Graphs 2

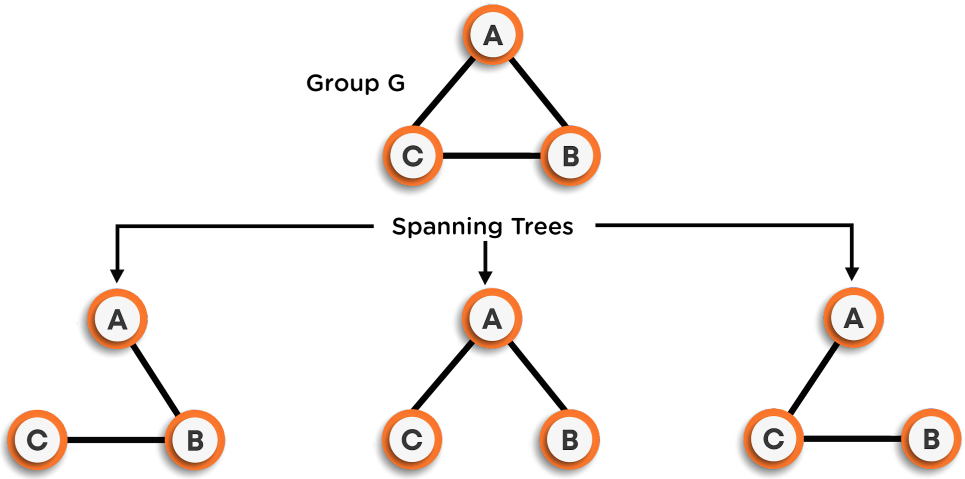
MST & Kruskal's introduction

As discussed earlier, a tree is a graph, which:

- Is always connected.
- Contains no cycle.

If we are given an undirected and connected graph, a **spanning tree** means a tree that contains all the vertices of the same. For a given graph, we can have multiple spanning trees.

Refer to the example below for a better understanding of spanning trees.



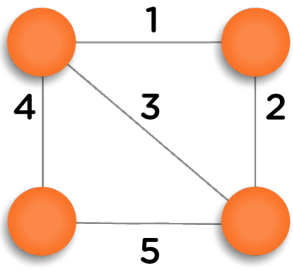
If there are *n* vertices and *e* edges in the graph, then any spanning tree corresponding to that graph contains *n* vertices and *n-1* edges.

Properties of spanning trees:

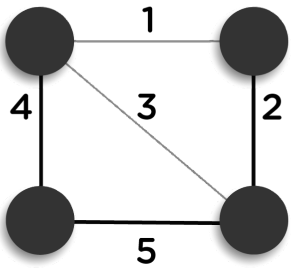
- A connected and undirected graph can have more than one spanning tree.
- Spanning tree is free of loops, i.e., it is acyclic.
- Removing any one of the edges will make the graph disconnected.
- Adding an extra edge to the spanning tree will create a loop in the graph.

Minimum Spanning Tree(MST) is a spanning tree with weighted edges.

In a weighted graph, the MST is a spanning tree with minimum weight than all other spanning trees of that graph. Refer to the image below for better understanding.

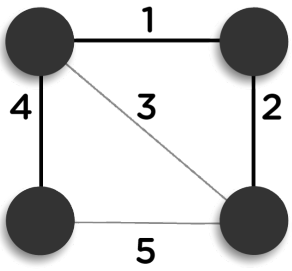


Undirected graph



Spanning tree

Cost=11(=4+5+2)



