2
4	4	1	4	NIL	1
5	NIL	NIL	NIL	5	NIL

Step 3: When K=3 (Node '3' is intermediate node)

$d_{ij}^{(k)} \leftarrow \min (d_{ij}^{(k-1)}, dik^{(k-1)} + d_{kj}^{(k-1)})$

$d_{12}^{(3)} = \min(d_{12}^{(2)} , d_{13}^{(2)} + d_{32}^{(2)})$

$d_{12}^{(3)} = \min (3, 8+ 4) = 3$

$d_{13}^{(3)} = \min(d_{13}^{(2)} , d_{13}^{(2)} + d_{33}^{(2)})$

$d_{13}^{(3)} = \min (8, 8+0) = 8$

$d_{14}^{(3)} = \min(d_{14}^{(2)} , d_{13}^{(2)} + d_{34}^{(2)})$

$d_{14}^{(3)} = \min (4, 8+11) = 4$

$d_{15}^{(3)} = \min(d_{15}^{(2)} , d_{13}^{(2)} + d_{35}^{(2)})$

$d_{15}^{(3)} = \min (-4, 8+11) = -4$

$d_{21}^{(3)} = \min(d_{21}^{(2)} , d_{23}^{(2)} + d_{31}^{(2)})$

$d_{21}^{(3)} = \min (\infty, \infty+ \infty) = \infty$

$d_{23}^{(3)} = \min(d_{23}^{(2)} , d_{23}^{(2)} + d_{33}^{(2)})$

$d_{23}^{(3)} = \min (\infty, \infty+0) = \infty$

$d_{24}^{(3)} = \min(d_{24}^{(2)} , d_{23}^{(2)} + d_{34}^{(2)})$

$d_{24}^{(3)} = \min (1, \infty+(-5)) = 1$

$d_{25}^{(3)} = \min(d_{25}^{(2)} , d_{23}^{(2)} + d_{35}^{(2)})$

$d_{25}^{(3)} = \min (7, \infty+11) = 7$

$d_{31}^{(3)} = \min(d_{31}^{(2)} , d_{33}^{(2)} + d_{31}^{(2)})$

$d_{31}^{(3)} = \min (\infty, 0+ \infty) = \infty$

$d_{32}^{(3)} = \min(d_{32}^{(2)} , d_{33}^{(2)} + d_{32}^{(2)})$

$d_{32}^{(3)} = \min (4, 0+4) = 4$

$d_{34}^{(3)} = \min(d_{34}^{(2)} , d_{33}^{(2)} + d_{34}^{(2)})$

$d_{34}^{(3)} = \min (-5, 0+(-5)) = -5$

$d_{35}^{(3)} = \min(d_{35}^{(2)} , d_{33}^{(2)} + d_{35}^{(2)})$

$d_{35}^{(3)} = \min (11, 0+11) = 11$

$d_{41}^{(3)} = \min(d_{41}^{(2)} , d_{43}^{(2)} + d_{31}^{(2)})$

$d_{41}^{(3)} = \min (2, 10+ \infty) = 2$

$d_{42}^{(3)} = \min(d_{42}^{(2)} , d_{43}^{(2)} + d_{32}^{(2)})$

$d_{42}^{(3)} = \min (5, -5 +4) = -1$

$d_{43}^{(3)} = \min(d_{43}^{(2)} , d_{43}^{(2)} + d_{33}^{(2)})$

$d_{43}^{(3)} = \min (10, 10+ 0) = 10$

$d_{45}^{(3)} = \min(d_{45}^{(2)} , d_{43}^{(2)} + d_{35}^{(2)})$

$d_{45}^{(3)} = \min (-2, 10+11) = -2$

$d_{51}^{(3)} = \min(d_{51}^{(2)} , d_{53}^{(2)} + d_{31}^{(2)})$

$d_{51}^{(3)} = \min (\infty, \infty+ \infty) = \infty$

$d_{52}^{(3)} = \min(d_{52}^{(2)} , d_{53}^{(2)} + d_{32}^{(2)})$

$d_{52}^{(3)} = \min (\infty, \infty+4) = \infty$

$d_{53}^{(3)} = \min(d_{53}^{(2)} , d_{53}^{(2)} + d_{33}^{(2)})$

$d_{53}^{(3)} = \min (\infty, \infty+ 0) = \infty$

$d_{54}^{(3)} = \min(d_{54}^{(2)} , d_{53}^{(2)} + d_{34}^{(2)})$

$d_{54}^{(3)} = \min (6, \infty+(-5)) = 6$

Preparing π^3 from D^3: 3 as Intermediate vertex

There will be two cases:

a) Simply copy all the values of π^2 to π^3 where there is no change in D^3 matrix from D^2. Basically, this will satisfy that condition of recursive formulation in which

$\Pi_{ij}^{(K)}=\Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)} <= d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

b) If there is the changeinD^3 matrix from D^2 then we simply calculate the predecessor node of vertex j with the help of following condition of recursive formulation:

$\Pi_{ij}^{(K)}=\Pi_{kj}^{(K-1)}$ If $d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

D^3	1	2	3	4	5
1	0	3	8	***4***	-4
2	∞	0	∞	1	7
3	∞	4	0	***5***	***11***
4	2	***–1***	–5	0	***–2***
5	∞	∞	∞	6	0

π^3	1	2	3	4	5
1	NIL	1	1	2	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	2	2
4	4	3	4	NIL	1
5	NIL	NIL	NIL	5	NIL

Step 4: When K=4 (Node '4' is intermediate node)

$d_{12}^{(4)}= \min(d_{12}^{(3)}, d_{14}^{(3)} + d_{42}^{(3)})$

$d_{12}^{(4)} = \min(3, 3+5) = 3$

$d_{13}^{(4)} = \min(d_{13}^{(3)}, d_{14}^{(3)} + d_{43}^{(3)})$

$d_{13}^{(4)} = \min(8, 4+(-5)) = -1$

$d_{14}^{(4)} = \min(d_{14}^{(3)}, d_{14}^{(3)} + d_{44}^{(3)})$

$d_{14}^{(4)} = \min(3, 3+0) = 3$

$d_{15}^{(4)} = \min(d_{15}^{(3)}, d_{14}^{(3)} + d_{45}^{(3)})$

$d_{15}^{(4)} = \min(-4, 3+(-2)) = -4$

$d_{21}^{(4)} = \min(d_{21}^{(3)}, d_{24}^{(3)} + d_{41}^{(3)})$

$d_{21}^{(4)} = \min(\infty, 1+2) = 3$

$d_{23}^{(4)} = \min(d_{23}^{(3)}, d_{24}^{(3)} + d_{43}^{(3)})$

$d_{23}^{(4)} = \min(\infty, 1+(-5)) = -4$

$d_{24}^{(4)} = \min(d_{24}^{(3)}, d_{24}^{(3)} + d_{44}^{(3)})$

$d_{24}^{(4)} = \min(1, 1+0) = 1$

$d_{25}^{(4)} = \min(d_{25}^{(3)}, d_{24}^{(3)} + d_{45}^{(3)})$

$d_{25}^{(4)} = \min(7, 1+(-2)) = -1$

$d_{31}^{(4)} = \min(d_{31}^{(3)}, d_{34}^{(3)} + d_{41}^{(3)})$

$d_{31}^{(4)} = \min(\infty, 5+2) = 7$

$d_{32}^{(4)} = \min(d_{32}^{(3)}, d_{34}^{(3)} + d_{42}^{(3)})$

$d_{32}^{(4)} = \min(4, -5+5) = 0$

$d_{34}^{(4)} = \min(d_{34}^{(3)}, d_{34}^{(3)} + d_{44}^{(3)})$

$d_{34}^{(4)} = \min(-5, -5+0) = -5$

$d_{35}^{(4)} = \min(d_{35}^{(3)}, d_{34}^{(3)} + d_{45}^{(3)})$

$d_{35}^{(4)} = \min(11, 5+(-2)) = 3$

$d_{41}^{(4)}= \min(d_{41}^{(3)}, d_{44}^{(3)} + d_{41}^{(3)})$

$d_{41}^{(4)} = \min(2, 0+2) = 2$

$d_{42}^{(4)} = \min(d_{42}^{(3)}, d_{44}^{(3)} + d_{42}^{(3)})$

$d_{42}^{(4)} = \min(5, 0+5) = 5$

$d_{43}^{(4)} = \min(d_{43}^{(3)}, d_{44}^{(3)} + d_{43}^{(3)})$

$d_{43}^{(4)} = \min(10, 0+10) = 10$

$d_{45}^{(4)} = \min(d_{45}^{(3)}, d_{44}^{(3)} + d_{45}^{(3)})$

$d_{45}^{(4)} = \min(-2, 0+(-2)) = -2$

$d_{51}^{(4)} = \min(d_{51}^{(3)}, d_{54}^{(3)} + d_{41}^{(3)})$

$d_{51}^{(4)} = \min(\infty, 6+2) = 8$

$d_{52}^{(4)} = \min(d_{52}^{(3)}, d_{54}^{(3)} + d_{42}^{(3)})$

$d_{52}^{(4)} = \min(\infty, 6+(-1) = 5$

$d_{53}^{(4)} = \min(d_{53}^{(3)}, d_{54}^{(3)} + d_{43}^{(3)})$ $\quad$ $d_{54}^{(4)} = \min(d_{54}^{(3)}, d_{54}^{(3)} + d_{44}^{(3)})$

$d_{53}^{(4)} = \min(\infty, 6+(-5)) = 1$ $\quad$ $d_{54}^{(4)} = \min(6, 6+0) = 6$

Preparing π^4 from D^4: 4 as Intermediate vertex

There will be two cases:

a) Simply copy all the values of $\pi^3$to $\pi^4$where there is no change in D^4 matrix from D^3. Basically, this will satisfy that condition of recursive formulation in which

$\Pi_{ij}^{(K)}=\Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)} <= d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

b) If there is the changeinD^4 matrix from D^3 then we simply calculate the predecessor node of vertex j with the help of the following condition of recursive formulation:

$\Pi_{ij}^{(K)}=\Pi_{kj}^{(K-1)}$ If $d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

D^4	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

π^4	1	2	3	4	5
1	NIL	1	4	2	1
2	4	NIL	4	2	1
3	4	3	NIL	2	1
4	4	3	4	NIL	1
5	4	3	4	5	NIL

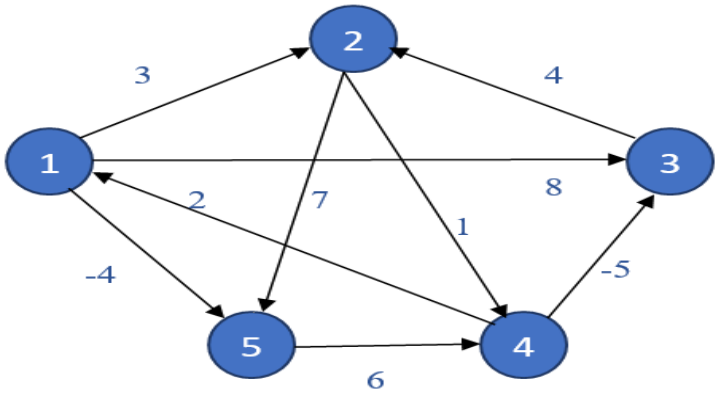
Step 5: When K=5 (Node '5' is an intermediate node)

Similarly, by applying the values, we get $D^{(5)}$.

