R.PUSHPANATHAN.,MCA.,M.Phil.,MBA

ADS UNIT-III

Breadth First Search (BFS) Algorithm with EXAMPLE

What is BFS Algorithm (Breadth-First Search)?

Breadth-first search (BFS) is an algorithm that is used to graph data or searching tree or traversing structures. The full form of BFS is the Breadth-first search.

The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion. This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. Remember, BFS accesses these nodes one by one.

Once the algorithm visits and marks the starting node, then it moves towards the nearest unvisited nodes and analyses them. Once visited, all nodes are marked. These iterations continue until all the nodes of the graph have been successfully visited and marked.

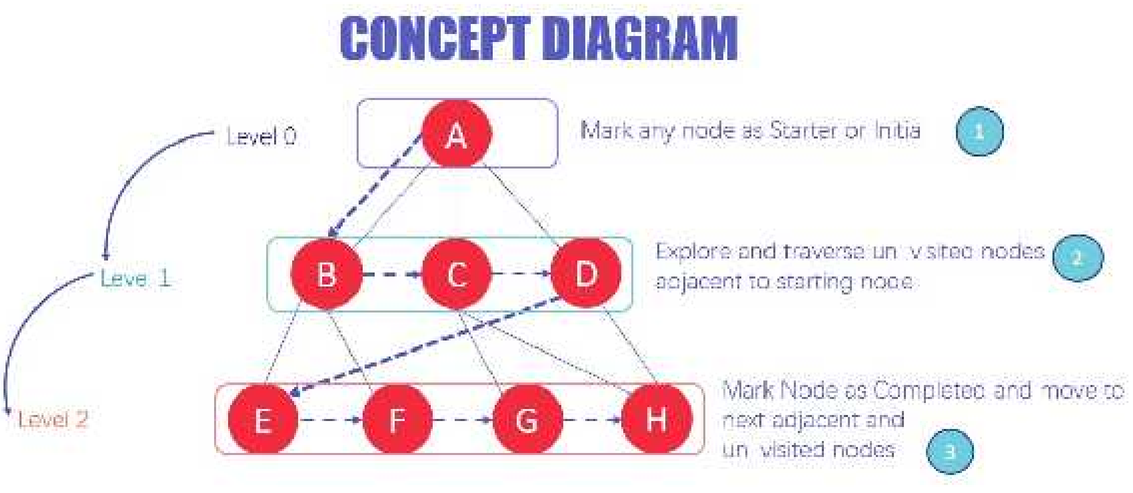
In this Algorithm, you will learn:

- What is BFS Algorithm (Breadth-First Search)?
- What is Graph traversals?
- The architecture of BFS algorithm
- Why do we need BFS Algorithm?
- How does BFS Algorithm Work?
- Example BFS Algorithm
- Rules of BFS Algorithm
- Applications of BFS Algorithm

What is Graph traversals?

A graph traversal is a commonly used methodology for locating the vertex position in the graph. It is an advanced search algorithm that can analyze the graph with speed and precision along with marking the sequence of the visited vertices. This process enables you to quickly visit each node in a graph without being locked in an infinite loop.

The architecture of BFS algorithm


CONCEPT DIAGRAM
Level 0
Level 1
Level 2
A
B
C
D
E
F
G
H
Mark any node as Starter or Initial
1
Explore and traverse un-visited nodes adjacent to starting node
2
Mark Node as Completed and move to next adjacent and un-visited nodes
3

In the various levels of the data, you can mark any node as the starting or initial node to begin traversing. The BFS will visit the node and mark it as visited and places it in the queue.

Now the BFS will visit the nearest and un-visited nodes and marks them. These values are also added to the queue. The queue works on the FIFO model.

In a similar manner, the remaining nearest and un-visited nodes on the graph are analyzed marked and added to the queue. These items are deleted from the queue as receive and printed as the result.

Why do we need BFS Algorithm?

There are numerous reasons to utilize the BFS Algorithm to use as searching for your dataset. Some of the most vital aspects that make this algorithm your first choice are:

- BFS is useful for analyzing the nodes in a graph and constructing the shortest path of traversing through these.
- BFS can traverse through a graph in the smallest number of iterations.
- The architecture of the BFS algorithm is simple and robust.

- The result of the BFS algorithm holds a high level of accuracy in comparison to other algorithms.
- BFS iterations are seamless, and there is no possibility of this algorithm getting caught up in an infinite loop problem.

BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

Visited

Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

Start by putting any one of the graph's vertices at the back of a queue.

Take the front item of the queue and add it to the visited list.

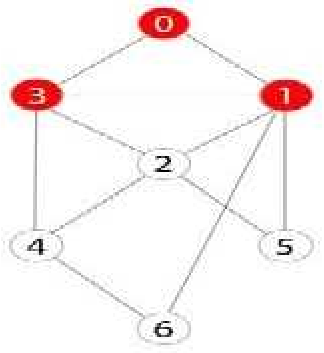
Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

Example BFS Algorithm

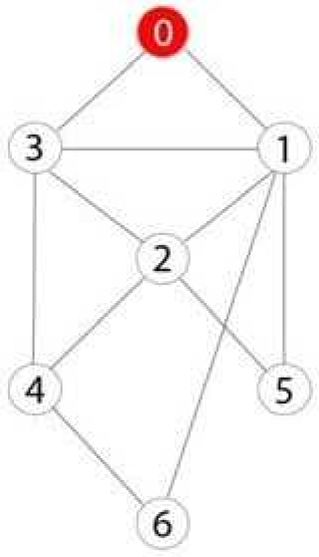
STEP1:



1. Mark 0 as visited
2. Insert 0 to the queue
3. Traverse the un-visited adjacent nodes which are 3 and 1

You have a graph of seven numbers ranging from 0 – 6.

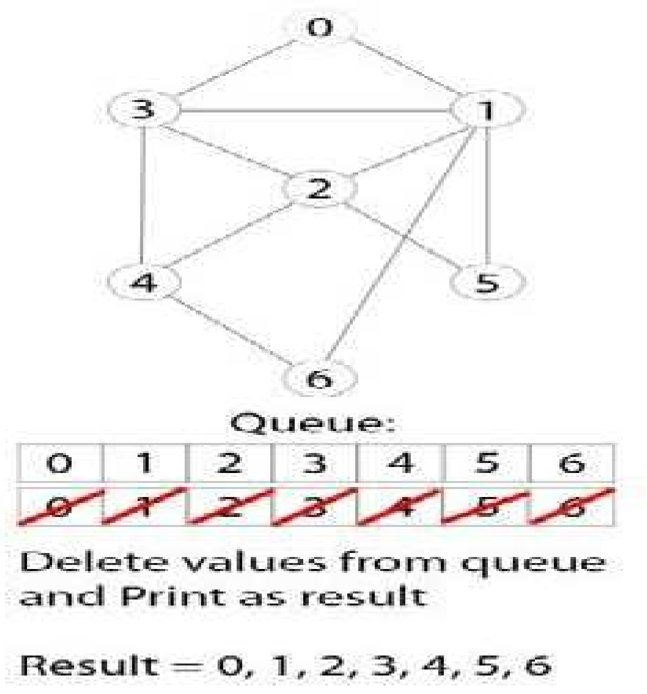
STEP2:



Root node = 0

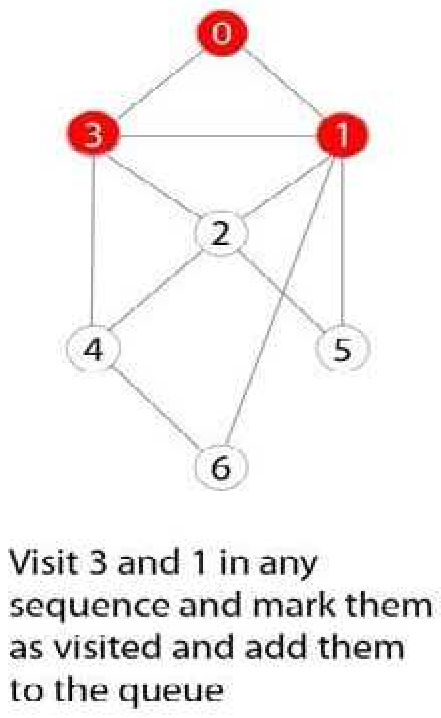
0 or zero has been marked as a root node.

STEP3:


0
3
1
2
4
5
6
Queue:
0 1 2 3 4 5 6
~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~
Delete values from queue and Print as result
Result = 0, 1, 2, 3, 4, 5, 6

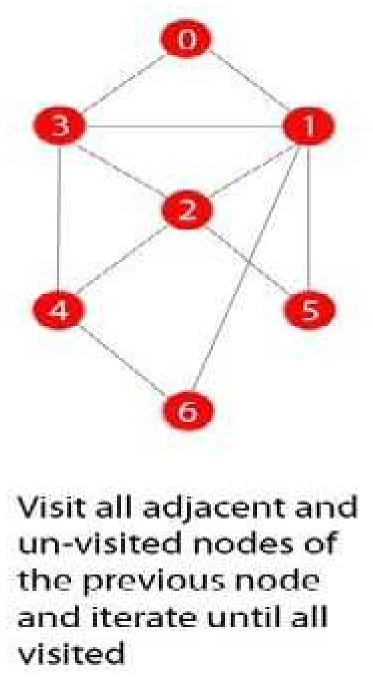
0 is visited, marked, and inserted into the queue data structure.

STEP4:


0
3
1
2
4
5
6
Visit 3 and 1 in any sequence and mark them as visited and add them to the queue

Remaining 0 adjacent and unvisited nodes are visited, marked, and inserted into the queue.

STEP5:


0
1
2
3
4
5
6
Visit all adjacent and un-visited nodes of the previous node and iterate until all visited

Traversing iterations are repeated until all nodes are visited.

Rules of BFS Algorithm

Here, are important rules for using BFS algorithm:

- A queue (FIFO-First in First Out) data structure is used by BFS.
- You mark any node in the graph as root and start traversing the data from it.
- BFS traverses all the nodes in the graph and keeps dropping them as completed.
- BFS visits an adjacent unvisited node, marks it as done, and inserts it into a queue.
- Removes the previous vertex from the queue in case no adjacent vertex is found.
- BFS algorithm iterates until all the vertices in the graph are successfully traversed and marked as completed.
- There are no loops caused by BFS during the traversing of data from any node.

Applications of BFS Algorithm

- Let’s take a look at some of the real-life applications where a BFS algorithm implementation can be highly effective.
- Un-weighted Graphs: BFS algorithm can easily create the shortest path and a minimum spanning tree to visit all the vertices of the graph in the shortest time possible with high accuracy.
- P2P Networks: BFS can be implemented to locate all the nearest or neighboring nodes in a peer to peer network. This will find the required data faster.
- Web Crawlers: Search engines or web crawlers can easily build multiple levels of indexes by employing BFS. BFS implementation starts from the source, which is the web page, and then it visits all the links from that source.
- Navigation Systems: BFS can help find all the neighboring locations from the main or source location.
- Network Broadcasting: A broadcasted packet is guided by the BFS algorithm to find and reach all the nodes it has the address for.

BFS Implementation Program

```
#include<bits/stdc++.h>

using namespace std;

#define MAX 1005

vector <int> adj[MAX];      // adjacency matrix, where adj[i] is a list, which denotes there are edges from i to each vertex in the list adj[i]

bool visited[MAX];          // boolean array, hacing value true / false, which denotes if a vertex 'i' has been visited or not.
```

```
void init(){            // initialization function
    int i;
    for(i = 0; i < MAX; i++){
        visited[i] = false; // marking all vertex as unvisited
        adj[i].clear();     // clearing all edges
    }
}

void BFS(int start){

    init();
    queue <int> q;
    q.push(start);

    int iterator, current_node, next_node;
    cout<<"BFS Traversal is:\n";
    while(q.empty() == false){
        current_node = q.front();
        q.pop();

        if(visited[current_node] == true){
            continue;

        }



        cout<<current_node<<" ";

        visited[current_node] = true;



        for(iterator = 0; iterator < adj[current_node].size(); iterator++){

            next_node = adj[current_node][iterator];



            if(visited[next_node] == false) {

                q.push(next_node);

            }

        }

    }



}



int main(){

    int vertices, edges;

    cout<<"Enter Number of Vertices:\n";

    cin>>vertices;

    cout<<"Enter number of edges:\n";
    cin>>edges;


    int i;
    int source, destination;
    cout<<"Enter Edges as (source) <space> (destination):\n";
    for(i=0; i<edges; i++){
        cin>>source>>destination;
        if(source > vertices || destination > vertices){
            cout<<"Invalid Edge";
            i--;
            continue;
        }
        adj[source].push_back(destination);
        adj[destination].push_back(source);
    }


    int start;
    cout<<"Enter Starting Vertex:";
    cin>>start;


    BFS(start);
}
```

DEPTH FIRST SEARCH ALGORITHM | DFS EXAMPLE

DEPTH FIRST SEARCH-

Depth First Search or DFS is a graph traversal algorithm.