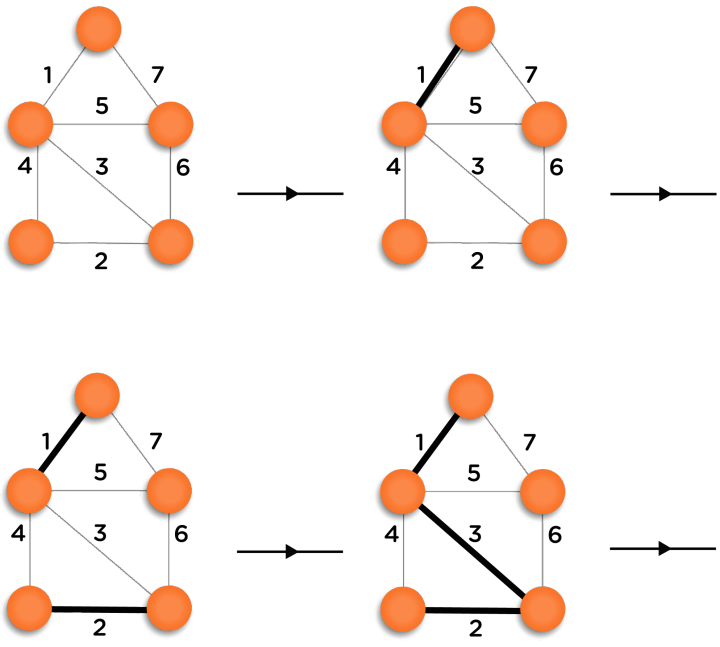
Minimum Spanning tree

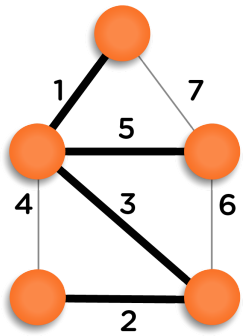
Cost=7(=4+1+2)

Kruskal's Algorithm:

This algorithm is used to find MST for the given graph. It builds the spanning tree by adding edges one at a time. We start by picking the edge with minimum weight, adding that edge into the MST, and increasing the count of edges in the spanning tree by one. Now, we will be picking the minimum weighted edge by excluding the already chosen ones and correspondingly increasing the count. While choosing the edge, we will also make sure that the graph remains acyclic after including the same. This process will continue until the count of edges in the MST reaches *n-1*. Ultimately, the graph obtained will be MST.

Refer to the example below for a better understanding of the same.





This is the final MST obtained using Kruskal's algorithm. It can be checked manually that the final graph is the MST for the given graph.

Cycle Detection

While inserting a new edge in the MST, we have to check if introducing that edge makes the MST cyclic or not. If not, then we can include that edge, otherwise not.

Now, let's figure out a way to detect the cycle in a graph. The following are the possible cases:

- By including an edge between the nodes A and B, if both the nodes A and B are not present in the graph, then it is safe to include that edge as including it, will not bring a cycle to the graph.
- Out of two vertices, if any of them has not been visited (or not present in the MST), then that vertex can also be included in the MST.
- If both the vertices are already present in the graph, they can introduce a cycle in the MST. It means we can't use this method to detect the presence of the cycle.

Let's think of a better approach. We have already solved the **hasPath** question in the previous module, which returns true if there is a path present between two vertices v1 and v2, otherwise false.

Now, before adding an edge to the MST, we will check if a path between two vertices of that edge already exists in the MST or not. If not, then it is safe to add that edge to the MST.

As discussed in previous lectures, the time complexity of the **hasPath** function is O(E+V), where E is the number of edges in the graph and, V is the number of vertices. So, for (n-1) edges, this function will run (n-1) times, leading to bad time complexity, as in the worst case, $E = V^2$.

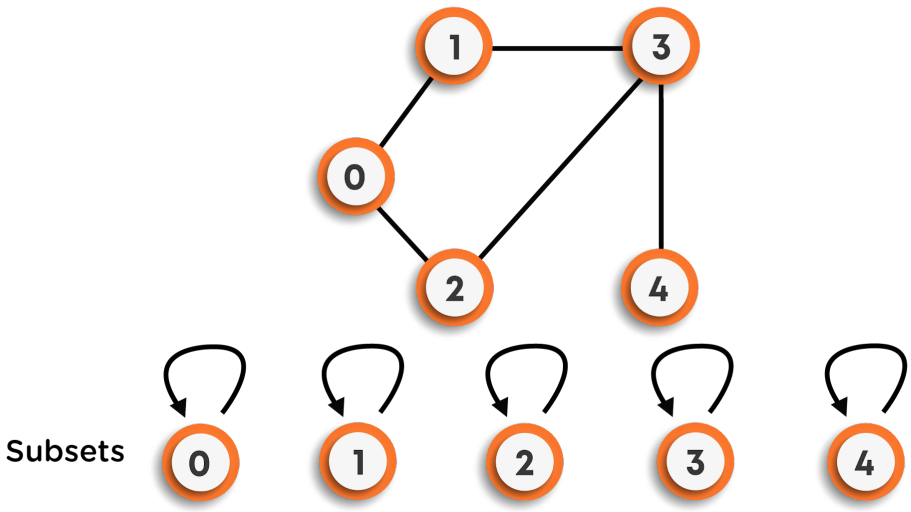
Now, moving on to a better approach for cycle detection in the graph.