D^5	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

π^5	1	2	3	4	5
1	NIL	3	4	5	1
2	4	NIL	4	2	1
3	4	3	NIL	2	1
4	4	3	4	NIL	1
5	4	3	4	5	NIL

4.5.11. Floyd Warshall's Algorithm: Method 2



Step 1: Start by Calculating D Matrix: From the given directed graph, First of all, we will prepare the weighted adjacency matrix highlighting no intermediate vertex between the vertex i and j. Simply assign weights to the matrix under these three categories: a) If there is a directed edge from vertex i to j, then simply assign the weights associated with that particular edge. b) If there is no directed edge from vertex i to j, then simply assign the largest value that is ∞. c) If there is a self-loop, then simply assign 0.

D^0	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

Step 2: Preparing D^1 from D^0 Matrix: (1 is behaving as an Intermediate from vertex i to j)

In this, category there may be four possible cases:

Here, Intermediate Vertex 1 is either behaving as source and destination vertex at the following positions such as (1,1)(1,2)(1,3)(1,4)(1,5)(2,1)(3,1)(4,1)(5,1). Hence, we simply copy all the values of D^0 in D^1 at these positions. Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value) from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied row contains any value as ∞, then simply copy the corresponding column from D^0 to D^1. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{kj}contains values as ∞.Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value)from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied column contains any value as ∞, then simply copy the corresponding row from D^0 to D^1. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{ij}contains values as ∞.Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value)from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

Simply compute remaining values from the recursive formulation: $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

D^1	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	***5***	–5	0	***–2***
5	∞	∞	∞	6	0

Step 3: Preparing D^2 from D^1 Matrix: (2 is behaving as an Intermediate from vertex i to j):

In this, category there may be four possible cases:

Here, Intermediate Vertex 2 is either behaving as source and destination vertex at the following positions such as (2,1)(2,2)(2,3)(2,4)(2,5)(1,2)(3,2)(4,2)(5,2). Hence, we simply copy all the values of D^1 in D^2 at these positions. Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value)from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied row contains any value as ∞, then simply copy the corresponding column from D^1 to D^2. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{kj} contains values as ∞.Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value)from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied column contains any value as ∞, then simply copy the corresponding row from D^1 to D^2. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{ij}contains values as ∞.Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value)from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

Simply compute remaining values from the recursive formulation: $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

D^2	1	2	3	4	5
1	0	3	8	***4***	-4
2	∞	0	∞	1	7
3	∞	4	0	***5***	***11***
4	2	***5***	–5	0	***–2***
5	∞	∞	∞	6	0

Step 4: Preparing D^3 from D^2 Matrix: (3 are behaving as an Intermediate from vertex i to j):

In this, category there may be four possible cases:

Here, Intermediate Vertex 3 is either behaving as the source and destination vertex at the following positions such as (3,1)(3,2)(3,3)(3,4)(3,5)(1,3)(2,3)(4,3)(5,3). Hence, we simply copy all the values of D^3 in D^2 at these positions. Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value)from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied row contains any value as ∞, then simply copy the corresponding column from D^3 to D^2. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{kj} contains values as ∞.Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$ (minimum value) from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied column contains any value as ∞, then simply copy the corresponding row from D^3 to D^2. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{ij} contains values as ∞.Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value)from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

Simply compute remaining values from the recursive formulation: $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

D^3	1	2	3	4	5
1	0	3	8	***4***	-4
2	∞	0	∞	1	7
3	∞	4	0	***5***	11
4	2	***–1***	–5	0	***–2***
5	∞	∞	∞	6	0

Step 5: Preparing D^4 from D^3 Matrix: (4 are behaving as an Intermediate from vertex i to j):

In this, category there may be four possible cases:

Here, Intermediate Vertex 4 is either behaving as a source and destination vertex at the following positions such as (4,1)(4,2)(4,3)(4,4)(4,5)(1,4)(2,4)(3,4)(5,4). Hence, we simply copy all the values of D^4 in D^3 at these positions. Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value) from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied row contains any value as ∞, then simply copy the corresponding column from D^4 to D^3. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{kj}contains values as ∞.Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value)from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied column contains any value as ∞, then simply copy the corresponding row from D^4 to D^3. This is because of the recursive formulation derived to compute the shortest path weights

between any vertices with the help of the Intermediate vertex. Here, d_{ij}contains values as ∞.Here, we simply have $d_{ij}^{k}=d_{ij}^{k-1}$(minimum value) from the recursive formulation $d_{ij}^{k} = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

Simply compute remaining values from the recursive formulation:$d_{ij}^{k} = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

D^4	1	2	3	4	5
1	0	3	***-1***	***4***	-4
2	***3***	0	***–4***	1	***-1***
3	***7***	4	0	***5***	***3***
4	2	***–1***	–5	0	***–2***
5	***8***	***5***	***1***	6	0

Step 6: Preparing D^5 from D^4 Matrix: (5 are behaving as an Intermediate from vertex i to j):

In this, category there may be four possible cases:

Here, Intermediate Vertex 5 is either behaving as a source and destination vertex at the following positions such as (5,1)(5,2)(5,3)(5,4)(5,5)(1,5)(2,5)(3,5)(4,5). Hence, we simply copy all the values of D^5 in D^4 at these positions. Here, we simply have $d_{ij}^{k}=d_{ij}^{k-1}$(minimum value) from the recursive formulation $d_{ij}^{k} = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied row contains any value as ∞, then simply copy the corresponding column from D^5 to D^4. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{kj} contains values as ∞.Here, we simply have $d_{ij}^{k}=d_{ij}^{k-1}$ (minimum value) from the recursive formulation $d_{ij}^{k} = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied column contains any value as ∞, then simply copy the corresponding row from D^5 to D^4. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{ij} contains values as ∞.Here, we simply have $d_{ij}^{k}=d_{ij}^{k-1}$(minimum value) from the recursive formulation $d_{ij}^{k} = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

Simply compute remaining values from the recursive formulation: $d_{ij}^{k} = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

D^5	1	2	3	4	5
1	0	***1***	***-3***	***2***	-4
2	***3***	0	***–4***	1	***-1***
3	***7***	4	0	***5***	***3***
4	2	***–1***	–5	0	***–2***
5	***8***	***5***	***1***	6	0

Now we will calculate the predecessor matrix π from the distance matrix D:

Step 1: Preparing π^0 from D^0: Without Intermediate vertex

There will be two cases:

Assign NIL, where there is no directed edge as well as a self-loop. That is we simply assign NIL to all the places in π^0matrix where, in D^0 matrix there is∞ and 0.

Assign row value i, where there is a directed edge from vertex i to vertex j. Here we simply return i, as i is the predecessor of vertex j.

D^0	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	–5	0	∞
5	∞	∞	∞	6	0

π^0	1	2	3	4	5
1	NIL	1	1	NIL	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	NIL	NIL
4	4	NIL	4	NIL	NIL
5	NIL	NIL	NIL	5	NIL

Step 2: Preparing π^1 from D^1: 1 as Intermediate vertex

There will be two cases:

Simply copy all the values of π^0 to π^1 where there is no change in D^1 matrix from D^0. Basically, this will satisfy that condition of recursive formulation in which

$\Pi_{ij}^{(K)}=\Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)}<= d_{ik}^{(k-1)} +d_{kj}^{(k-1)}$)

If there is the changeinD^1 matrix from D^0 then we simply calculate the predecessor node of vertex j with help of following condition of recursive formulation:

$\Pi_{ij}^{(K)}=\Pi_{kj}^{(K-1)}$ If $d_{ij}^{(k-1)}>d_{ik}^{(k-1)} +d_{kj}^{(k-1)}$)

D^1	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	–5	0	–2
5	∞	∞	∞	6	0

π^1	1	2	3	4	5
1	NIL	1	1	NIL	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	NIL	NIL
4	4	1	4	NIL	1
5	NIL	NIL	NIL	5	NIL

Step 3: Preparing π^2 from D^2: 2 as Intermediate vertex

There will be two cases:

a)Simply copy all the values of π^1 to π^2 where there is no change in D^2 matrix from D^1. Basically, this will satisfy that condition of recursive formulation in which

$\Pi_{ij}^{(K)}=\Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)}<= d_{ik}^{(k-1)} +d_{kj}^{(k-1)}$)

b)If there is the changeinD^2 matrix from D^1 then we simply calculate the predecessor node of vertex j with the help of following condition of recursive formulation:

$\Pi_{ij}^{(K)}=\Pi_{kj}^{(K-1)}$ If $d_{ij}^{(k-1)}>d_{ik}^{(k-1)} +d_{kj}^{(k-1)}$)