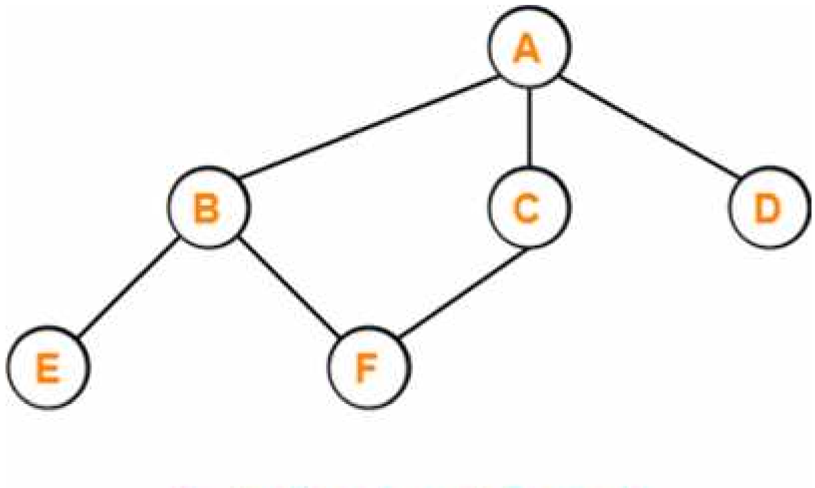
It is used for traversing or searching a graph in a systematic fashion.

DFS uses a strategy that searches “deeper” in the graph whenever possible.

Stack data structure is used in the implementation of depth first search.

DFS Example-

Consider the following graph-



Depth First Search Example

The depth first search traversal order of the above graph is-

A, B, E, F, C, D

Depth First Search Algorithm-

```
DFS (V,E)
for each vertex u in V[G]
do color[v] ← WHITE
π[v] ← NIL
time ← 0
for each vertex v in V[G]
do if color[v] ← WHITE
then Depth_First_Search(v)
Depth_First_Search (v)
color[v] ← GRAY
time ← time + 1
d[v] ← time
for each vertex u adjacent to v
do if color[u] ← WHITE
π[u] ← v
Depth_First_Search(u)
color[v] ← BLACK
time ← time + 1
f[v] ← time
```

Explanation-

The above depth first search algorithm is explained in the following steps-

Step-01

Create and maintain 4 variables for each vertex of the graph.

For any vertex ‘v’ of the graph, these 4 variables are-

1. color[v]-

This variable represents the color of the vertex ‘v’ at the given point of time.

The possible values of this variable are- WHITE, GREY and BLACK.

WHITE color of the vertex signifies that it has not been discovered yet.

GREY color of the vertex signifies that it has been discovered and it is being processed.

BLACK color of the vertex signifies that it has been completely processed.

2. Π[v]-

This variable represents the predecessor of vertex ‘v’.

3. d[v]-

This variable represents a timestamp when a vertex ‘v’ is discovered.

3. f[v]-

This variable represents a timestamp when the processing of vertex ‘v’ is completed.

Step-02

For each vertex of the graph, initialize the variables as-

color[v] = WHITE

π[v] = NIL

time = 0 (Global Variable acting as a timer)

Step-03

Repeat the following procedure until all the vertices of the graph become BLACK-

Consider any white vertex ‘v’ and call the following Depth_First_Search function on it.

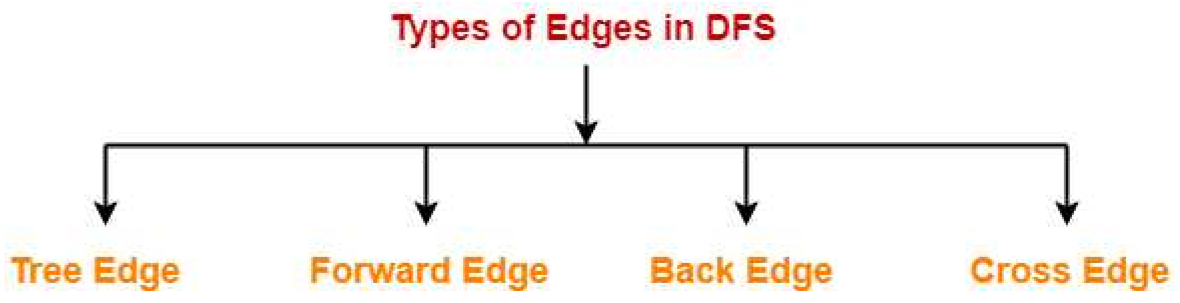
```
Depth_First_Search (G,v)

1. color[v] = GRAY

2. time = time + 1

3. d[v] = time

4. For each adjacent WHITE vertex ‘u’ of ‘v’, set π[u] = v and call Depth_First_Search (G,u)

5. color[v] = BLACK

6. time = time + 1

7. f[v] = time
```

DFS Time Complexity-

The total running time for Depth First Search is θ (V+E).

Types of Edges in DFS-

After a DFS traversal of any graph G, all its edges can be put in one of the following 4 classes-



1. Tree Edge-

A tree edge is an edge that is included in the DFS tree.

2. Back Edge-

An edge from a vertex ‘u’ to one of its ancestors ‘v’ is called as a back edge.

A self-loop is considered as a back edge.

A back edge is discovered when-

DFS tries to extend the visit from a vertex ‘u’ to vertex ‘v’

And vertex ‘v’ is found to be an ancestor of vertex ‘u’ and grey at that time.

3. Forward Edge-

An edge from a vertex ‘u’ to one of its descendants ‘v’ is called as a forward edge.

A forward edge is discovered when-

DFS tries to extend the visit from a vertex ‘u’ to a vertex ‘v’

And finds that color(v) = BLACK and $d(v) > d(u)$.

4. Cross Edge-

An edge from a vertex ‘u’ to a vertex ‘v’ that is neither its ancestor nor its descendant is called as a cross edge.

A cross edge is discovered when-

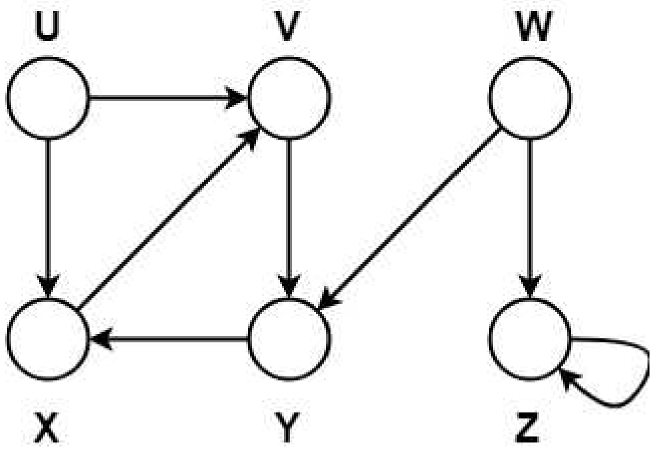
DFS tries to extend the visit from a vertex ‘u’ to a vertex ‘v’

And finds that color(v) = BLACK and $d(v) < d(u)$.

PRACTICE PROBLEM BASED ON DEPTH FIRST SEARCH-

Problem-

Compute the DFS tree for the graph given below-



Also, show the discovery and finishing time for each vertex and classify the edges.

Solution-

Initially for all the vertices of the graph, we set the variables as-

color[v] = WHITE

π[v] = NIL

time = 0 (Global)

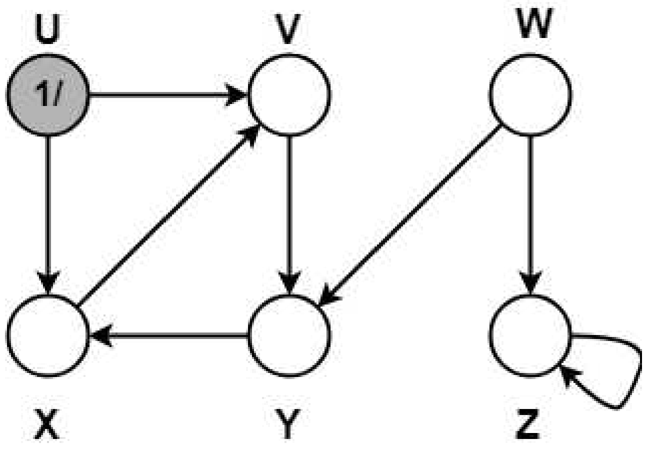
Let us start processing the graph from vertex U.

Step-01:

color[U] = GREY

time = 0 + 1 = 1

d[U] = 1



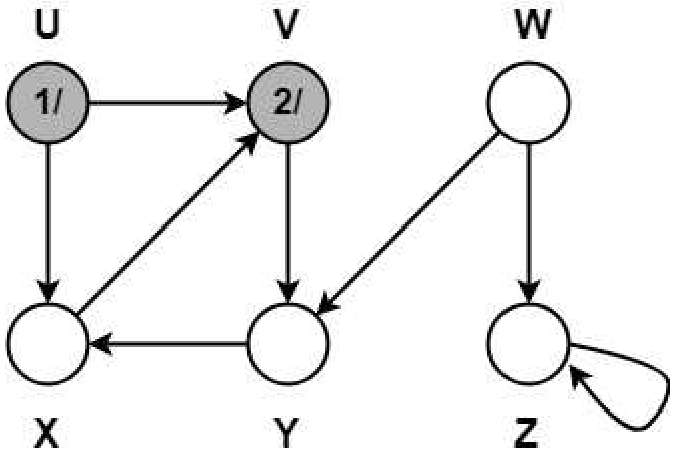
Step-02:

π[V] = U

color[V] = GREY

time = 1 + 1 = 2

d[V] = 2



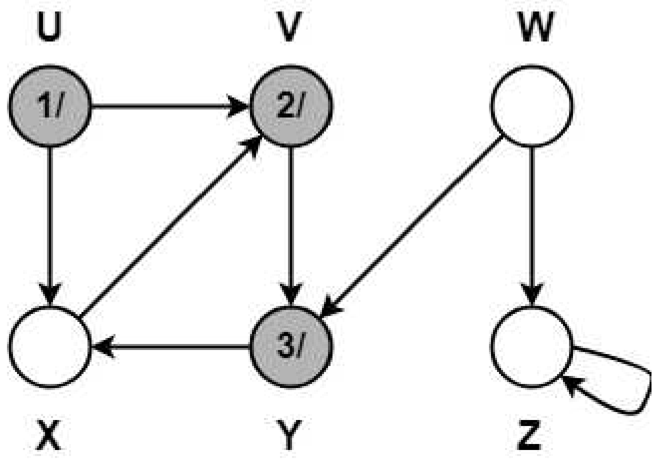
Step-03:

π[Y] = V

color[Y] = GREY

time = 2 + 1 = 3

d[Y] = 3



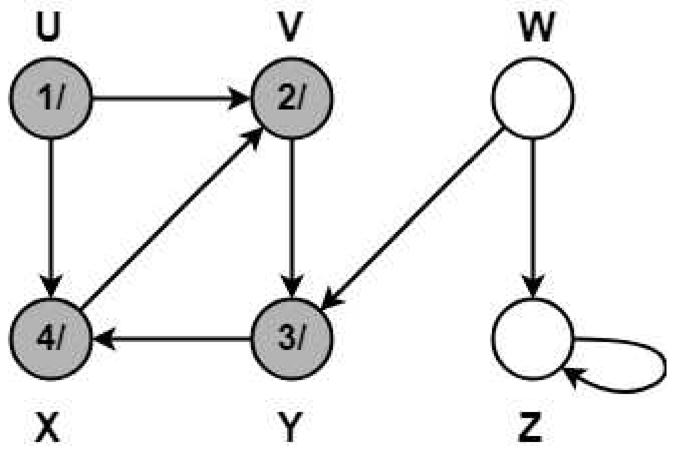
Step-04:

π[X] = Y

color[X] = GREY

time = 3 + 1 = 4

d[X] = 4



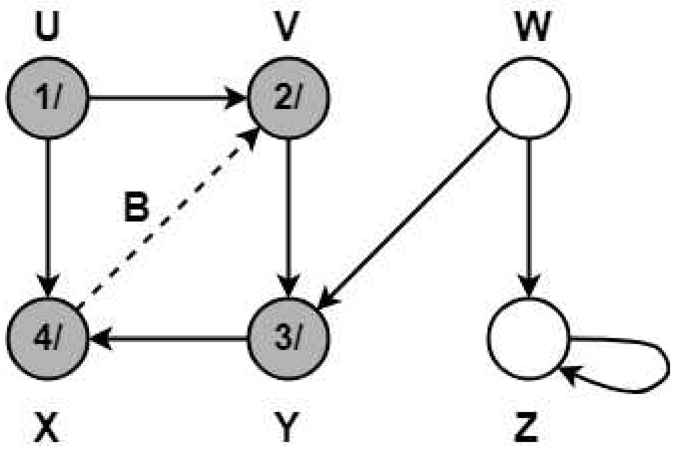
Step-05:

When DFS tries to extend the visit from vertex X to vertex V, it finds-

Vertex V is an ancestor of vertex X since it has already been discovered

Vertex V is GREY in color.