Union-find algorithm:

Before adding any edge to the graph, we will check if the two vertices of the edge lie in the same component of MST or not. If not, then it is safe to add that edge to the MST.

Following the steps of the algorithm:

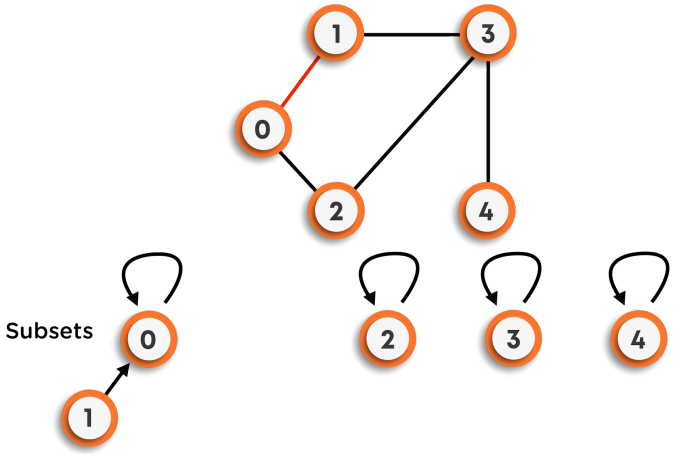
- We will assume that initially, the total number of disjoint sets is equal to the number of vertices in the graph starting from 0 to n-1.
- We will maintain a parent array specifying the parent vertex of each of the vertices of the graph. Initially, as each vertex belongs to the different disjoint set (connected component), hence each vertex will be its own parent.



Initially all parent pointers are pointing to self
means only one element in each subset

- Now, before inserting any edge into the MST, we will check the parent of the vertices. If their parent vertices are equal, they belong to the same connected component; hence it is unsafe to add that edge. Otherwise, we can add that edge into the MST, and simultaneously update the parent array so that they belong to the same component(Refer to the code on how to do so).

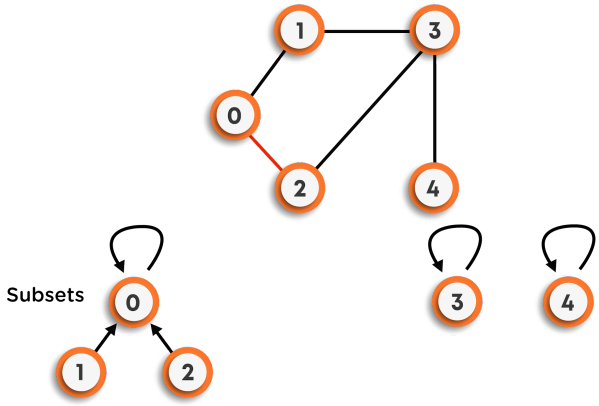
  Look at the following example, for better understanding:



Edge 0-1

Find: 0 belongs to subset 0 and 1 belongs to subset 1 so they are in different subsets.

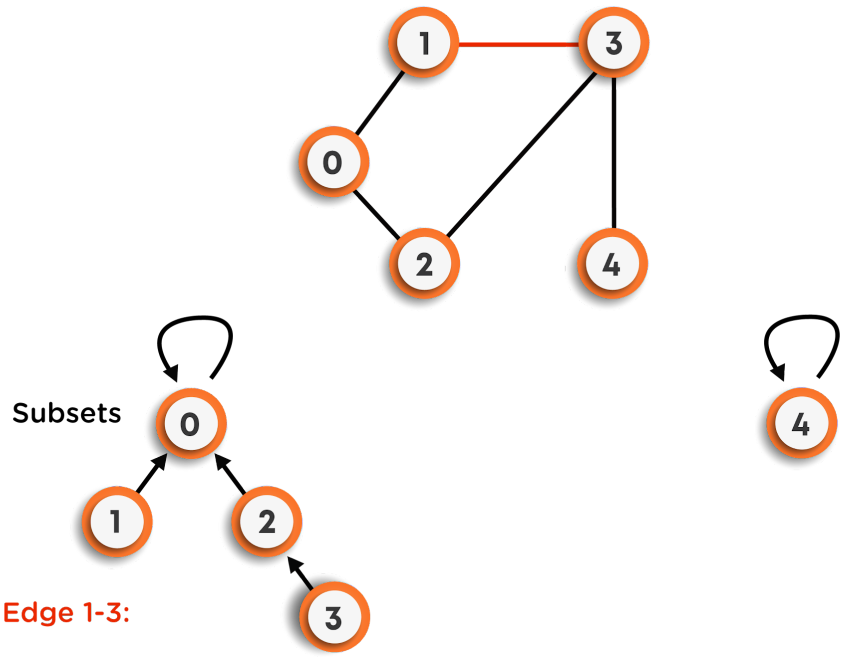
Union: Make 0 as the parent of 1, Updated set is {0,1}. 0 is the set representative since 0 is parent for itself.