D^2	1	2	3	4	5
1	0	3	8	***4***	-4
2	∞	0	∞	1	7
3	∞	4	0	***5***	***11***
4	2	***5***	–5	0	***–2***
5	∞	∞	∞	6	0

π^2	1	2	3	4	5
1	NIL	1	1	2	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	2	2
4	4	1	4	NIL	1
5	NIL	NIL	NIL	5	NIL

Step 4: Preparing π^3 from D^3: 3 as Intermediate vertex

There will be two cases:

Simply copy all the values of π^2 to π^3 where there is no change in D^3 matrix from D^2. Basically, this will satisfy that condition of recursive formulation in which

$\Pi_{ij}^{(K)}=\Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)} <= d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

If there is the changeinD^3 matrix from D^2 then we simply calculate the predecessor node of vertex j with help of following condition of recursive formulation:

$\Pi_{ij}^{(K)}=\Pi_{kj}^{(K-1)}$ If $d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

D^3	1	2	3	4	5
1	0	3	8	***4***	-4
2	∞	0	∞	1	7
3	∞	4	0	***5***	***11***
4	2	***–1***	–5	0	***–2***
5	∞	∞	∞	6	0

π^3	1	2	3	4	5
1	NIL	1	1	2	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	2	2
4	4	3	4	NIL	1
5	NIL	NIL	NIL	5	NIL

Step 5: Preparing π^4 from D^4: 4 as Intermediate vertex: There will be two cases:

Simply copy all the values of π^3 to π^4 where there is no change in D^4 matrix from D^3. Basically, this will satisfy that condition of recursive formulation in which

$\Pi_{ij}^{(K)}=\Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)} <= d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

If there is the changeinD^4 matrix from D^3 then we simply calculate the predecessor node of vertex j with help of following condition of recursive formulation:

$\Pi_{ij}^{(K)}=\Pi_{kj}^{(K-1)}$ If $d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

D^4	1	2	3	4	5
1	0	3	***-1***	***4***	-4
2	***3***	0	***–4***	1	***-1***
3	***7***	4	0	***5***	***3***
4	2	***–1***	–5	0	***–2***
5	***8***	***5***	***1***	6	0

π^4	1	2	3	4	5
1	NIL	1	4	2	1
2	4	NIL	4	2	1
3	4	3	NIL	2	1
4	4	3	4	NIL	1
5	4	3	4	5	NIL

Step 6: Preparing π^5 from D^5: 5 as Intermediate vertex: There will be two cases:

Simply copy all the values of π^4 to π^5 where there is no change in D^5 matrix from D^4. Basically, this will satisfy that condition of recursive formulation in which

$\Pi_{ij}^{(K)}=\Pi_{ij}^{(K-1)}$ If $d_{ij}^{(k-1)} <= d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$)

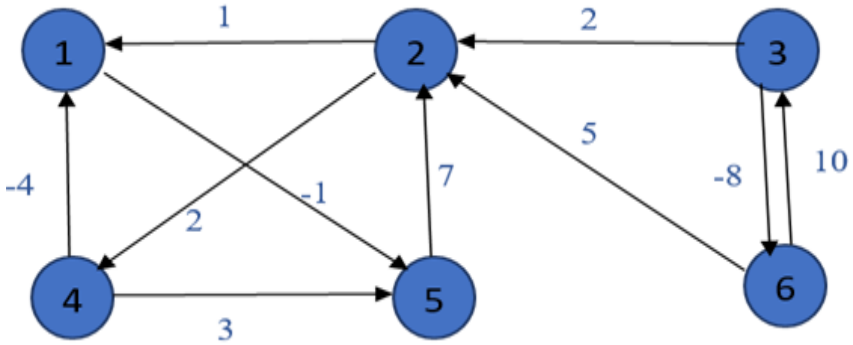
If there is the changeinD^5 matrix from D^4 then we simply calculate the predecessor node of vertex j with help of following condition of recursive formulation:

$\Pi_{ij}^{(K)}=\Pi_{kj}^{(K-1)}$ If $d_{ij}^{(k-1)}>d_{ik}^{(k-1)}+d_{kj}^{(k-1)}$)

D^5	1	2	3	4	5
1	0	***1***	***-3***	***2***	-4
2	***3***	0	***–4***	1	***-1***
3	***7***	4	0	***5***	***3***
4	2	***–1***	–5	0	***–2***
5	***8***	***5***	***1***	6	0

π^5	1	2	3	4	5
1	NIL	3	4	5	1
2	4	NIL	4	2	1
3	4	3	NIL	2	1
4	4	3	4	NIL	1
5	4	3	4	5	NIL

Example 2: Apply Floyd Warshall's Algorithm to compute the shortest path. Compute D^n Matrix



Solution 2: Step 1: Start by Calculating D Matrix: From the given directed graph, First of all, we will prepare the weighted adjacency matrix highlighting no intermediate vertex between the vertex i and j. Simply assign weights to the matrix under these three categories: a) If there is a directed edge from vertex i to j, then simply assign the weights associated with that particular edge. b) If there is no directed edge from vertex i to j, then simply assign the largest value, that is ∞. c) If there is a self-loop, then simply assign 0.

D^0	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	∞	∞
3	∞	2	0	∞	∞	–8
4	-4	∞	∞	0	3	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

Step 2: Preparing D^1 from D^0 Matrix: (1 is behaving as an Intermediate from vertex i to j):

In this, category there may be four possible cases:

Here, Intermediate Vertex 1 is either behaving as source and destination vertex at the following positions such as (1,1)(1,2)(1,3)(1,4)(1,5)(1,6)(2,1)(3,1)(4,1)(5,1)(6,1). Hence, we simply copy all the values of D^0 in D^1 at these positions. Here, we simply have $d_{ij}^k=d_{ij}^{k-1}$(minimum value) from the recursive formulation $d_{ij}^k = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied row contains any value as ∞, then simply copy the corresponding column from D^0 to D^1. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{kj} contains values as ∞.Here,

we simply have $d_{ij}^{k}=d_{ij}^{k-1}$(minimum value) from the recursive formulation $d_{ij}^{k} = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

If the copied column contains any value as ∞, then simply copy the corresponding row from D^0 to D^1. This is because of the recursive formulation derived to compute the shortest path weights between any vertices with the help of the Intermediate vertex. Here, d_{ij} contains values as ∞.Here, we simply have $d_{ij}^{k}=d_{ij}^{k-1}$(minimum value) from the recursive formulation $d_{ij}^{k} = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

Simply compute remaining values from the recursive formulation: $d_{ij}^{k} = \min (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

D^1	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	∞	∞
3	∞	2	0	∞	∞	–8
4	-4	∞	∞	0	–5	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

Step 3: Keep on repeating the same process until we traversed the entire Intermediate vertices:

D^2	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	–8
4	-4	∞	∞	0	–5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

D^3	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	–8
4	-4	∞	∞	0	–5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

D^4	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	–2	0	∞	2	–3	∞
3	3	2	0	4	–1	–8
4	-4	∞	∞	0	–5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

D^5	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	–2	0	∞	2	–3	∞
3	0	2	0	4	–1	–8
4	-4	2	∞	0	–5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

D^6	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	–2	0	∞	2	–3	∞
3	–5	-3	0	–1	–6	–8
4	-4	2	∞	0	–5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

Floyd Warshall's Algorithm Animated Links:

https://www.youtube.com/watch?v=n1iP2DtgOCw

https://www.youtube.com/watch?v=4OQeCuLYj-4

https://www.youtube.com/watch?v=VoNMulNisiU

4.5.12. Objective Type Questions: Company and GATE/PSUs:

1	Floyd Warshall's Algorithm is used for solving
A	All Pair Shortest Path
B	Single-Source Shortest Path
C	Network Flow Problem
D	Sorting Problems
ANS	A

2	Floyd Warshall's Algorithm can be applied on

A	Undirected and Un-weighted Graphs
B	Undirected Graphs
C	Directed Graphs
D	Acyclic Graphs
ANS	C

3	What is the running time of the Floyd Warshall's Algorithm?
A	Bigot(V)
B	Theta(V^2)
C	Bigot(VE)
D	Theta(V^3)
ANS	D

4	What approach is being followed in Floyd Warshall's Algorithm?
A	Greedy
B	Dynamic
C	Backtracking
D	Linear Programming
ANS	B

5	Floyd Warshall's Algorithm can be used for finding
A	Single-Source Shortest Path
B	Transitive Closure
C	Minimum Spanning tree
D	Topological Sort
ANS	B

6	What procedure is being followed in Floyd Warshall's Algorithm?
A	Top Down
B	Bottom-Up
C	Big Bang
D	Sandwich
ANS	B

7	Floyd Warshall's algorithm was proposed by
A	Robert Floyd and Stephen Marshall
B	Stephen Floyd and Robert Marshall
C	Bernad Floyd and Robert Marshall
D	Robert Floyd and BernadWarshall

ANS	A

8	Who proposed the modern formulation of Floyd-Warshall's Algorithm as three nested loops?
A	Robert Floyd
B	Stephen Marshall
C	Bernad Floyd
D	Peter Ingerman
ANS	D

9	Complete the program. n=rows [W] D (0) =W for k=1 to n do for i=1 to n do for j=1 to n ---------- do..... return D(n)
A	dij(k)=min(dij(k-1), dik(k-1) - dkj(k-1))
B	dij(k)=max(dij(k-1), dik(k-1) - dkj(k-1))
C	dij(k)=min(dij(k-1), dik(k-1) + dkj(k-1))
D	dij(k)=max(dij(k-1), dik(k-1) + dkj(k-1))
ANS	C