Thus, edge XV is a back edge.

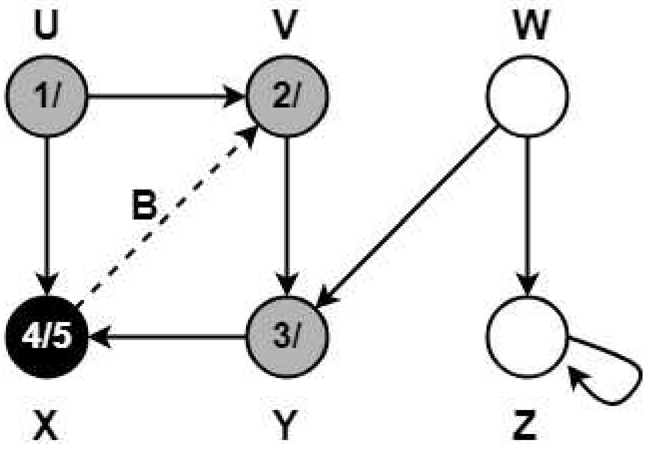


Step-06:

color[X] = BLACK

time = 4 + 1 = 5

f[X] = 5

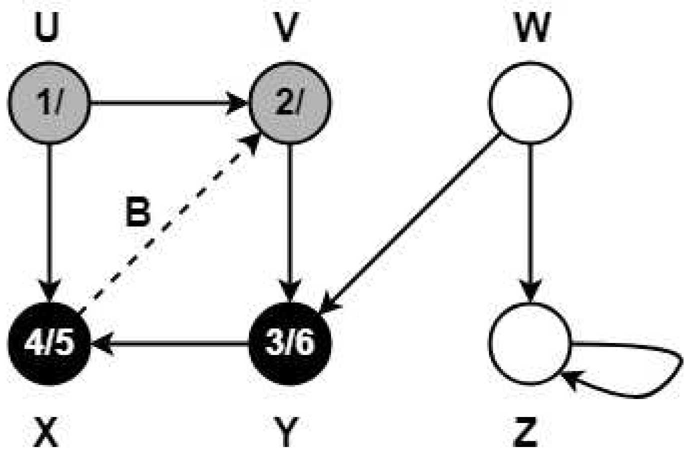


Step-07:

color[Y] = BLACK

time = 5 + 1 = 6

f[Y] = 6

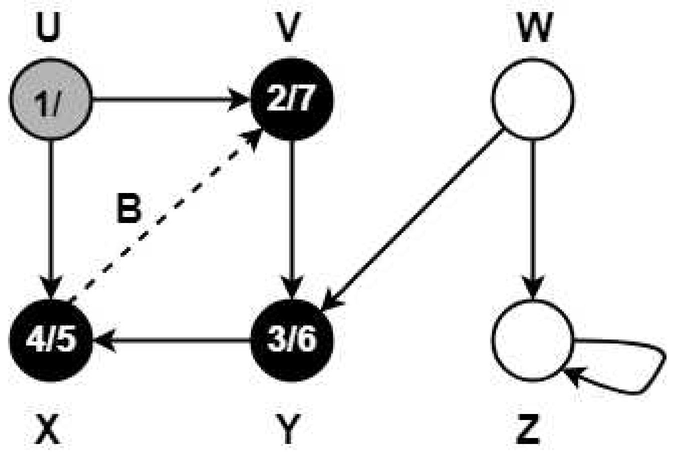


Step-08

color[V] = BLACK

time = 6 + 1 = 7

f[V] = 7



Step-09:

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

Vertex X has already been completely processed i.e. vertex X has finished and is black.

But vertex U has still not finished.

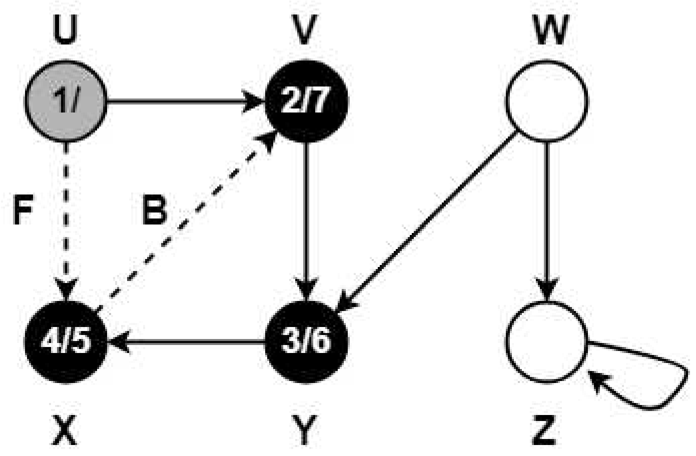
Alternatively,

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

Color(X) = BLACK

d(X) > d(U)

Thus, edge UX is a forward edge.

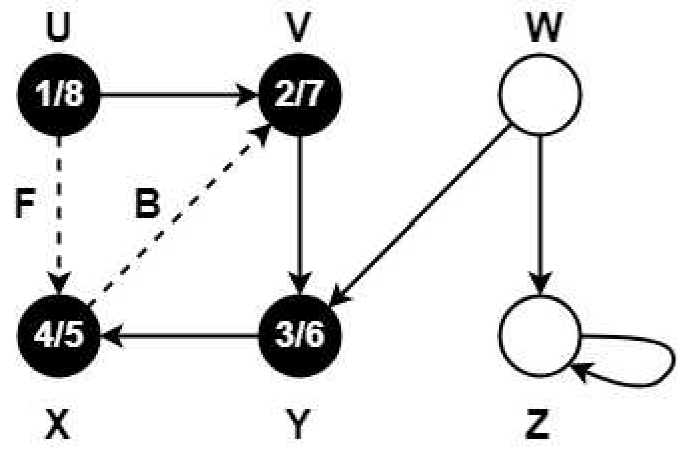


Step-10:

color[U] = BLACK

time = 7 + 1 = 8

f[U] = 8

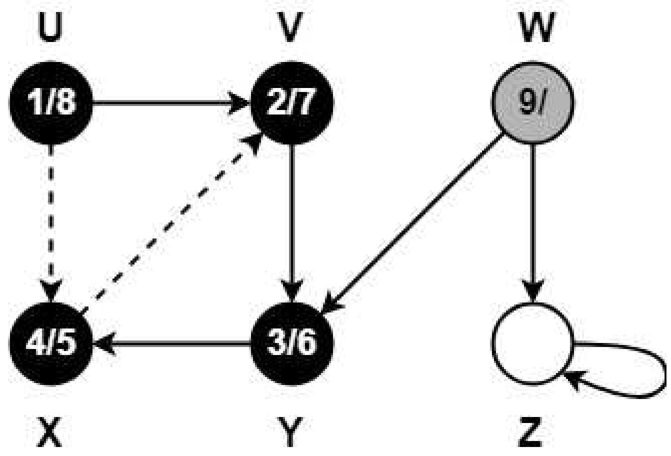


Step-11:

color[W] = GREY

time = 8 + 1 = 9

d[W] = 9



Step-12:

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

Vertex Y has already been completely processed i.e. vertex Y has finished.

Vertex Y is neither a descendant nor an ancestor of vertex W.

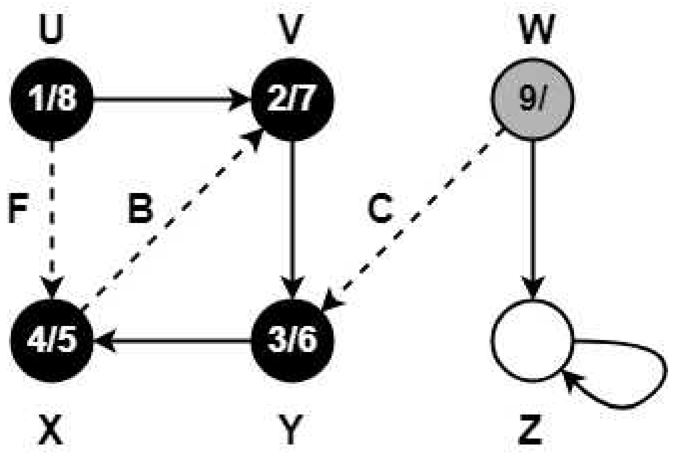
Alternatively,

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

Color(Y) = BLACK

d(Y) < d(W)

Thus, edge WY is a cross edge.



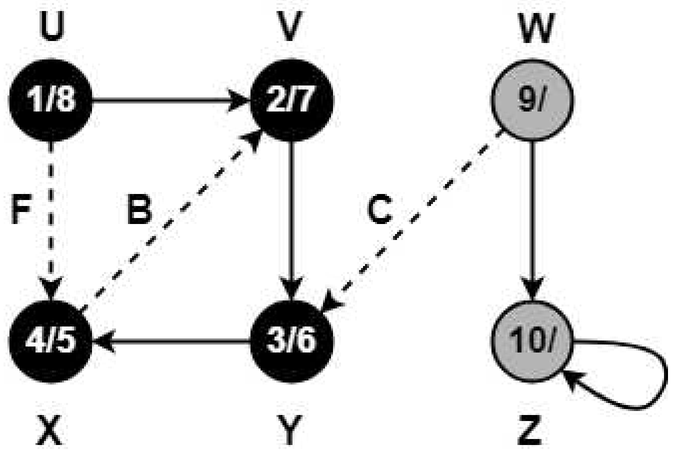
Step-13:

π[Z] = W

color[W] = GREY

time = 9 + 1 = 10

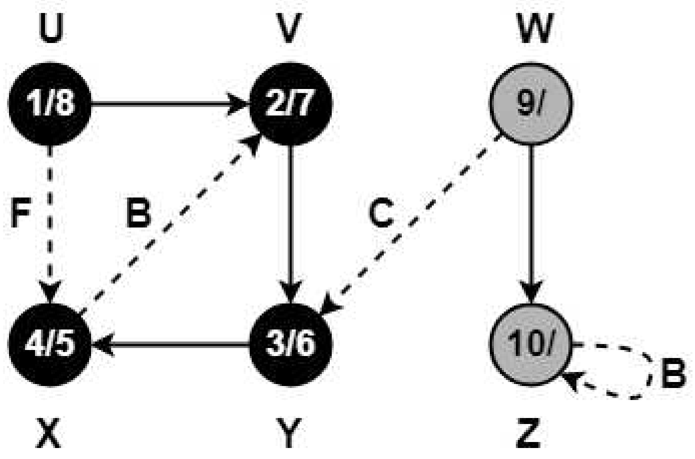
d[W] = 10



Step-14:

Since, self-loops are considered as back edges.

Therefore, self-loop present on vertex Z is considered as a back edge.

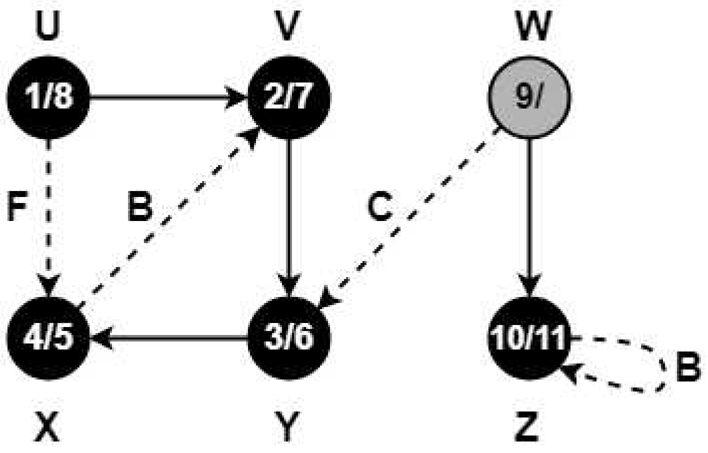


Step-15:

color[Z] = BLACK

time = 10 + 1 = 11

f[Z] = 11

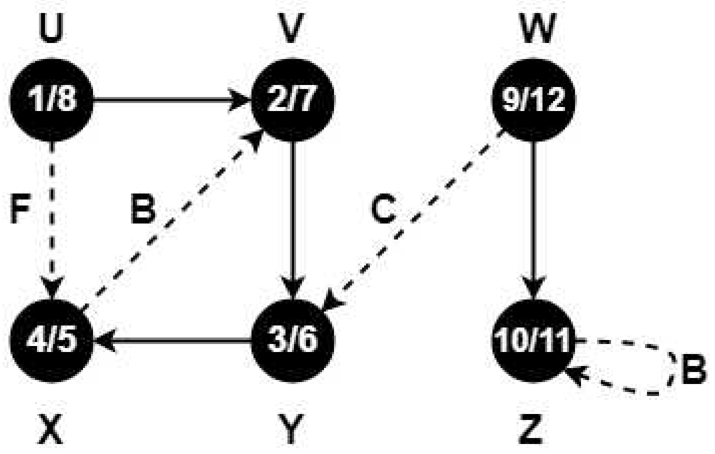


Step-16:

color[W] = BLACK

time = 11 + 1 = 12

f[W] = 12



Since all the vertices have turned black, so we stop.

program to print DFS traversal from

```
// a given vertex in a given graph
#include <bits/stdc++.h>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph {
public:
        map<int, bool> visited;
        map<int, list<int> > adj;

        // function to add an edge to graph
        void addEdge(int v, int w);

        // DFS traversal of the vertices
        // reachable from v
```

```
	void DFS(int v);
};

void Graph::addEdge(int v, int w)
{
	adj[v].push_back(w); // Add w to v’s list.
}

void Graph::DFS(int v)
{
	// Mark the current node as visited and
	// print it
	visited[v] = true;
	cout << v << " ";

	// Recur for all the vertices adjacent
	// to this vertex
	list<int>::iterator i;
	for (i = adj[v].begin(); i != adj[v].end(); ++i)
		if (!visited[*i])
			DFS(*i);
}

// Driver code
int main()
{
    // Create a graph given in the above diagram
    Graph g;
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
            " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}

// improved by Vishnudev C
```

TOPOLOGICAL SORT | TOPOLOGICAL SORT EXAMPLES

TOPOLOGICAL SORT-

Topological Sort is a linear ordering of the vertices in such a way that

if there is an edge in the DAG going from vertex ‘u’ to vertex ‘v’,

then ‘u’ comes before ‘v’ in the ordering.