Edge 0-2:

Find: 0 belongs to subset 0 and 2 belongs to subset 2 so they are in different subsets.

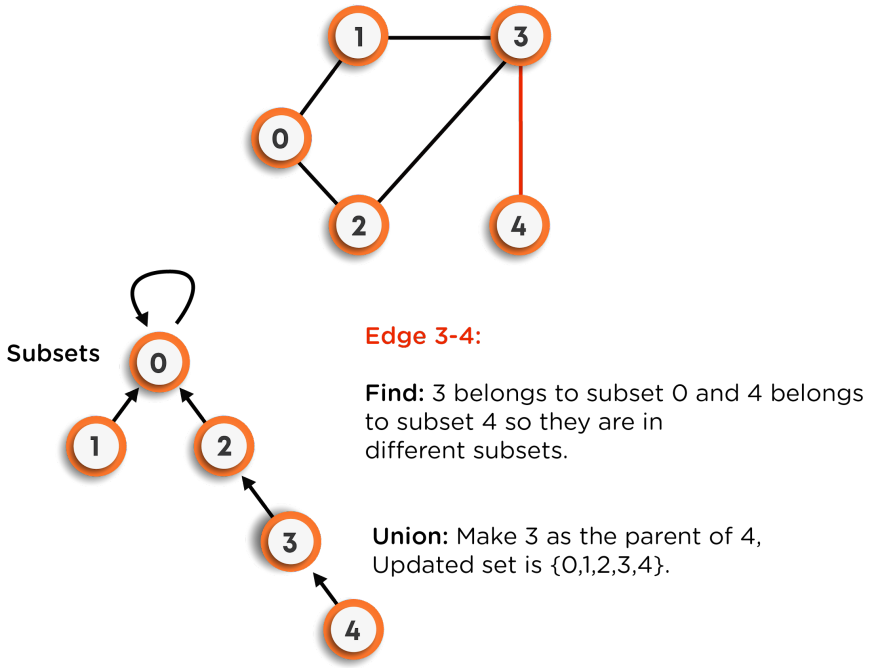
Union: Make 0 as the parent of 2, Updated set is {0,1,2}. 0 is the set representative since 0 is parent for itself.



Edge 1-3:

Find: 1 belongs to subset 0 and 3 belongs to subset 3 so they are in different subsets.

Union: Make 1 as the parent of 3, Updated set is {0,1,2,3}. 0 is the set representative since 0 is parent for itself.



Note: While finding the parent of the vertex, we will be finding the topmost parent(Oldest Ancestor). For example: suppose, the vertex 0 and the vertex 1 were connected, where the parent of 0 is 1, and the parent of 1 is 1. Now, while determining the parent of the vertex 0, we will visit the parent array and check the vertex at index 0. In our case, it is 1. Now we will go to index 1 and check the parent of index 1, which is also 1. Hence, we can't go any further as the index is the parent of itself. This way, we will be determining the parent of any vertex.

The time complexity of the union-find algorithm becomes O(V) for each vertex in the worst case due to skewed-tree formation, where V is the number of vertices in the graph. Here, we can see that time complexity for cycle detection has significantly improved compared to the previous approach.

Kruskal's algorithm: Implementation

Till now, we have studied the logic behind Kruskal's algorithm for finding MST. Now, let's discuss how to implement it in code.