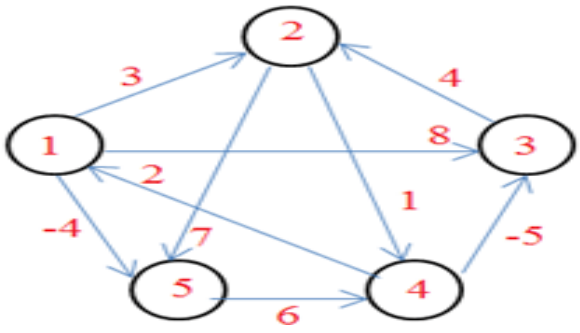
10	What happens when the value of k is 0 in the Floyd Warshall Algorithm?
A	1 Intermediate vertex
B	0 Intermediate vertex
C	N Intermediate vertex
D	N-1 Intermediate vertex
ANS	B

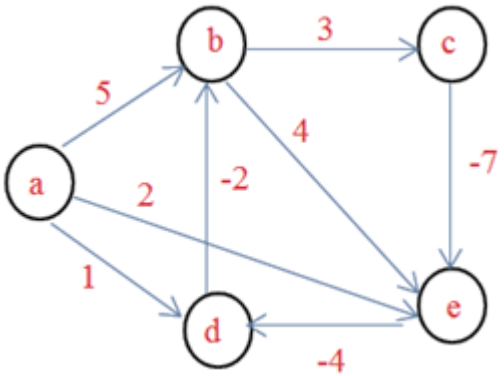
11	Using logical operator's instead of arithmetic operators saves time and space.
A	True
B	False
ANS	A

12	The time taken to compute the transitive closure of a graph is Theta (n^2).

A	True
B	False
ANS	B

13	What is the formula to compute the transitive closure of a graph?
A	tij(k) = tij(k-1) AND (tik(k-1) OR tkj(k-1))
B	tij(k) = tij(k-1) OR (tik(k-1) AND tkj(k-1))
C	tij(k) = tij(k-1) AND (tik(k-1) AND tkj(k-1))
D	tij(k) = tij(k-1) OR (tik(k-1) OR tkj(k-1))
ANS	B

14	In the given graph, what is the minimum cost to travel from vertex 1 to vertex 3? 
A	3
B	2
C	10
D	-3
ANS	D

15	In the given graph, how many intermediate vertices are required to travel from node a to node eat a minimum cost? 

A	2
B	0
C	1
D	3
ANS	C

4.5.13. All Pairs Shortest Paths: Johnson's Algorithm for Sparse Graph:

Johnson's Algorithm: The Problem is to find the shortest path between all pair of vertices in a weighted directed graph. Graph contains negative weights associated with the edges but does not contain the negative cycle.

In this context, we have discussed the Floyd warshall's algorithm that's running time complexity for computing the shortest path weights between all pair of vertices is O (V^3).
For further optimization of running time complexity, a Johnson Algorithm is being proposed which works as a combination of Dijkstra and bellman Ford Algorithm. Johnson Algorithm finds the shortest path weights among all the pairs of vertices in O (V^2logV+ VE).

As we are aware that the Dijkstra algorithm is a single source shortest path algorithm that is, this compute the shortest path from a single source to all other vertices. For computing the shortest path from single source to all other vertices it takes O (V log V) time.

But here, we have to find the shortest path among all pair of vertices. Therefore, we take each vertex of the graph as a source one by one and try to compute the shortest path from that source vertex to all other vertex of the graph.

Then, by applying Dijkstra algorithm for V vertices, this will return the running time complexity as O (V^2logV). Dijkstra Algorithm seems to be the better alternative to Floyd warshall's algorithm for computing the shortest path among all pair of vertices.

But the biggest drawback with the Dijkstra algorithm is that it will not work on the negative weight edges. In order to overcome this problem of negative weight edges we will use bellman ford algorithm.
Hence, the basic objective of Johnson Algorithm is to first remove the negative weights from the graph by re-weighting them using the bellman ford algorithm and once all the edges of the graph contains positive weights then we will simply apply the Dijkstra Algorithm to find the shortest path between all pair of vertices. In this way, we propose the optimized solution in comparison to the

Floyd warshall's algorithm.
Conversion of a given graph to graph containing non negative weights:

Basic objective of this step is to remove negative weights from the graph in order to apply Dijkstra Algorithm. A simple approach for this conversion is to find the minimum weight edge from the graph and simply add this weight to all the edges of the graph.
But this will not work always as there may be different number of edges in different paths. If there are multiple paths from vertex u to vertex v then all paths must be increased by same value so that shortest path remains the shortest in the converted graph.
The basic idea behind the Johnson's Algorithm is to assign a weight to every vertex. Let the weight assigned to vertex u is h[u]. Now we will reweight the edges using the weights associated with each vertex.

For example: For a given edge (u, v) of weight w (u, v), the new weight will be w (u, v) + h[u] – h[v]. Most important thing with this approach is that all set of paths between any pair of vertices is increased by the same value as well as all negative weights will become the non-negative weights. Let us consider any path between two vertices s and t, weight of every path is increased by h[s]-h[t], all h values of vertices on path from s to t cancel each other.

Now a question arises how to compute the h[] values. For this purpose the bellman ford algorithm will be used. In this, a new vertex is added to the graph and connected to all the vertices of the graph. The shortest path weights from a new vertex to all other existing vertices are the h [] values.
Johnson's Step by Step for computing shortest path:

Step 1: Given a graph G (V, E), add a new vertex s to the graph. Add edges from a new vertex s to all other vertices of the Graph G. Then, Obtain a modified Graph G' from the Graph G.
Step 2: Apply bellman ford algorithm on Graph G' with s as a source. Then calculate the distances using bellman ford algorithm for all the vertices in the graph as h [0], h [1], h [2] …….h [V-1]. If we find the negative weight cycle then simply return.
Step 3: Now, again reweight all the edges of the original graph. For each edge (u, v) assign the new weights as the original weight +h[u]-h[v]
Step 4: Now simply remove the added vertex s and apply Dijkstra Algorithm for every vertex.
Johnson's Algorithm:
Given a weighted directed graph G= (V, E) with the weights associated with all the edges and let h is the value associated on the vertices of the graph.
For each edge (u, v) belongs to E define a formulation as w (u, v) = w (u, v) +h[u] – h[v]
Pseudo code of Johnson algorithm is given, keeping in mind the three things, first reweight the edges, and then apply bellman ford algorithm. Once all the edges contain positive weights then apply the Dijkstra Algorithm

Johnson (G)
1. Compute G' where V [G'] = V [G] ∪ {S} and
E [G'] = E [G] ∪ {(s, v): v ∈ V [G]}

2. IF BELLMAN-FORD (G', w, s) = FALSE
Then "input graph contains a negative weight cycle"
Else For each vertex v ∈ V [G']
Do h (v) ← δ(s, v)
Computed by Bellman-Ford algorithm
For each edge (u, v) ∈ E[G']
Do w (u, v) ← w (u, v) + h (u) - h (v)
For each vertex u ∈ V [G]
Do run DIJKSTRA (G, w, u) to compute δ (u, v) for all v ∈ V [G]
For each vertex v ∈ V [G]
Do d_{uv}← δ (u, v) + h (v) - h (u)
Return D

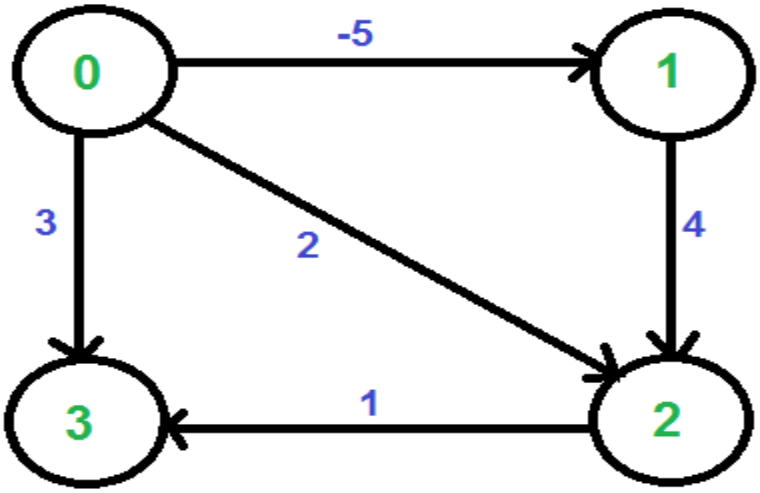
Complexity Analysis of Johnson Algorithm:

Step by Step complexity analysis of Johnson Algorithm is as follows: Line 1 produces new Graph G'. Line 2 executes the bellman ford algorithm on Graph G' with weights w and source vertex s. If Graph G', and hence Graph G, contains a negative-weight cycle, simply reports the problem through Line 3. Lines 4-11 assume that Graph G' contains no negative-weight cycles. Lines 4-5 set h(v) to the shortest-path weight δ(s, v) computed by the Bellman-Ford algorithm for all v. Lines 6-7 compute the new weights in order to remove negative weights from the graph. For each pair of vertices u, v, then for loop of lines 8-11 computes the shortest-path weight by using Dijkstra's algorithm once from each vertex in V. Line 11 is used to again restore the graph from G' to G and hence updating the shortest path weights after applying the Dijkstra's Algorithm. Finally, line 12 returns the completed D matrix. Figure 25.6 shows the execution of Johnson's algorithm.
If the minimum priority queues in Dijsktra algorithm is implemented using the Fibonacci heap then the running time complexity of Johnson's algorithm is O (V^2logV+ VE). This is faster, then the Floyd warshall's algorithm who's running time complexity is O (V^3)

Transformation of Graph G to Graph G': This step is used to ensure the changes that occur in the transformation of Graph G to Graph G'. Our basic objective is to reweight all the negative weight edges in the graph in such a way that all the edges in the graph contain the non-negative weights. Then, further will simply apply the Dijkstra algorithm to compute the shortest path weights among all the pairs of vertices.