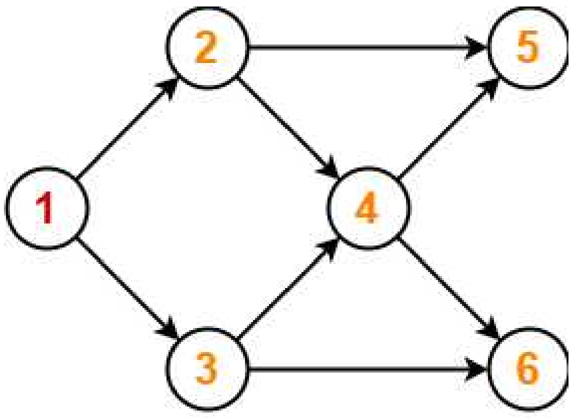
It is important to note that-

Topological Sorting is possible if and only if the graph is a Directed Acyclic Graph.

There may exist multiple different topological orderings for a given directed acyclic graph.

Topological Sort Example-

Consider the following directed acyclic graph-



Topological Sort Example

For this graph, following 4 different topological orderings are possible-

1 2 3 4 5 6

1 2 3 4 6 5

1 3 2 4 5 6

1 3 2 4 6 5

Applications of Topological Sort-

Few important applications of topological sort are-

Scheduling jobs from the given dependencies among jobs

Instruction Scheduling

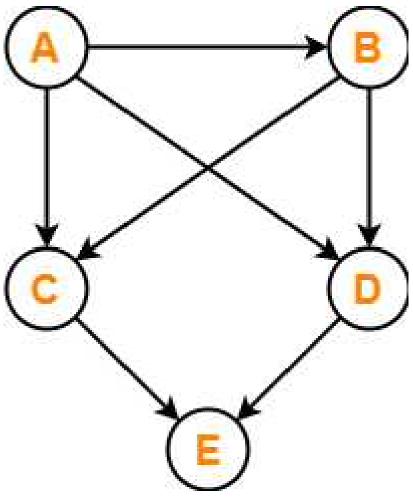
Determining the order of compilation tasks to perform in makefiles

Data Serialization

PRACTICE PROBLEMS BASED ON TOPOLOGICAL SORT-

Problem-01:

Find the number of different topological orderings possible for the given graph-

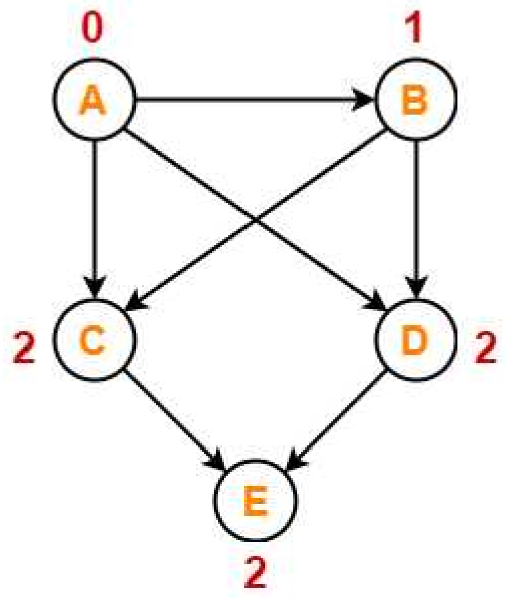


Solution-

The topological orderings of the above graph are found in the following steps-

Step-01:

Write in-degree of each vertex-

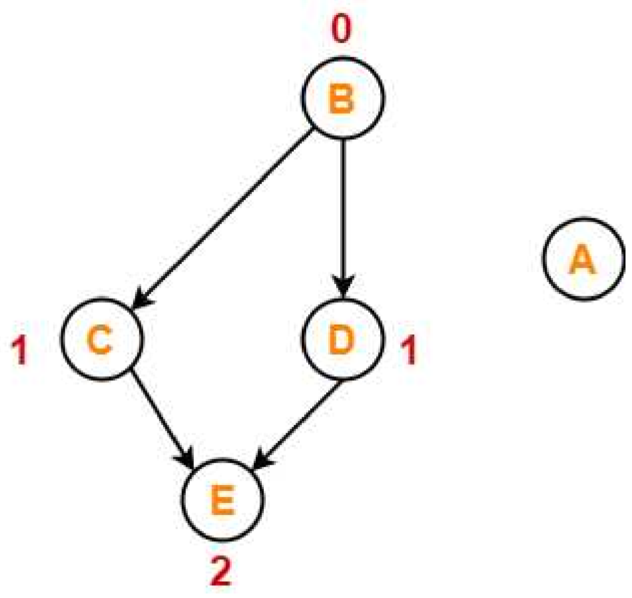


Step-02:

Vertex-A has the least in-degree.

So, remove vertex-A and its associated edges.

Now, update the in-degree of other vertices.

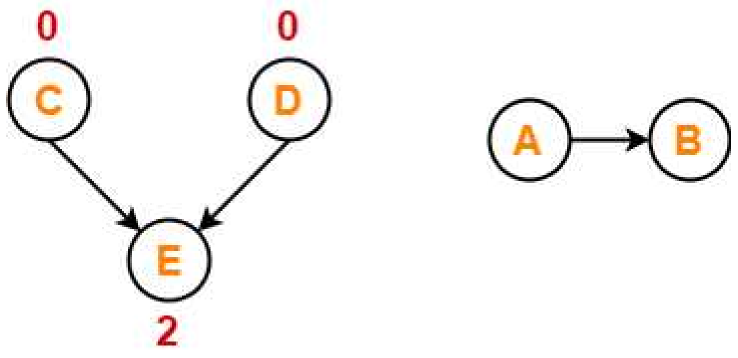


Step-03:

Vertex-B has the least in-degree.

So, remove vertex-B and its associated edges.

Now, update the in-degree of other vertices.



Step-04:

There are two vertices with the least in-degree. So, following 2 cases are possible-

In case-01,

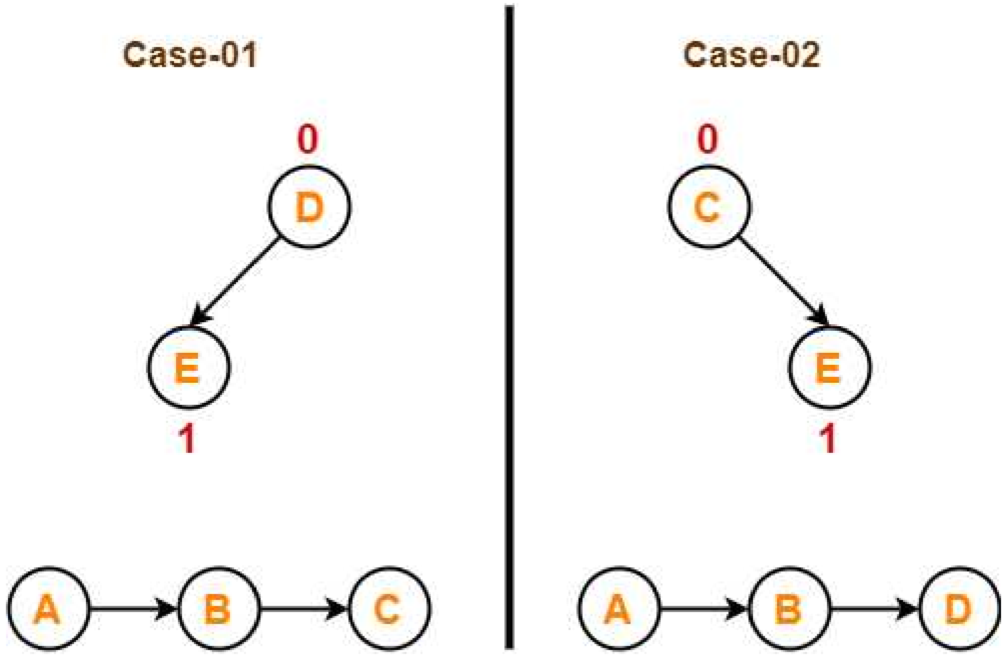
Remove vertex-C and its associated edges.

Then, update the in-degree of other vertices.

In case-02,

Remove vertex-D and its associated edges.

Then, update the in-degree of other vertices.



Step-05:

Now, the above two cases are continued separately in the similar manner.

In case-01,

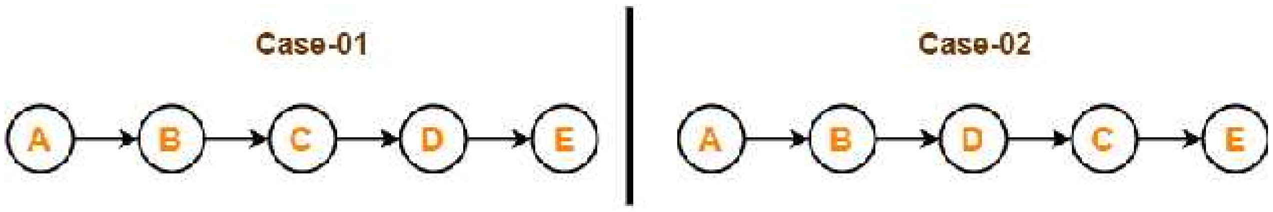
Remove vertex-D since it has the least in-degree.

Then, remove the remaining vertex-E.

In case-02,

Remove vertex-C since it has the least in-degree.

Then, remove the remaining vertex-E.



For the given graph, following 2 different topological orderings are possible-

A B C D E

A B D C E

PRIM'S ALGORITHM-

Prim's Algorithm is a famous greedy algorithm.

It is used for finding the Minimum Spanning Tree (MST) of a given graph.

To apply Prim's algorithm, the given graph must be weighted, connected and undirected.

PRIM'S ALGORITHM IMPLEMENTATION-

The implementation of Prim's Algorithm is explained in the following steps-

Step-01:

Randomly choose any vertex.

The vertex connecting to the edge having least weight is usually selected.

Step-02:

Find all the edges that connect the tree to new vertices.

Find the least weight edge among those edges and include it in the existing tree.

If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step-03:

Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

Prim's Algorithm Time Complexity-

Worst case time complexity of Prim's Algorithm is-

O(ElogV) using binary heap

O(E + VlogV) using Fibonacci heap

Time Complexity Analysis

If adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in O(V + E) time.

We traverse all the vertices of graph using breadth first search and use a min heap for storing the vertices not yet included in the MST.

To get the minimum weight edge, we use min heap as a priority queue.

Min heap operations like extracting minimum element and decreasing key value takes O(logV) time.