Consider the code below and follow the comments for a better understanding.

```
#include <iostream>
#include <algorithm>
using namespace std;

// Class that store values for each vertex
class Edge {
    public:
        int source;
        int dest;
        int weight;
};
// Comparator function used to sort edges
bool compare(Edge e1, Edge e2) {
    // Edges will sorted in order of their weights
    return e1.weight < e2.weight;
}
// Function to find the parent of a vertex
int findParent(int v, int *parent) {
    // Base case, when a vertex is parent of itself
    if (parent[v] == v) {
        return v;
    }
    // Recursively called to find the topmost parent of the vertex.
    return findParent(parent[v], parent);
}

void kruskals(Edge *input, int n, int E) {
    // In-built sort function: Sorts the edges in
    // increasing order of their weights
```

```
    sort(input, input + E, compare);

    // Array to store final edges of MST
    Edge *output = new Edge[n-1];
    // Parent array initialized with their indexes
    int *parent = new int[n];

    for (int i = 0; i < n; i++) {
        parent[i] = i;
    }

    int count = 0;       //  To maintain the count of number of edges in
the MST
    int i = 0;                  // Index to traverse over the input array
    while (count != n - 1) {     // As the MST contains n-1 edges.
        Edge currentEdge = input[i];
        // Figuring out the parent of each edge's vertices
        int sourceParent = findParent(currentEdge.source, parent);
        int destParent = findParent(currentEdge.dest, parent);
        // If their parents are not equal, then we added that edge to
output
        if(sourceParent != destParent) {
            output[count] = currentEdge;
            count++;       // Increased the count
            parent[sourceParent] = destParent; // Updated the parent array
        }
        i++;
    }
    // Finally, printing the MST obtained.
    for (int i = 0; i < n-1; i++) {
        if(output[i].source < output[i].dest) {
            cout << output[i].source << " " << output[i].dest << " " <<
output[i].weight << endl;
        }
        else {
            cout<< output[i].dest << " " <<output[i].source << " " <<
output[i].weight << endl;
        }

    }
```

```
}

int main() {
    int n, E;
    cin >> n;
    cin >> E;

    Edge *input = new Edge[E];

    for (int i = 0; i < E; i++) {
        int s, d, w;
        cin >> s >> d >> w;
        input[i].source = s;
        input[i].dest = d;
        input[i].weight = w;
    }

    kruskals(input, n, E);
            return 0;
}
```

Time Complexity of Kruskal's Algorithm:

In our code, we have the following three steps: (Here, the total number of vertices is n, and the total number of edges is E)

- Take input in the array of size E.
- Sort the input array on the basis of edge-weight. This step has the time complexity of O(E log(E)).
- Pick (n-1) edges and put them in MST one-by-one. Also, before adding the edge to the MST, we checked for cycle detection for each edge. For cycle

detection, in the worst-case time complexity of E edges will be O(E.n), as discussed earlier.

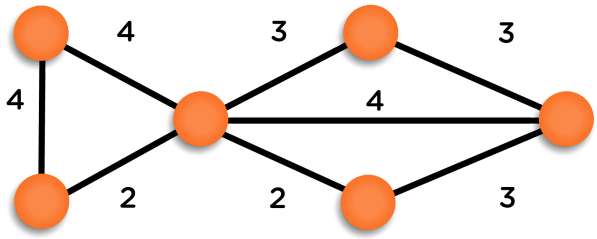
Hence, the total time complexity of Kruskal's algorithm becomes O(E log(E) + n.E). This time complexity is bad and needs to be improved. We can't reduce the time taken for sorting, but the time taken for cycle detection can be improved using another algorithm named **Union by Rank and Path Compression**. You need to explore this on yourselves. The basic idea in these algorithms is that we will be avoiding the formation of skewed-tree structure, which reduces the time complexity for each vertex to O(log(E)).

Prim's Algorithm

This algorithm is used to find MST for a given undirected-weighted graph (which can also be achieved using Kruskal's Algorithm).

In this algorithm, the MST is built by adding one edge at a time. In the beginning, the spanning tree consists of only one vertex, which is chosen arbitrarily from the set of all vertices of the graph. Then the minimum weighted edge, outgoing from this vertex, is selected and simultaneously inserted into the MST. Now, the tree contains two edges. Further, we will be selecting the edge with the minimum weight such that one end is already present there in the MST and the other one from the unselected set of vertices. This process is repeated until we have inserted a total of (n-1) edges in the MST.

Consider the following example for a better understanding.