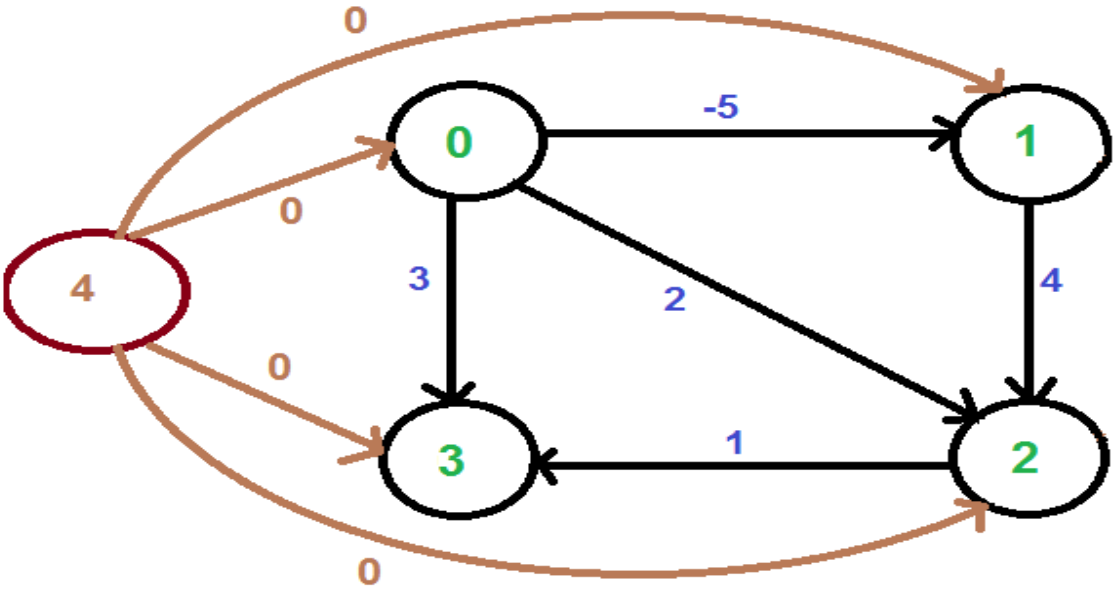
This property will always be true for all the h [] values as they are the shortest distances as h[v] = h[u] + w. (u, v). This simply mean that the shortest distance from vertex s to vertex v must be smaller than or equal to shortest distance from vertex s to vertex u plus weight of the edge(u, v). The new weights are w (u, v) +h[u]-h[v]. Values of the new weights must be greater than or equal to zero.

Example 1: Consider the following graph:

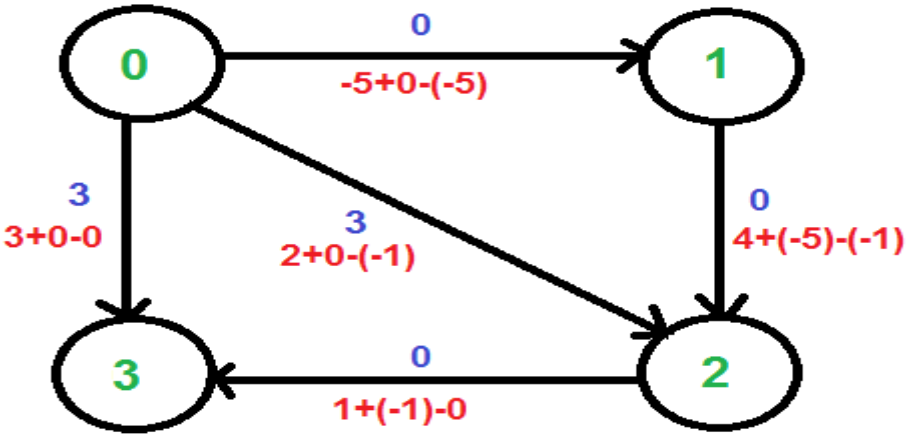


Solution 1:

Step 1: Simply add a source vertex s and add edges from s to all vertices of the original graph. In the following diagram s is 4.



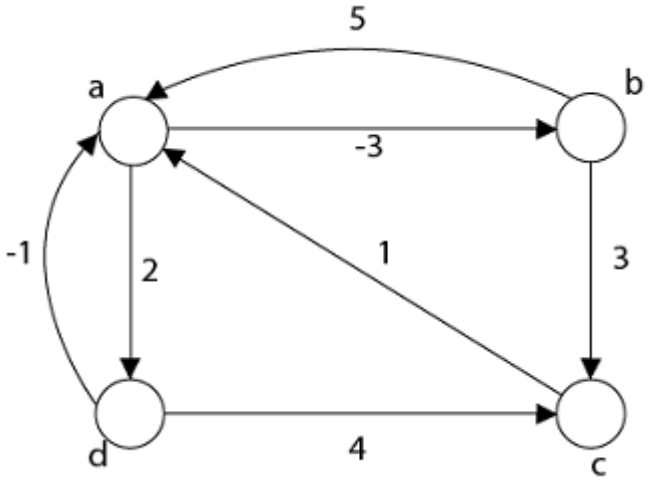
Step 2: Now calculate the shortest path weights from 4 to all other vertices using the Bellman ford Algorithm. Shortest path weights from vertex 4 to vertex 0, vertex 1, vertex 2 and vertex 3 are 0, -5, -1 and 0 respectively, i.e., h [] = {0, -5, -1, 0}. Once we get these shortest path weights, we remove the source vertex 4 and reweight the edges again using following formula. w (u, v) = w (u, v) + h[u] - h[v].



Distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectievely.

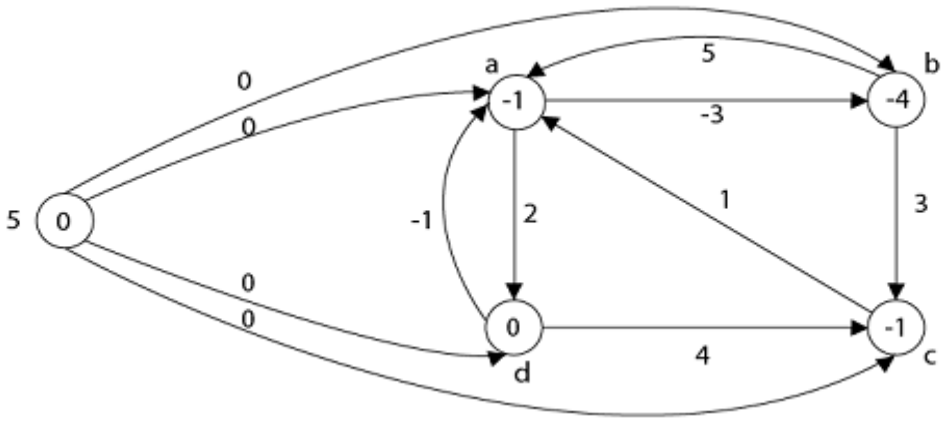
Step 3: Now all the weights in the graph contain non negative weights then we will simply apply the Dijkstra's Algorithm to compute the shortest path from a source vertex to all other vertices in the graph. We will apply the Dijkstra's Algorithm considering all the vertices of the graph as a source once.

Example 2: Consider the following graph. Apply Johnson Algorithm for computing shortest path weights among all pair of vertices.



Solution 2:

Step 1: Consider the new vertex s as a source vertex and make the shortest path weights from s to all other vertices in the graph as 0



Step 2: Apply Bellman ford algorithm for computing the minimum weight on each vertex.
Now, our objective is to re-weight all the negative weight edges of the graph in order to obtain non negative weight edges.

Updating Weight from vertex a to b: w (a, b) = w (a, b) + h (a) - h (b) = -3 + (-1) - (-4) = 0

Updating Weight from vertex b to a: w (b, a) = w (b, a) + h (b) - h (a) = 5 + (-4) - (-1) = 2

Updating Weight from vertex b to c: w (b, c) = w (b, c) + h (b) - h (c) = 3 + (-4) - (-1) = 0

Updating Weight from vertex c to a: w (c, a) = w (c, a) + h (c) - h (a) = 1 + (-1) - (-1) = 1

Updating Weight from vertex d to c: w (d, c) = w (d, c) + h (d) - h (c) = 4 + 0 - (-1) = 5

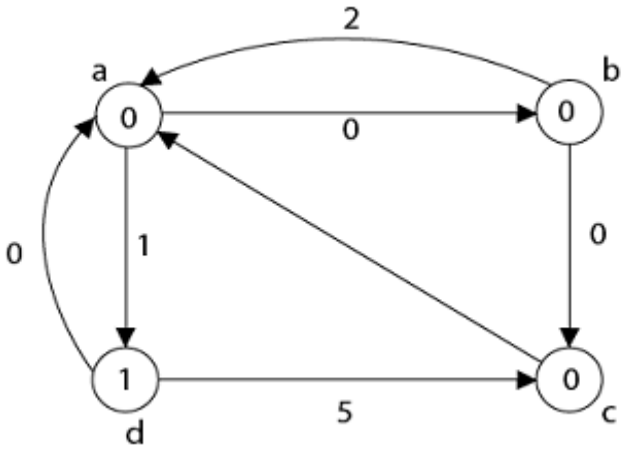
Updating Weight from vertex d to a: w (d, a) = w (d, a) + h (d) - h (a) = -1 + 0 - (-1) = 0

Updating Weight from vertex a to d: w (a, d) = w (a, d) + h (a) - h (d) = 2 + (-1) - 0 = 1

Will further update all the weights in the graph and remove the source vertex s from the graph, means restoring the original graph with non-negative weights.

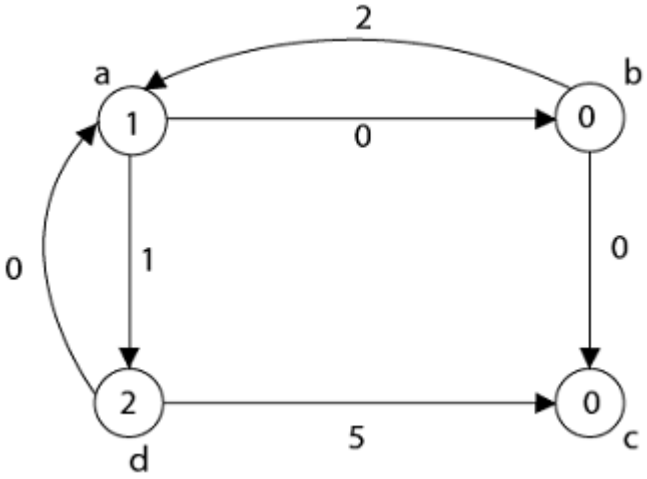
Step 3: All Edge weights are non-negative and now we can further apply Dijkstra Algorithm to compute the shortest path from a single source vertex to all other vertices in the graph. This work will be repeated for considering the entire vertex as a source one by one.