So, overall time complexity

= O(E + V) x O(logV)

= O((E + V)logV)

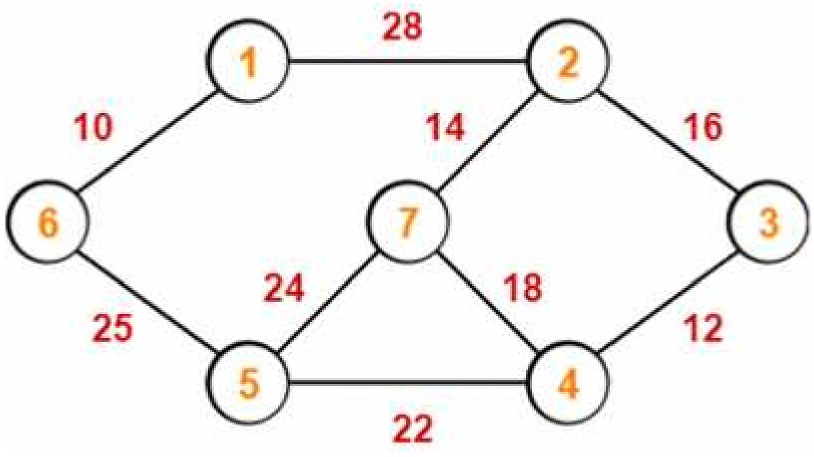
= O(ElogV)

This time complexity can be improved and reduced to O(E + VlogV) using Fibonacci heap.

PRACTICE PROBLEMS BASED ON PRIM’S ALGORITHM-

Problem-01:

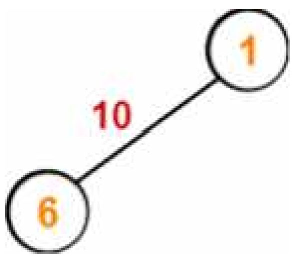
Construct the minimum spanning tree (MST) for the given graph using Prim’s Algorithm-



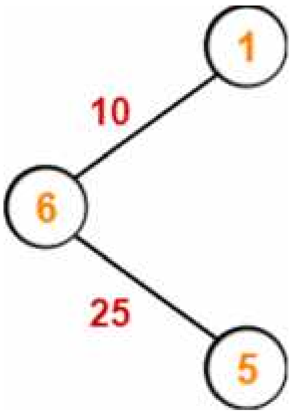
Solution-

The above discussed steps are followed to find the minimum cost spanning tree using Prim’s Algorithm-

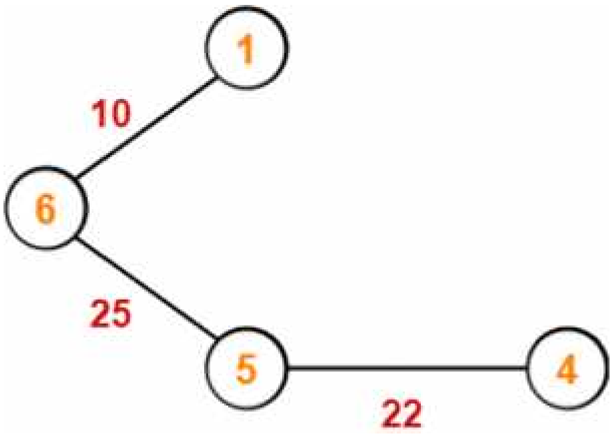
Step-01:



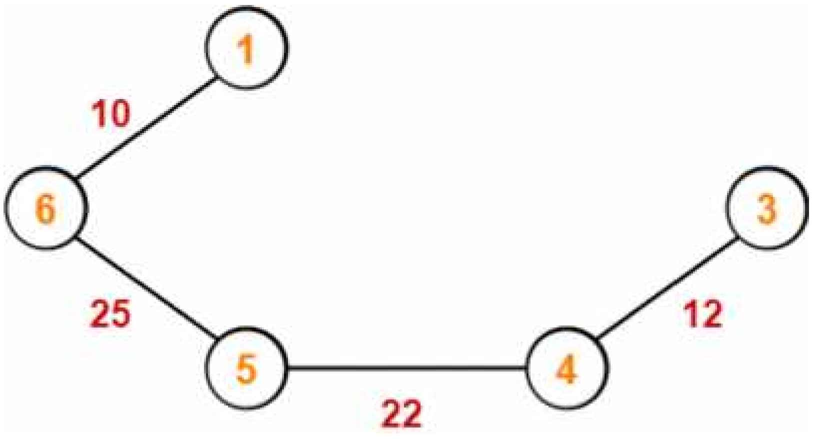
Step-02:



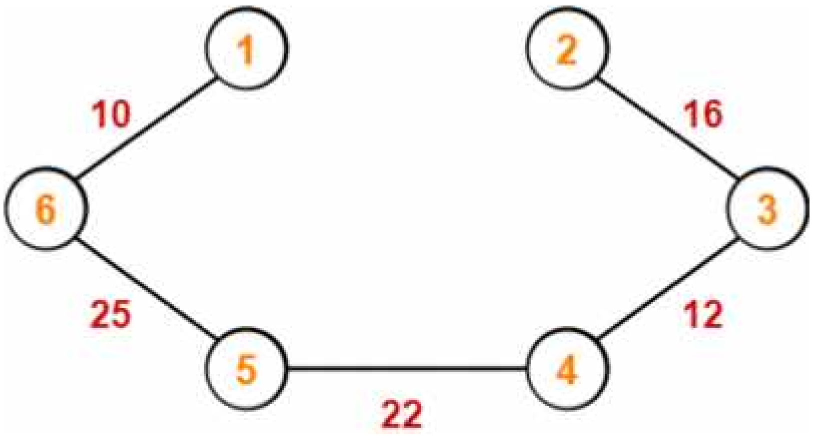
Step-03:



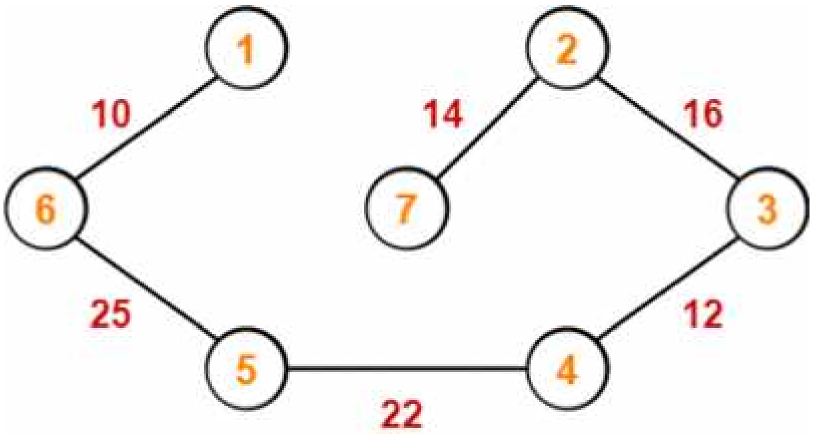
Step-04:



Step-05:



Step-06:



Since all the vertices have been included in the MST, so we stop.

Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

KRUSKAL'S ALGORITHM-

Kruskal's Algorithm is a famous greedy algorithm.

It is used for finding the Minimum Spanning Tree (MST) of a given graph.

To apply Kruskal's algorithm, the given graph must be weighted, connected and undirected.

Kruskal's Algorithm Implementation-

The implementation of Kruskal's Algorithm is explained in the following steps-

Step-01:

Sort all the edges from low weight to high weight.

Step-02:

Take the edge with the lowest weight and use it to connect the vertices of graph.

If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

Step-03:

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

Thumb Rule to Remember

The above steps may be reduced to the following thumb rule-

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

Kruskal’s Algorithm Time Complexity-

Worst case time complexity of Kruskal’s Algorithm

= O(ElogV) or O(ElogE)

Analysis-

The edges are maintained as min heap.

The next edge can be obtained in O(logE) time if 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.

Special Case-

If the edges are already sorted, then there is no need to construct min heap.

So, deletion from min heap time is saved.

In this case, time complexity of Kruskal’s Algorithm = O(E + V)

PRACTICE PROBLEMS BASED ON KRUSKAL’S ALGORITHM-

Problem-01:

Construct the minimum spanning tree (MST) for the given graph using Kruskal’s Algorithm-

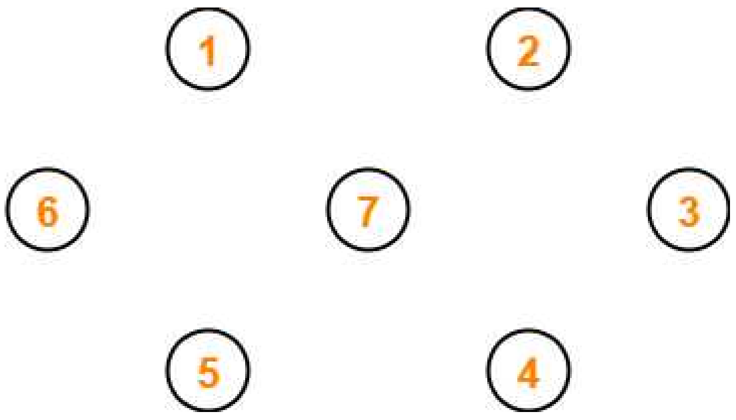
Solution-

To construct MST using Kruskal’s Algorithm,

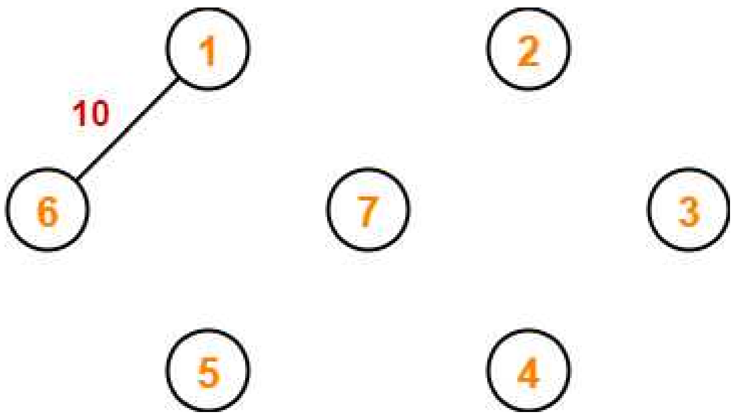
Simply draw all the vertices on the paper.

Connect these vertices using edges with minimum weights such that no cycle gets formed.

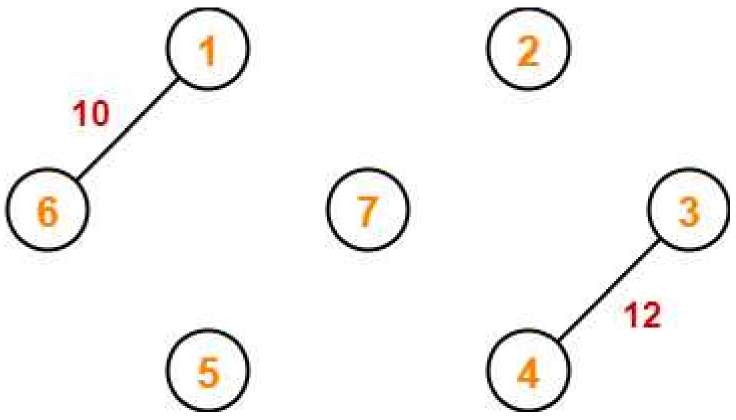
Step-01:



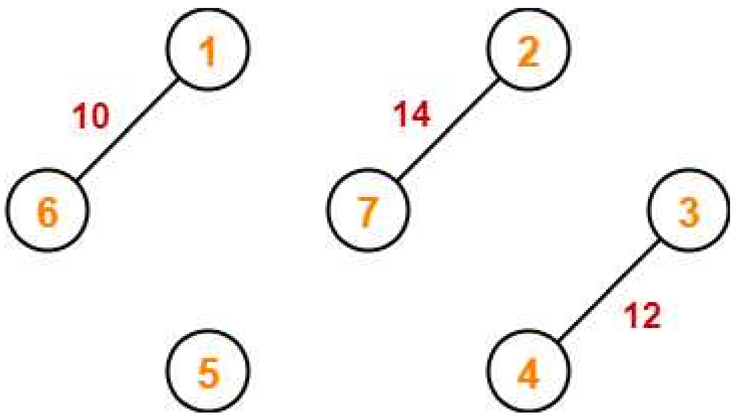
Step-02:



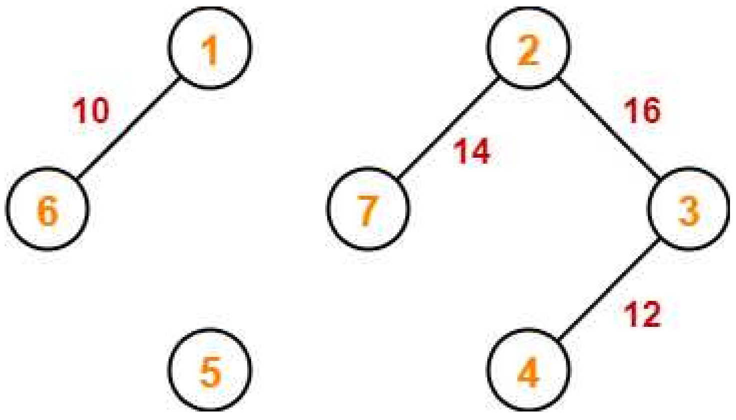
Step-03:



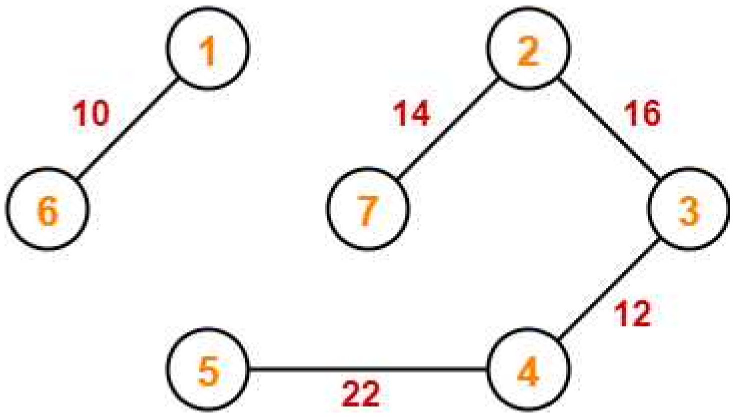
Step-04:



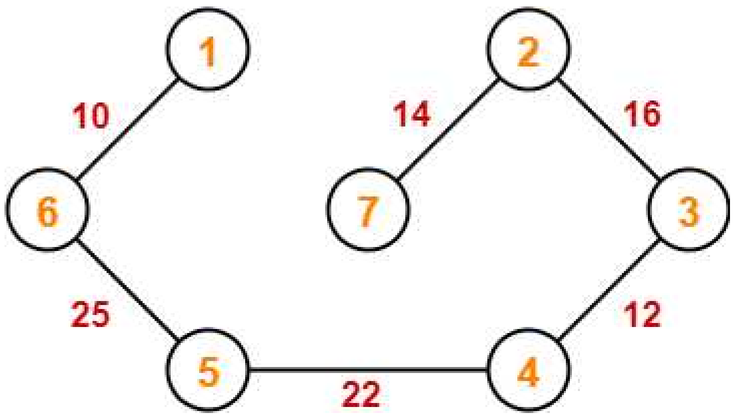
Step-05:



Step-06:



Step-07:



Since all the vertices have been connected / included in the MST, so we stop.

Weight of the MST

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

SHORTEST PATH PROBLEM | SHORTEST PATH ALGORITHMS | EXAMPLES

SHORTEST PATH PROBLEM

- Shortest path problem is a problem of finding the shortest path(s) between vertices of a given graph.
- Shortest path between two vertices is a path that has the least cost as compared to all other existing paths.

Shortest Path Algorithms-

Shortest path algorithms are a family of algorithms used for solving the shortest path problem.

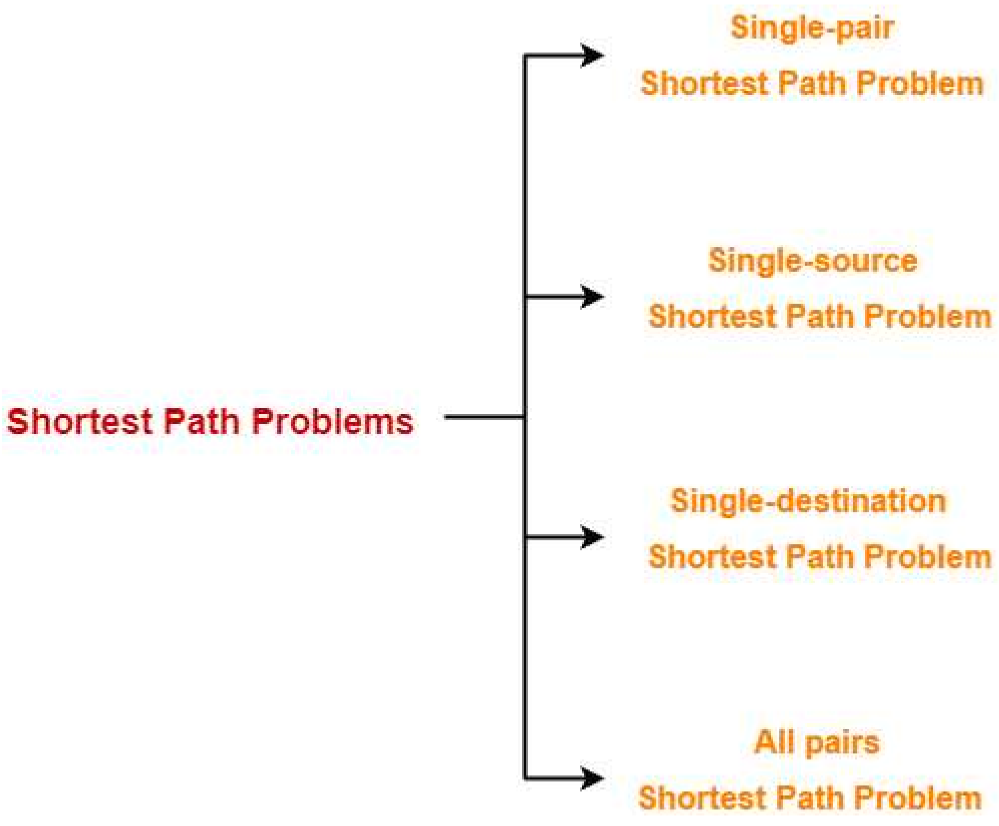
Applications-

Shortest path algorithms have a wide range of applications such as in-

- Google Maps
- Road Networks
- Logistics Research

Types of Shortest Path Problem-

Various types of shortest path problem are-


Shortest Path Problems
Single-pair Shortest Path Problem
Single-source Shortest Path Problem
Single-destination Shortest Path Problem
All pairs Shortest Path Problem

- Single-pair shortest path problem
- Single-source shortest path problem
- Single-destination shortest path problem
- All pairs shortest path problem

Single-Pair Shortest Path Problem-

It is a shortest path problem where the shortest path between a given pair of vertices is computed.

A* Search Algorithm is a famous algorithm used for solving single-pair shortest path problem.

Single-Source Shortest Path Problem-

It is a shortest path problem where the shortest path from a given source vertex to all other remaining vertices is computed.

Dijkstra’s Algorithm and Bellman Ford Algorithm are the famous algorithms used for solving single-source shortest path problem.

Single-Destination Shortest Path Problem-

It is a shortest path problem where the shortest path from all the vertices to a single destination vertex is computed.

By reversing the direction of each edge in the graph, this problem reduces to single-source shortest path problem.

Dijkstra’s Algorithm is a famous algorithm adapted for solving single-destination shortest path problem.

All Pairs Shortest Path Problem-

It is a shortest path problem where the shortest path between every pair of vertices is computed.

Floyd-Warshall Algorithm and Johnson’s Algorithm are the famous algorithms used for solving All pairs shortest path problem.

Dijkstra Algorithm | Example | Time Complexity

Dijkstra Algorithm-

Dijkstra Algorithm is a very famous greedy algorithm.

It is used for solving the single source shortest path problem.

It computes the shortest path from one particular source node to all other remaining nodes of the graph.

Conditions-

It is important to note the following points regarding Dijkstra Algorithm-

Dijkstra algorithm works only for connected graphs.

Dijkstra algorithm works only for those graphs that do not contain any negative weight edge.

The actual Dijkstra algorithm does not output the shortest paths.

It only provides the value or cost of the shortest paths.

By making minor modifications in the actual algorithm, the shortest paths can be easily obtained.

Dijkstra algorithm works for directed as well as undirected graphs.

Dijkstra Algorithm-

```
dist[S] ← 0                         // The distance to source vertex is set to 0
Π[S] ← NIL                          // The predecessor of source vertex is set as NIL
for all v ∈ V - {S}                 // For all other vertices
do dist[v] ← ∞                      // All other distances are set to ∞
Π[v] ← NIL                          // The predecessor of all other vertices is set as NIL
```

```
S ← ∅                                // The set of vertices that have been visited 'S' is initially empty
Q ← V                                // The queue 'Q' initially contains all the vertices
while Q ≠ ∅                          // While loop executes till the queue is not empty
do u ← mindistance (Q, dist)         // A vertex from Q with the least distance is selected
S ← S ∪ {u}                          // Vertex 'u' is added to 'S' list of vertices that have been visited
for all v ∈ neighbors[u]             // For all the neighboring vertices of vertex 'u'
do if dist[v] > dist[u] + w(u,v)     // if any new shortest path is discovered
        then dist[v] ← dist[u] + w(u,v)   // The new value of the shortest path is selected
```

Implementation-

The implementation of above Dijkstra Algorithm is explained in the following steps-

Step-01:

In the first step. two sets are defined-

One set contains all those vertices which have been included in the shortest path tree.

In the beginning, this set is empty.

Other set contains all those vertices which are still left to be included in the shortest path tree.

In the beginning, this set contains all the vertices of the given graph.

Step-02:

For each vertex of the given graph, two variables are defined as-

Π[v] which denotes the predecessor of vertex ‘v’

d[v] which denotes the shortest path estimate of vertex ‘v’ from the source vertex.

Initially, the value of these variables is set as-

The value of variable 'Π' for each vertex is set to NIL i.e. Π[v] = NIL

The value of variable 'd' for source vertex is set to 0 i.e. d[S] = 0

The value of variable 'd' for remaining vertices is set to ∞ i.e. d[v] = ∞

Step-03:

The following procedure is repeated until all the vertices of the graph are processed-

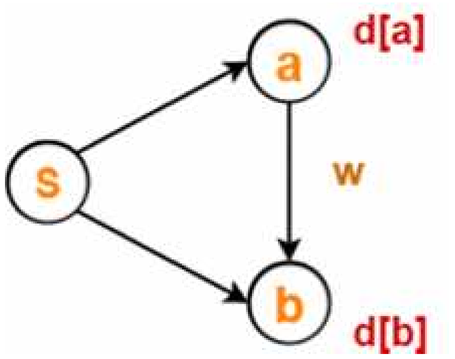
Among unprocessed vertices, a vertex with minimum value of variable 'd' is chosen.

Its outgoing edges are relaxed.

After relaxing the edges for that vertex, the sets created in step-01 are updated.

What is Edge Relaxation?

Consider the edge (a,b) in the following graph-



Here, d[a] and d[b] denotes the shortest path estimate for vertices a and b respectively from the source vertex 'S'.

Now,

If d[a] + w < d[b]

then d[b] = d[a] + w and Π[b] = a

This is called as edge relaxation.

Time Complexity Analysis-

Case-01:

This case is valid when-

The given graph G is represented as an adjacency matrix.

Priority queue Q is represented as an unordered list.

Here,

A[i,j] stores the information about edge (i,j).

Time taken for selecting i with the smallest dist is O(V).

For each neighbor of i, time taken for updating dist[j] is O(1) and there will be maximum V neighbors.

Time taken for each iteration of the loop is O(V) and one vertex is deleted from Q.

Thus, total time complexity becomes O(V2).

Case-02:

This case is valid when-

The given graph G is represented as an adjacency list.

Priority queue Q is represented as a binary heap.

Here,

With adjacency list representation, all vertices of the graph can be traversed using BFS in O(V+E) time.

In min heap, operations like extract-min and decrease-key value takes O(logV) time.

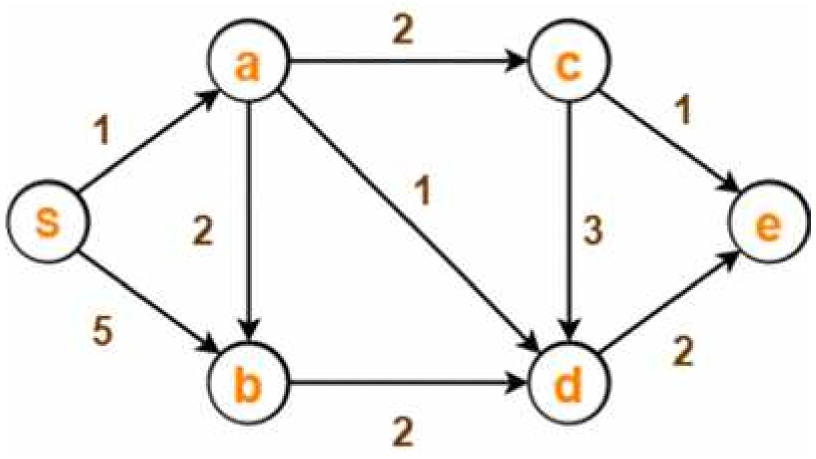
So, overall time complexity becomes O(E+V) x O(logV) which is O((E + V) x logV) = O(ElogV)

This time complexity can be reduced to O(E+VlogV) using Fibonacci heap.

PRACTICE PROBLEM BASED ON DIJKSTRA ALGORITHM-

Problem-

Using Dijkstra's Algorithm, find the shortest distance from source vertex 'S' to remaining vertices in the following graph-



Also, write the order in which the vertices are visited.