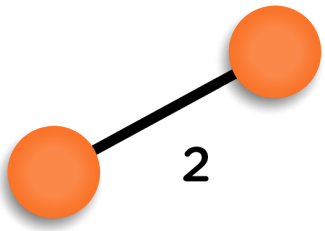


Step: 1

Start with a weighted graph

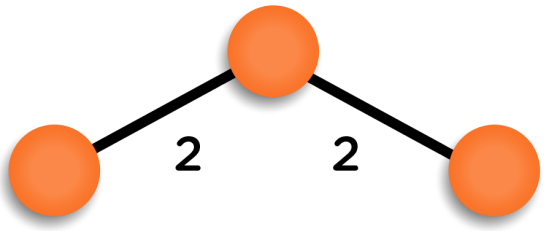
Step: 2

Choose a vertex



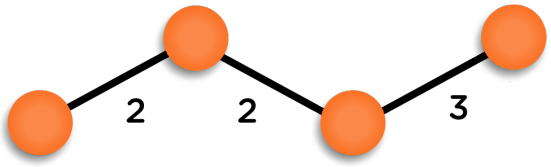
Step: 3

Choose the shortest edge from this vertex add it



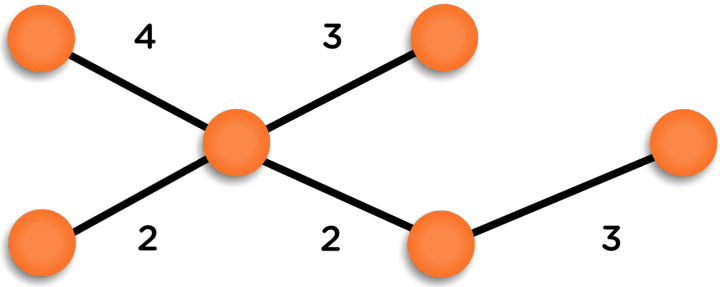
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution,
if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Implementation:

- We are considering the starting vertex to be 0 with a parent equal to -1, and weight is equal to 0 (The weight of the edge from vertex 0 to vertex 0 itself).
- The parent of all other vertices is assumed to be NIL, and the weight will be equal to infinity, which means that the vertex has not been visited yet.

- We will mark the vertex 0 as visited and rest as unvisited. If we add any vertex to the MST, then that vertex will be shifted from the unvisited section to the visited section.
- Now, we will update the weights of direct neighbors of vertex 0 with the edge weights as these are smaller than infinity. We will also update the parent of these vertices and assign them 0 as we reached these vertices from vertex 0.
- This way, we will keep updating the weights and parents, according to the edge, which has the minimum weight connected to the respective vertex.

Let's look at the code now:

```
#include <iostream>
#include <climits>
using namespace std;

int findMinVertex(int *weights, bool *visited, int n) {
    // Initialized to -1 means there is no vertex till now
    int minVertex = -1;
    for (int  i = 0; i < n; i++) {
//Conditions: the vertex must be unvisited and either minVertex value is -1
//  or if minVertex has some vertex to it, then weight of currentvertex
// should be less than the weight of the minVertex.
        if (!visited[i] && (minVertex == -1 || weights[i] <
weights[minVertex])) {
            minVertex = i;
        }
    }
    return minVertex;
}

void prims(int **edges, int n) {

    int *parent = new int[n];
    int *weights = new int[n];
    bool *visited = new bool[n];
    // Initially, the visited array is assigned to false and weights array
//to infinity.
```

```
    for(int i = 0; i < n; i++) {
        visited[i] = false;
        weights[i] = INT_MAX;
    }
    // Values assigned to vertex 0.(the selected starting vertex to begin
with)
    parent[0] = -1;
    weights[0] = 0;

    for (int i = 0; i < n-1; i++) {
        // Find min vertex
        int minVertex = findMinVertex(weights, visited, n);
        visited[minVertex] = true;
        // Explore unvisited neighbors
        for (int j = 0; j < n; j++) {
            if(edges[minVertex][j] != 0 && !visited[j]) {
                if(edges[minVertex][j] < weights[j]) {
                    // updating weight array and parent array
                    weights[j] = edges[minVertex][j];
                    parent[j] = minVertex;
                }
            }
        }
    }
    // Final MST printed
    for (int i = 0; i < n; i++) {
        if (parent[i] < i) {
            cout << parent[i] << " " << i << " " <<weights[i] << endl;
        }
        else {
            cout << i << " " << parent[i] << " " << weights[i] << endl;
        }
    }
}

int main() {

    int n;
    int e;
    cin >> n >> e;
```

```
    int **edges = new int*[n]; // Adjacency matrix used to store the graph
    for (int i = 0; i < n; i++) {
        edges[i] = new int[n];
        for (int j = 0; j < n; j++) {
            // Initially all pairs are assigned 0 weight which
            // means that there is no edge between them
            edges[i][j] = 0;
        }
    }

    for (int i = 0; i < e; i++) {
        int f, s, weight;
        cin >> f >> s >> weight;
        edges[f][s] = weight;
        edges[s][f] = weight;
    }

    prims(edges, n);

    for(int i = 0; i < n; i++) {
        delete [] edges[i];
    }
    delete [] edges;
    return 0;
}
```