Case 1: Considering 'a' as a source vertex:



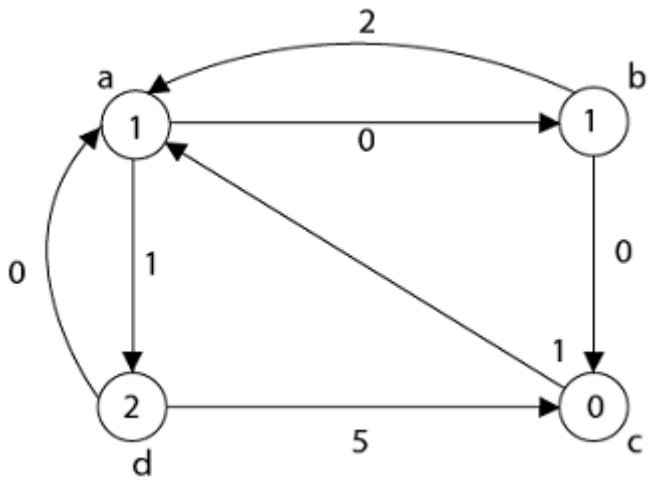
Source Vertex	Destination Vertex	Minimum Weight Associated
A	A	0
A	B	0
A	C	0
A	D	1

Case 2: Considering 'b' as a source vertex:



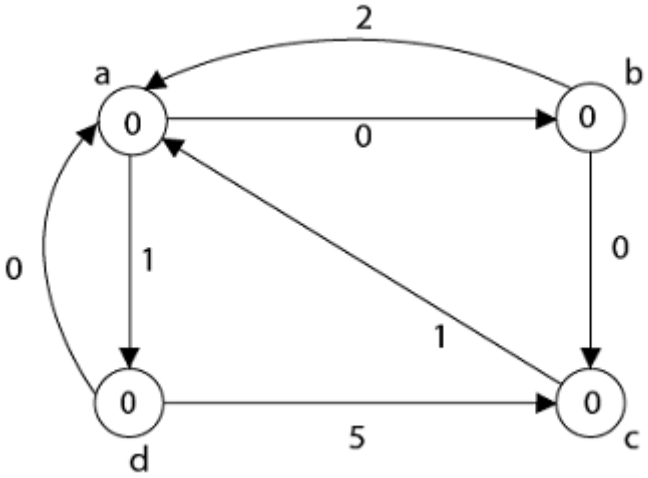
Source Vertex	Destination Vertex	Minimum Weight Associated
B	A	2
B	B	0
B	C	0
B	D	2

Case 3: Considering 'c' as a source vertex:



Source Vertex	Destination Vertex	Minimum Weight Associated
C	A	1
C	B	1
C	C	0
C	D	2

Case 4: Considering ’d’ as a source vertex:

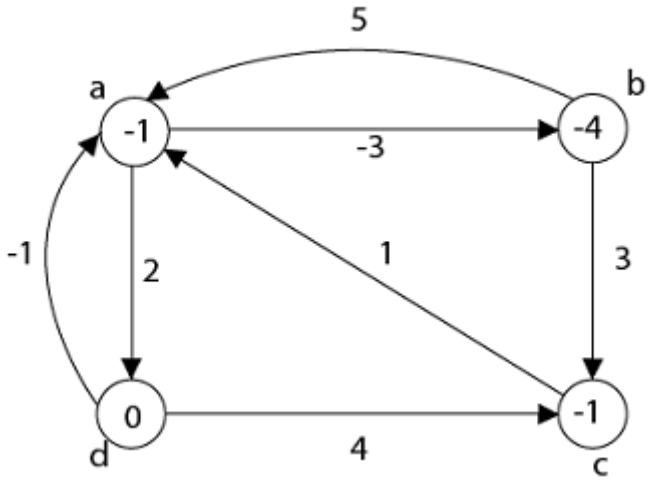


Source Vertex	Destination Vertex	Minimum Weight Associated
D	A	0
D	B	0
D	C	0
D	D	0

This matrix is representing the shortest path weights considering every vertex as a source vertex with all other vertices in the graph. This is basically running the Dijkstra algorithm V times.

	A	B	C	D
A	0	0	0	1
B	2	0	0	2
C	1	1	0	2
D	0	0	0	0

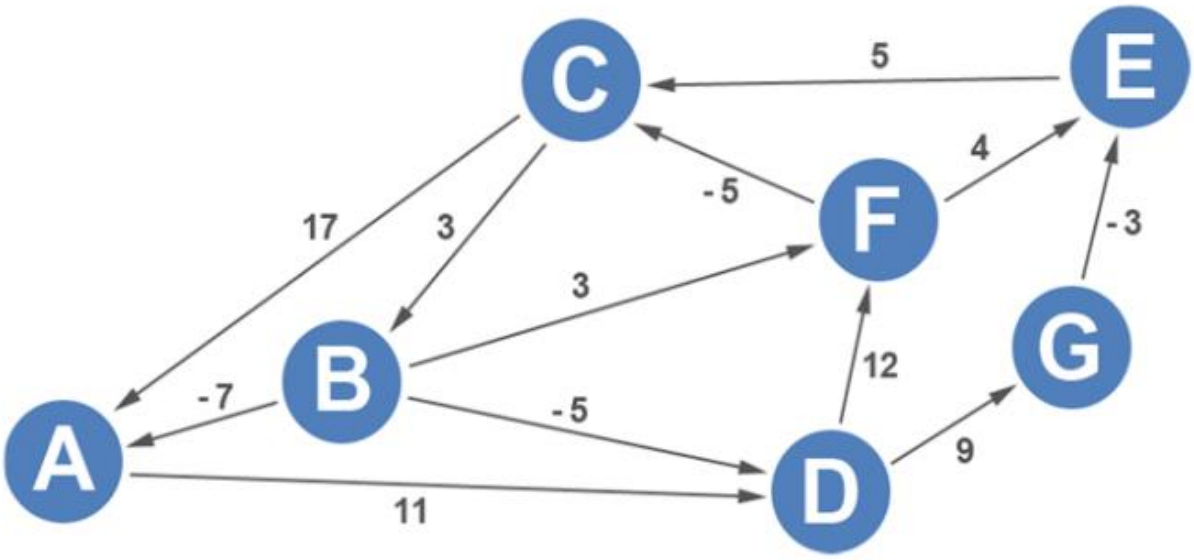
Step 5: Again restoring the original graph after finding the shortest path weights. This will be done by using the recursive formulation as w (u, v) = w (u, v) + h [v]-h [u]



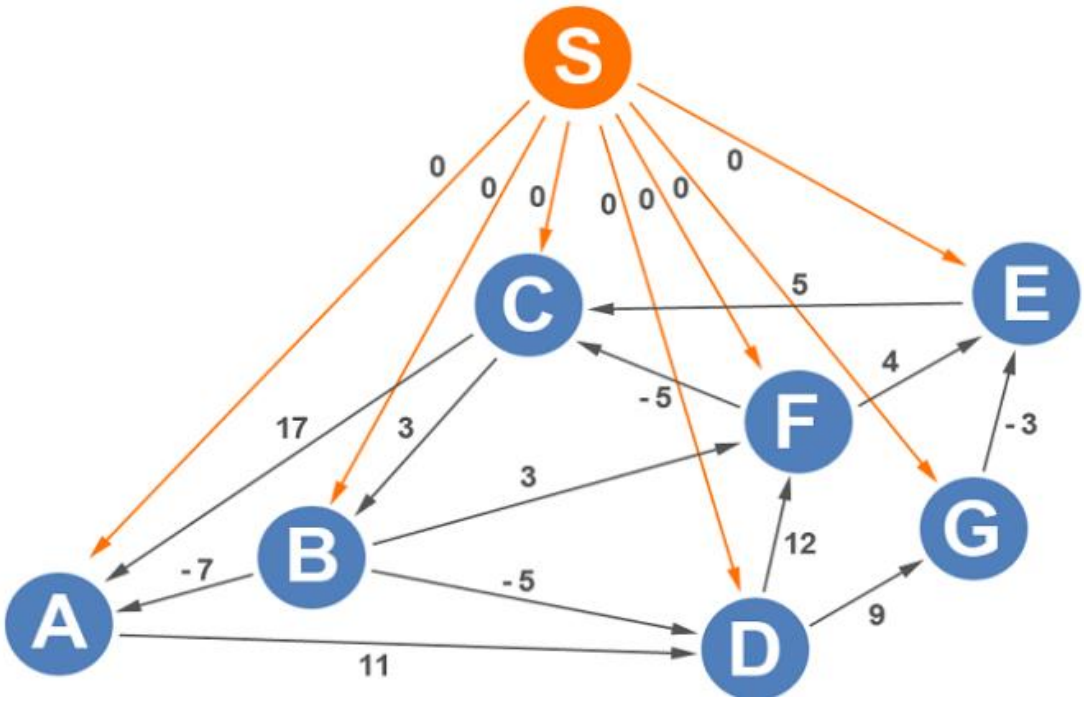
	A	B	C	D
A	0	-3	0	2
B	4	0	3	6
C	1	-2	0	3
D	-1	-4	-1	0

This matrix is obtained by restoring the graph to its original weights. This is done using the formulation as w (u, v) = w (u, v) +h[v]-h[u].

Example 3: Consider the following graph. Apply Johnson Algorithm for computing shortest path weights among all pair of vertices.