Solution-

Step-01:

The following two sets are created-

Unvisited set : {S , a , b , c , d , e}

Visited set : { }

Step-02:

The two variables Π and d are created for each vertex and initialized as-

Π[S] = Π[a] = Π[b] = Π[c] = Π[d] = Π[e] = NIL

d[S] = 0

d[a] = d[b] = d[c] = d[d] = d[e] = ∞

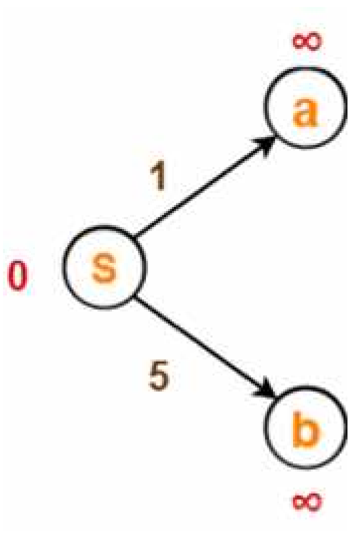
Step-03:

Vertex ‘S’ is chosen.

This is because shortest path estimate for vertex ‘S’ is least.

The outgoing edges of vertex ‘S’ are relaxed.

Before Edge Relaxation-



Now,

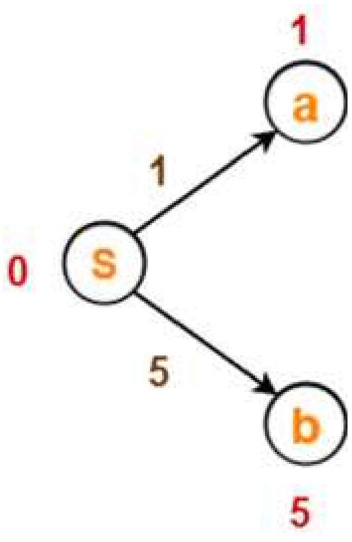
d[S] + 1 = 0 + 1 = 1 < ∞

∴ d[a] = 1 and Π[a] = S

d[S] + 5 = 0 + 5 = 5 < ∞

∴ d[b] = 5 and Π[b] = S

After edge relaxation, our shortest path tree is-



Now, the sets are updated as-

Unvisited set : {a , b , c , d , e}

Visited set : {S}

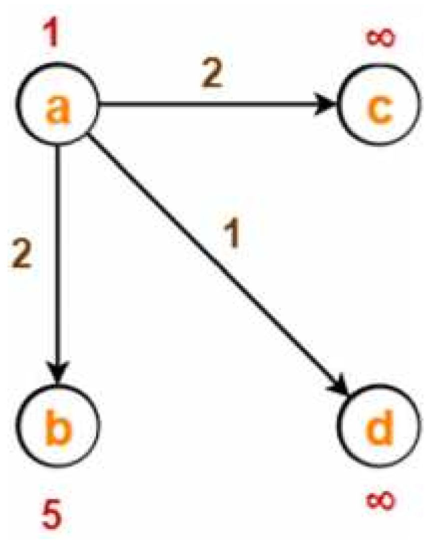
Step-04:

Vertex 'a' is chosen.

This is because shortest path estimate for vertex 'a' is least.

The outgoing edges of vertex 'a' are relaxed.

Before Edge Relaxation-



Now,

d[a] + 2 = 1 + 2 = 3 < ∞

∴ d[c] = 3 and Π[c] = a

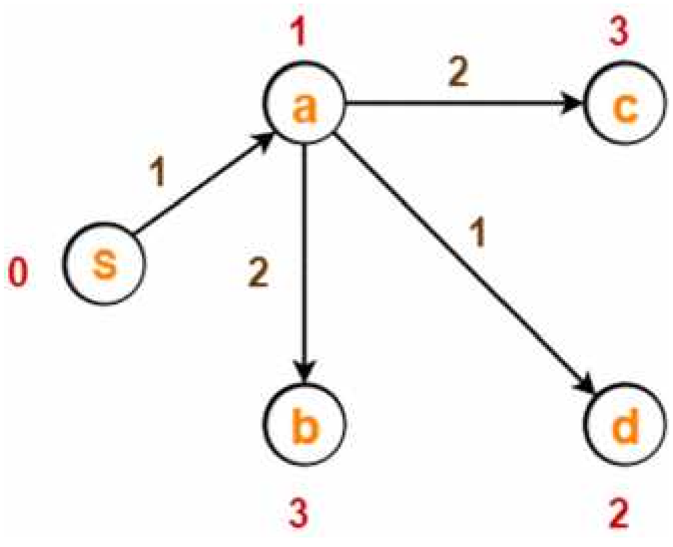
d[a] + 1 = 1 + 1 = 2 < ∞

∴ d[d] = 2 and Π[d] = a

d[b] + 2 = 1 + 2 = 3 < 5

∴ d[b] = 3 and Π[b] = a

After edge relaxation, our shortest path tree is-



Now, the sets are updated as-

Unvisited set : {b , c , d , e}

Visited set : {S , a}

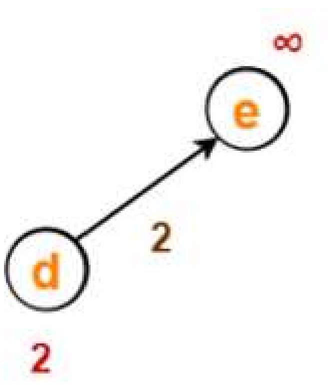
Step-05:

Vertex ‘d’ is chosen.

This is because shortest path estimate for vertex ‘d’ is least.

The outgoing edges of vertex ‘d’ are relaxed.

Before Edge Relaxation-

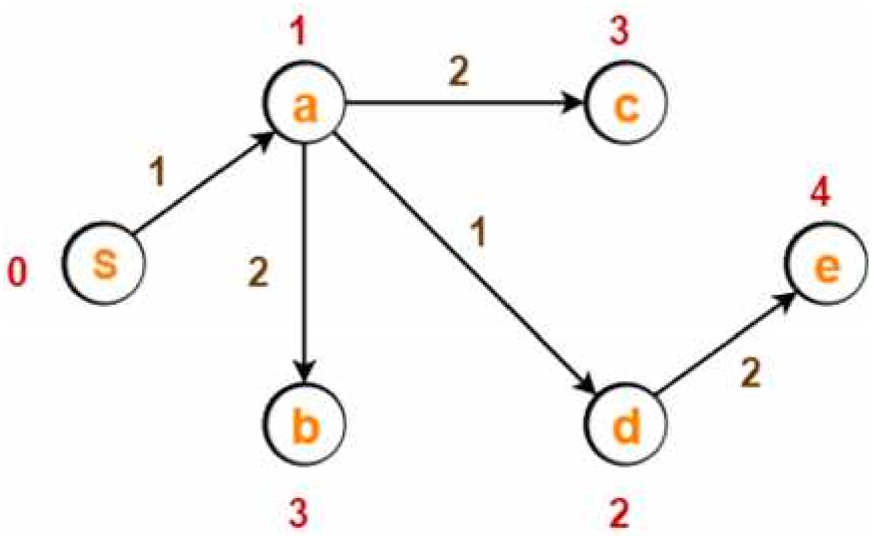


Now,

d[d] + 2 = 2 + 2 = 4 < ∞

d[e] = 4 and Π[e] = d

After edge relaxation, our shortest path tree is-



Now, the sets are updated as-

Unvisited set : {b , c , e}

Visited set : {S , a , d}

Step-06:

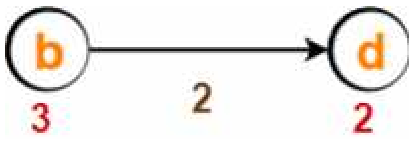
Vertex ‘b’ is chosen.

This is because shortest path estimate for vertex ‘b’ is least.

Vertex ‘c’ may also be chosen since for both the vertices, shortest path estimate is least.

The outgoing edges of vertex ‘b’ are relaxed.

Before Edge Relaxation-



Now,

d[b] + 2 = 3 + 2 = 5 > 2

No change

After edge relaxation, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-

Unvisited set : {c , e}

Visited set : {S , a , d , b}

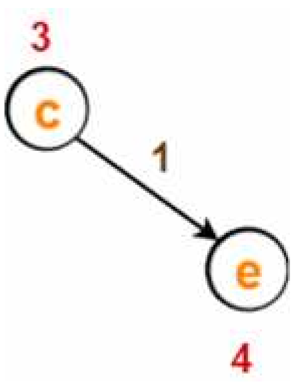
Step-07:

Vertex ‘c’ is chosen.

This is because shortest path estimate for vertex ‘c’ is least.

The outgoing edges of vertex ‘c’ are relaxed.

Before Edge Relaxation-



Now,

d[c] + 1 = 3 + 1 = 4 = 4

No change

After edge relaxation, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-

Unvisited set : {e}

Visited set : {S , a , d , b , c}

Step-08:

Vertex ‘e’ is chosen.

This is because shortest path estimate for vertex ‘e’ is least.

The outgoing edges of vertex ‘e’ are relaxed.

There are no outgoing edges for vertex ‘e’.

So, our shortest path tree remains the same as in Step-05.

Now, the sets are updated as-

Unvisited set : { }

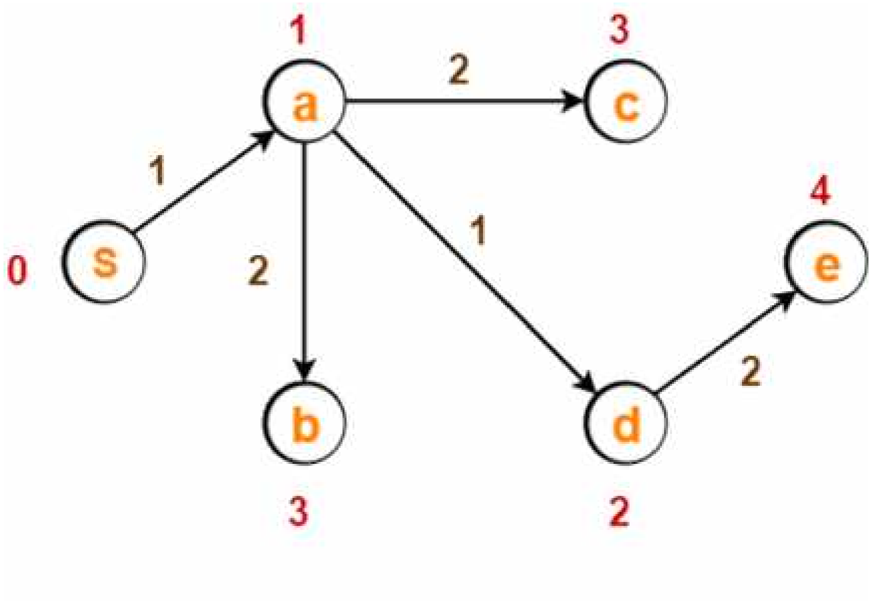
Visited set : {S , a , d , b , c , e}

Now,

All vertices of the graph are processed.

Our final shortest path tree is as shown below.

It represents the shortest path from source vertex 'S' to all other remaining vertices.



Shortest Path Tree

The order in which all the vertices are processed is :

S , a , d , b , c , e.

FLOYD WARSHALL ALGORITHM | EXAMPLE | TIME COMPLEXITY

FLOYD WARSHALL ALGORITHM-

Floyd Warshall Algorithm is a famous algorithm.

It is used to solve All Pairs Shortest Path Problem.

It computes the shortest path between every pair of vertices of the given graph.

Floyd Warshall Algorithm is an example of dynamic programming approach.

ADVANTAGES-

Floyd Warshall Algorithm has the following main advantages-

- It is extremely simple.
- It is easy to implement.

Algorithm-

Floyd Warshall Algorithm is as shown below-

```
Create a |V| x |V| matrix            // It represents the distance between every pair of vertices as given

For each cell (i,j) in M do-

if i = = j

M[ i ][ j ] = 0              // For all diagonal elements, value = 0

if (i , j) is an edge in E

M[ i ][ j ] = weight(i,j)      // If there exists a direct edge between the vertices, value = weight of edge

else

M[ i ][ j ] = infinity        // If there is no direct edge between the vertices, value = ∞
```

```
for k from 1 to |V|
for i from 1 to |V|
for j from 1 to |V|
if M[ i ][ j ] > M[ i ][ k ] + M[ k ][ j ]
M[ i ][ j ] = M[ i ][ k ] + M[ k ][ j ]
```

Time Complexity-

Floyd Warshall Algorithm consists of three loops over all the nodes.

The inner most loop consists of only constant complexity operations.

Hence, the asymptotic complexity of Floyd Warshall algorithm is O(n3).

Here, n is the number of nodes in the given graph.

When Floyd Warshall Algorithm Is Used?

Floyd Warshall Algorithm is best suited for dense graphs.