Step 1: Consider the new vertex s as a source vertex and make the shortest path weights from s to all other vertices in the graph as 0



Step 2: Apply Bellman ford algorithm for computing the minimum weight on each vertex.

Now, our objective is to re-weight all the negative weight edges of the graph in order to obtain non negative weight edges.

Weights associated with all the edges are stored in the table in the alphabetical order:

Edges	A-D	B-A	B-D	B-F	C-A	C-B	D-F	D-G	E-C	F-C	F-E	G-E
Weights	11	-7	-5	3	17	3	12	9	5	-5	4	-3
Edges	S-A	S-B	S-C	S-D	S-E	S-F	S-G					
Weights	0	0	0	0	0	0	0					

Source vertex is assigned distance as 0 and remaining vertex is assigned the distance as ∞

	S	A	B	C	D	E	F	G
Distance	0	∞	∞	∞	∞	∞	∞	∞
Predecessor Node	Nil	Nil	Nil	Nil	Nil	Nil	Nil	Nil

The edges are relaxed in following order.

Edges A-D, B-A, B-D, B-F, C-A, C-B, D-F, D-G, E-C, F-C, F-E, and G-E have current distances of infinity. Edges that start with the source vertex, S have a distance of 0 to S, so distances from S can be computed.
During the first iteration, edges S-A, S-B, S-C, S-D, S-E, S-F, and S-G are the only edges to be considered. Looking at the graph, the distance to all vertices from the source is 0. The predecessor to all vertices after the first iteration is S.

Iteration 1	S	A	B	C	D	E	F	G
Distance	0	∞	∞	∞	∞	∞	∞	∞
Predecessor Node	Nil	S	S	S	S	S	S	S

Start the Iteration 2, all the edges will be relaxed once again using the Relax procedure of computing shortest path weights.

Edges	Distance from source to destination	Update
A-D	0+11=11	Faster to get to D via S-D
B-A	0+(-7)= -7	(-7) is smaller , update to (-7)
B-D	0+(-5)= -5	(-5) is smaller , update to (-5)
B-F	0+3=3	Faster to get to F via S-F
C-A	0+17=17	Faster to get to A via S-A
C-B	0+3=3	Faster to get to B via S-B
D-F	(-5) +12=7	Faster to get to F via S-F
D-G	(-5) +9=4	Faster to get to G via S-G
E-C	0+5=5	Faster to get to C via S-C
F-C	0+(-5)=-5	(-5) is smaller , update to (-5)
F-E	0+4=4	Faster to get to E via S-E
G-E	0+(-3)=-3	(-3) is smaller , update to (-3)
S-A	0+0=0	Faster to get to A via S-B-A
S-B	0+0=0	No change
S-C	0+0=0	Faster to get to C via S-F-C
S-D	0+0=0	Faster to get to D via S-B-D
S-E	0+0=0	Faster to get to E via S-G-E
S-F	0+0=0	No change
S-G	0+0=0	No change

After Iteration 2 of bellman ford Algorithm we get resultant distance and predecessor values.

Iteration 2	S	A	B	C	D	E	F	G
Distance	0	-7	□	-5	-5	-3	□	□
Predecessor Node	Nil	B	S	F	B	G	S	S

Start the Iteration 3, all the edges will be relaxed once again using the Relax procedure of computing shortest path weights. Remove all the edges that originate from the source vertex S as they will not be further updated.

Edges	Distance from source to destination	Update
A-D	(-7)+11=4	Faster to get to D via S-B-D
B-A	0+(-7)= -7	No change
B-D	0+(-5)= -5	No change
B-F	0+3=3	No change
C-A	(-5)+17=12	Faster to get to A via S-B-A
C-B	(-5)+3=-2	(-2) is smaller, update (-2)
D-F	(-5) +12=7	Faster to get to F via S-F
D-G	(-5) +9=4	Faster to get to G via S-G
E-C	(-3)+5=2	Faster to get to C via S-F-C
F-C	0+(-5)=-5	No change
F-E	0+4=4	Faster to get to E via S-G-E
G-E	0+(-3)=-3	No change

After Iteration 3 of bellman ford Algorithm we get resultant distance and predecessor values.

Iteration 3	S	A	B	C	D	E	F	G
Distance	0	-7	-2	-5	-5	-3	□	□
Predecessor Node	Nil	B	C	F	B	G	S	S

Start the Iteration 4, all the edges will be relaxed once again using the Relax procedure of computing shortest path weights.

Edges	Distance from source to destination	Update
A-D	(-7)+11=4	No change
B-A	-2+(-7)= -9	-9 is smaller so update to -9 (S-F-C-B-A)
B-D	-2+(-5)= -7	-7 is smaller so update to -7 (S-F-C-B-D)
B-F	-2+3=1	No change
C-A	(-5)+17=12	No change
C-B	(-5)+3=-2	No change
D-F	(-7) +12=5	No change
D-G	(-7) +9=2	No change
E-C	(-3)+5=2	No change
F-C	0+(-5)=-5	No change
F-E	0+4=4	No change

G-E	0+(-3)=-3	No change

After Iteration 4 of bellman ford Algorithm we get resultant distance and predecessor values.

Iteration 4	S	A	B	C	D	E	F	G
Distance	0	-9	-2	-5	-7	-3	∞	∞
Predecessor Node	Nil	B	C	F	B	G	S	S

Start the Iteration 5, all the edges will be relaxed once again using the Relax procedure of computing shortest path weights.

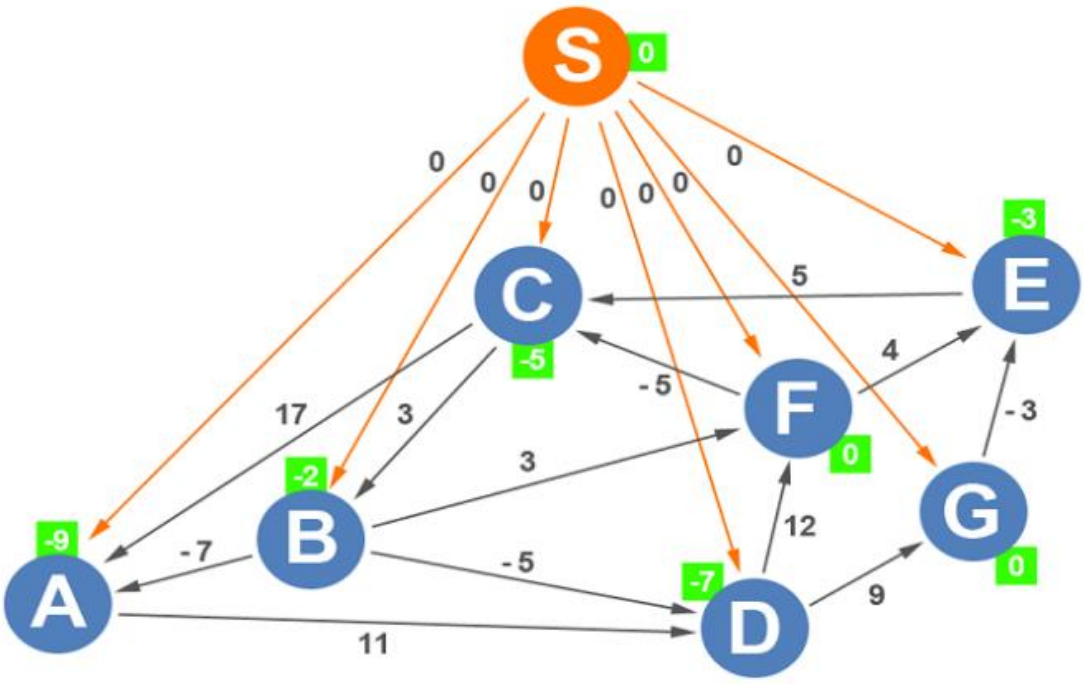
Edges	Distance from source to destination	Update
A-D	(-9)+11=2	No change
B-A	-2+(-7)= -9	-9 is smaller so update to -9 (S-F-C-B-A)
B-D	-2+(-5)= -7	-7 is smaller so update to -7 (S-F-C-B-D)
B-F	-2+3=1	No change
C-A	(-5)+17=12	No change
C-B	(-5)+3=-2	No change
D-F	(-7) +12=5	No change
D-G	(-7) +9=2	No change
E-C	(-3)+5=2	No change
F-C	0+(-5)=-5	No change
F-E	0+4=4	No change
G-E	0+(-3)=-3	No change

After Iteration 5 of bellman ford Algorithm we get resultant distance and predecessor values.

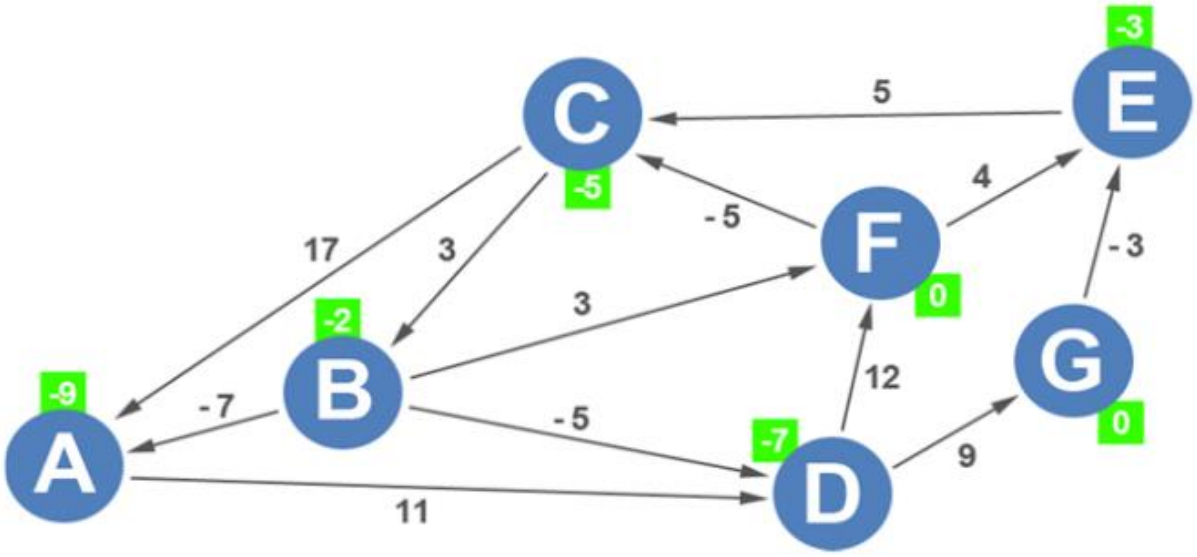
Iteration 5	S	A	B	C	D	E	F	G
Distance	0	-9	-2	-5	-7	-3	∞	∞
Predecessor Node	Nil	B	C	F	B	G	S	S

In the fifth Iteration we have observed that there is no change in the distance and predecessor matrix. This means that bellman ford portion of Johnson's algorithm is complete.