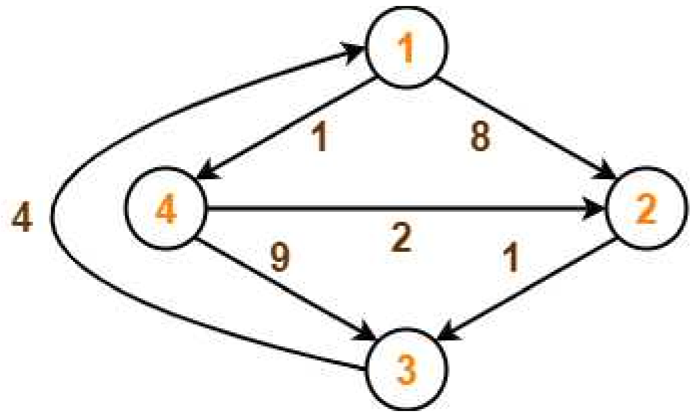
This is because its complexity depends only on the number of vertices in the given graph.

For sparse graphs, Johnson's Algorithm is more suitable.

PRACTICE PROBLEM BASED ON FLOYD WARSHALL ALGORITHM-

Problem-

Consider the following directed weighted graph-



Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.

Solution-

Step-01:

Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.

In the given graph, there are neither self edges nor parallel edges.

Step-02:

Write the initial distance matrix.

It represents the distance between every pair of vertices in the form of given weights.

For diagonal elements (representing self-loops), distance value = 0.

For vertices having a direct edge between them, distance value = weight of that edge.

For vertices having no direct edge between them, distance value = ∞.

Initial distance matrix for the given graph is-

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

Step-03:

Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{array}$$

$$D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

$$D_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

BELLMAN FORD ALGORITHM

What is Bellman-Ford Algorithm?

Bellman-Ford algorithm is used to find the shortest path from the source vertex to every vertex in a weighted graph. Unlike Dijkstra's algorithm, the bellman ford algorithm can also find the shortest distance to every vertex in the weighted graph even with the negative edges. The only difference between the Dijkstra algorithm and the bellman ford algorithm is that Dijkstra's algorithm just visits the neighbour vertex in each iteration but the bellman ford algorithm visits each vertex through each edge in every iteration.

Apart from Bellman-Ford Algorithm and Dijkstra's Algorithm, Floyd Warshall Algorithm is also the shortest path algorithm. But Bellman-Ford algorithm is used to compute the shortest path from the single source vertex to all other vertices whereas Floyd-Warshall algorithms compute the shortest path from each node to every other node.

Bellman ford algorithm follows the dynamic programming approach by overestimating the length of the path from the starting vertex to all other vertices. And then it starts relaxing the estimates by discovering the new paths which are shorter than the previous ones. This process is followed by all the vertices for N-1 times for finding the optimized result.

Algorithm

```
Begin
  count := 1
  maxEdge := a * (a - 1) / 2     //a is number of vertices
  for all vertices n of the graph, do
    distance[n] := ∞
    pred[n] := ?
  done
  distance[source] := 0
  eCount := number of edges in the graph
  create edge list named edgeList
  while count < a, do
    for k := 0 to eCount, do
      if distance[edgeList[k].n] > distance[edgeList[k].m] + (cost[m,n] for edge k)
distance[edgeList[k].n] > distance[edgeList[k].m] + (cost[m,n] for edge k)
pred[edgeList[k].n] := edgeList[k].m
    done
  done
  count := count + 1
  for all vertices k in the graph, do
    if distance[edgeList[k].n] > distance[edgeList[k].m] + (cost[m,n] for edge k),
      then return true
  done
  return false
End
```

1) Initializing the source vertex with distance to itself as 0 and all other vertices as infinity. Creating the array with size N

2) Calculate the shortest distance from the source vertex. Following this process for N-1 times where N is the number of vertices in the graph

For relaxing the path lengths for the vertices for each edge m-n:

if distance[n] > distance[m] + weight of edge mn, then

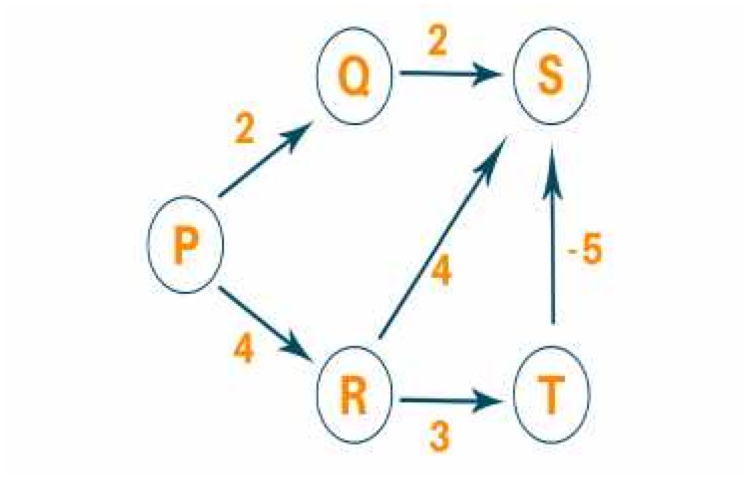
distance[n] = distance[m] + weight of edge mn

3) Even after minimizing the path lengths for each vertex after N-1 times, if we can still relax the path length where distance[n] > distance[m] + weight of edge mn then we can say that the graph contains the negative edge cycle.

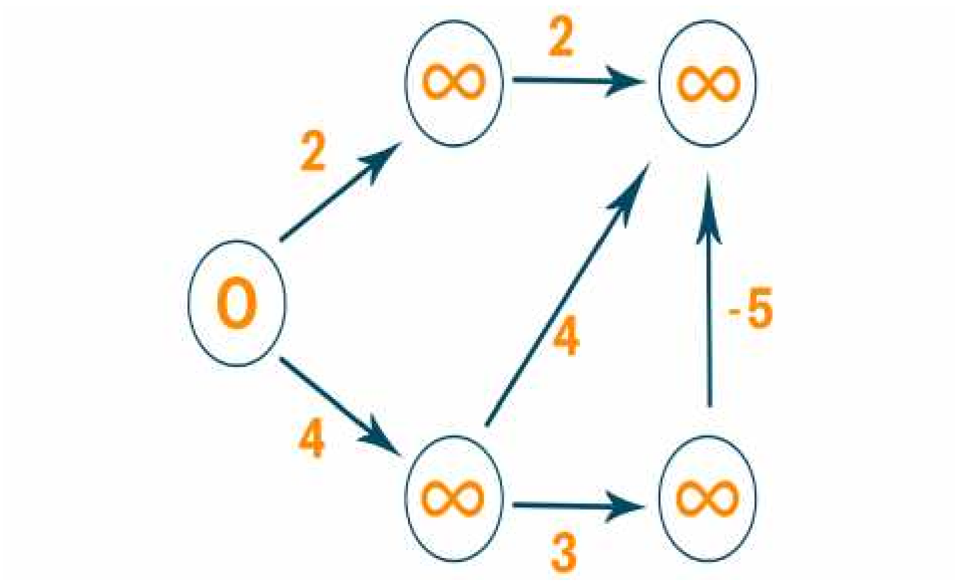
Therefore, the only limitation of the bellman ford algorithm is that it does not work with a graph having a negative edge cycle.

Example:

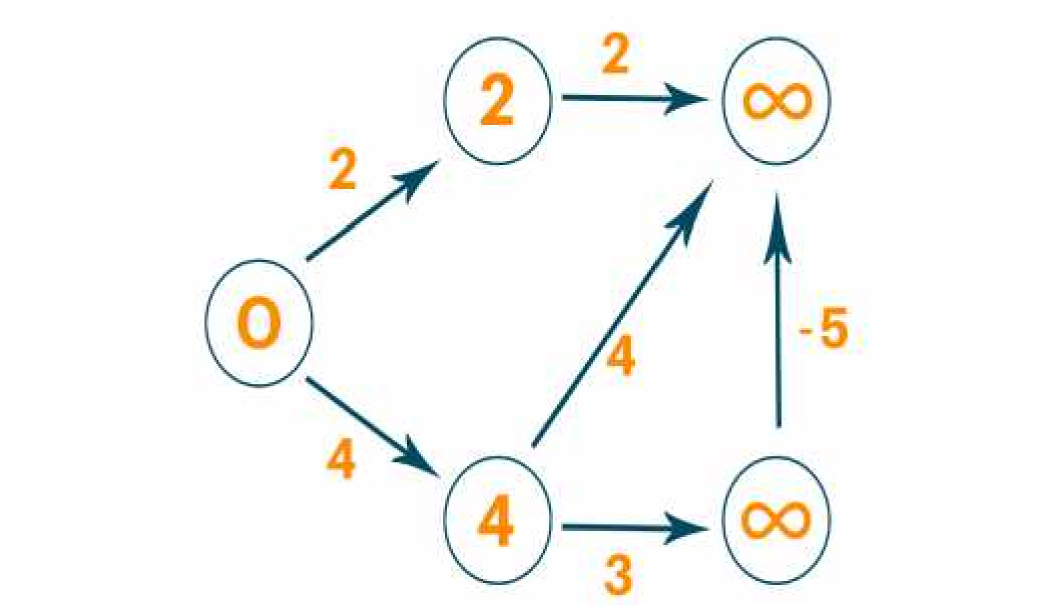
Consider the following weighted graph



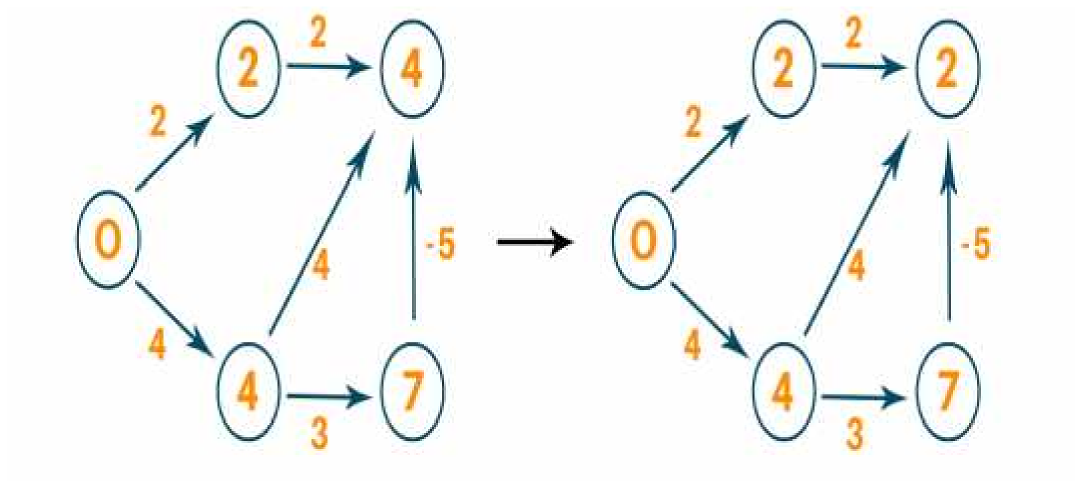
Select the source vertex with path value 0 and all other vertices as infinity.



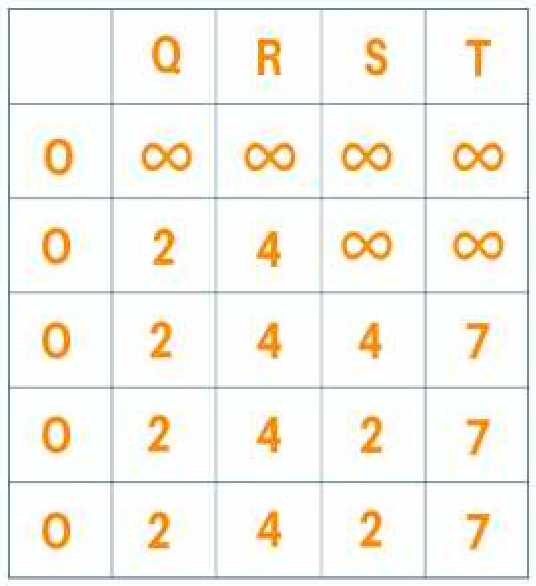
Visit the neighbouring edge from the source vertex and relax the path length of the neighbouring vertex if the new calculated path length is smaller than the previous path length



This process must be followed N-1 times where N is the total number of vertices. This is because in the worst-case scenario any vertex's path length can be changed to an even smaller path length for N times.



Therefore after N-1 iterations, we find our new path lengths and we can check if the graph has a negative cycle or not.



	Q	R	S	T
0	∞	∞	∞	∞
0	2	4	∞	∞
0	2	4	4	7
0	2	4	2	7
0	2	4	2	7

Time Complexity

The time complexity of the bellman ford algorithm for the best case is O(E) while average-case and worst-case time complexity are O(NE) where N is the number of vertices and E is the total edges to be relaxed. Also, the space complexity of the bellman ford algorithm is O(N) because the size of the array is N.

Applications

Used for distance-routing protocol helping in routing the data packets on the network

Used in internet gateway routing protocol

Used in routing information protocol