The shortest path weights to each vertex is updated from source vertex S. Initially, the shortest path weights from S-A was 0.But after applying the Bellman ford Algorithm the shortest path weights is updated by -9 via path S-F-C-B-A.



Now we have computed the shortest path weights at each of the vertex. Hence, we can simply remove the source vertex s from the graph.



Now, we will reweight the edges of the graph in order to obtain the non-negative weight edges so that we can further apply the Dijkstra Algorithm. For this, we will use the following recursive formulation for reweighting the edges is w (u. v) = w (u, v) +h [u]-h {v]

Edges	A-D	B-A	B-D	B-F	C-A	C-B	D-F	D-G	E-C	F-C	F-E	G-E
Weights	11	-7	-5	3	17	3	12	9	5	-5	4	-3

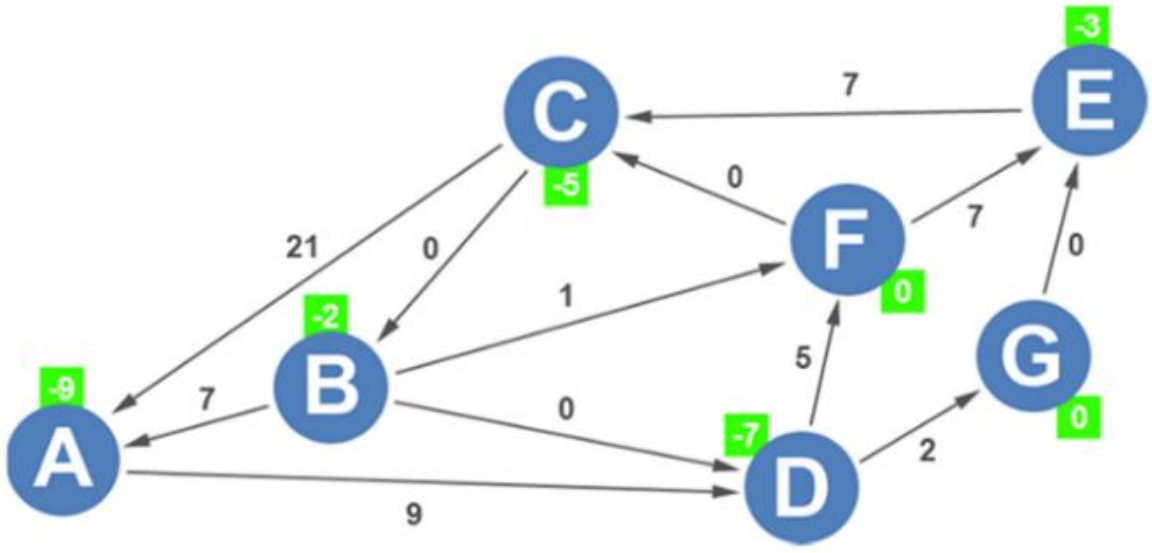
After Iteration 5 of bellman ford Algorithm we get resultant distance and predecessor values.

Iteration 5	S	A	B	C	D	E	F	G
Distance	0	-9	-2	-5	-7	-3	∞	∞
Predecessor Node	Nil	B	C	F	B	G	S	S

Now we will reweight all the edges using both table values and recursive formulation in order to obtain non-negative weight edges.

Edges	A-D	B-A	B-D	B-F	C-A	C-B	D-F	D-G	E-C	F-C	F-E	G-E
Updated Weights	9	7	0	1	21	0	5	2	7	0	7	0

Hence, the final graph after reweighting the edges. Now we will simply apply Dijkstra Algorithm to compute shortest path weights from a source to all other vertex in the graph. This Process is to be repeated V times to compute all pair shortest path in O ($V^2 \log V$)



Animated Links of Johnson's Algorithm:

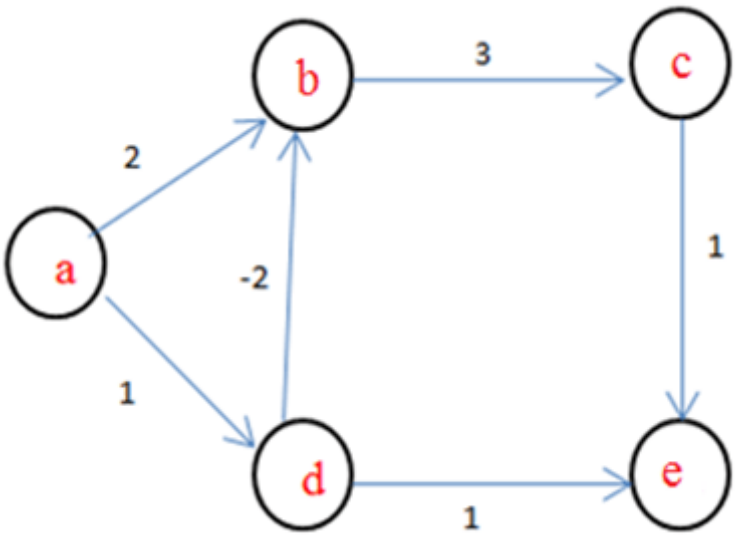
https://www.youtube.com/watch?v=hLEgT-2t8Ag
https://www.youtube.com/watch?v=UVCBj1EQRfo

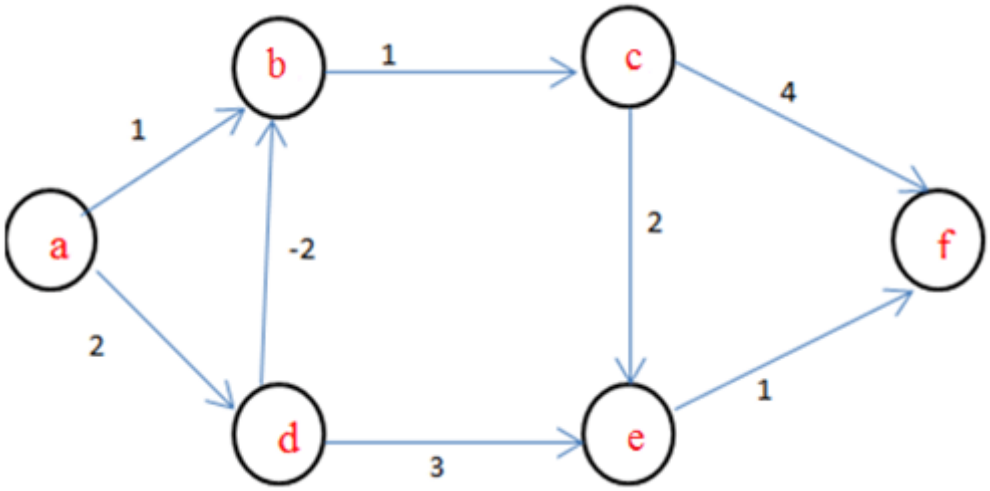
Applications of Johnson's Algorithm:

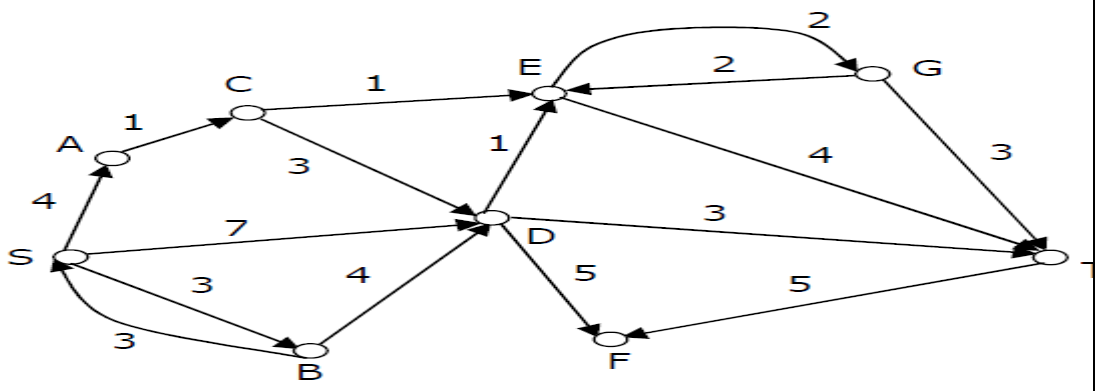
Johnson's algorithm is the shortest path algorithm that deals with the all pair of vertices. Johnson's algorithm is used in case of sparse graphs (graphs with a few edges.

4.5.14. Objective Type Questions: Company and GATE/PSUs:

1	How many times for loop in the Bellman Ford Algorithm gets executed?
A	V
B	V-1
C	E
D	E-1
ANS	B

2	Consider the following graph. What is the minimum cost to travel from node A to node C? 
A	5
B	2
C	1
D	3
ANS	B

3	In the given graph, identify the path that has minimum cost to travel from node a to node f. 
A	a-b-c-f
B	a-b-e-f
C	a-d-b-c-f
D	a-d-b-c-e-f
ANS	D

4	Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex v is updated only when a strictly shorter path to v is discovered. 
A	SDT
B	SBDT
C	SACDT
D	SACET
ANS	D