Time Complexity of Prim's Algorithm:

Here, n is the number of vertices, and E is the number of edges.

- The time complexity for finding the minimum weighted vertex is O(n) for each iteration. So for (n-1) edges, it becomes $O(n^2)$.
- Similarly, for exploring the neighbor vertices, the time taken is $O(n^2)$.

It means the time complexity of Prim's algorithm is $O(n^2)$. We can improve this in the following ways:

- For exploring neighbors, we are required to visit each and every vertex because of the adjacency matrix. We can improve this by using an adjacency list instead of a matrix.
- Now, the second important thing is the time taken to find the minimum weight vertex, which is also taking a time of $O(n^2)$. Here, out of the available list, we are trying to figure out the one with minimum weight. This can be optimally achieved using a **priority queue** where the priority will be taken as weights of the vertices. This will take O(log(n)) time complexity to remove a vertex from the priority queue.

These optimizations can lead us to the time complexity of O((n+E)log(n)), which is much better than the earlier one. Try to write the optimized code by yourself.

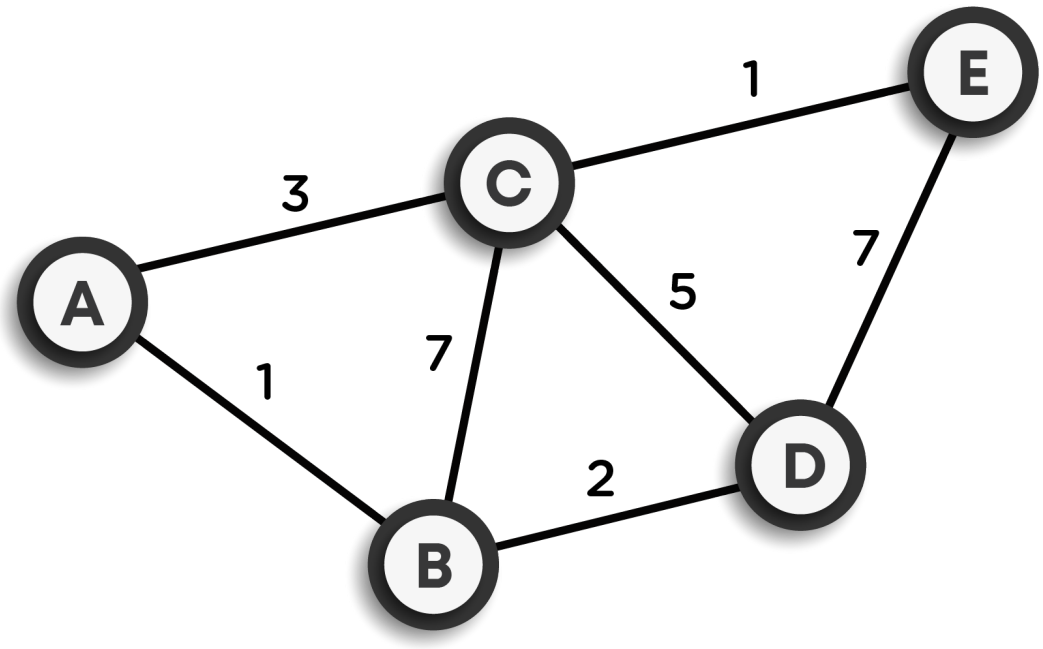
Dijkstra's Algorithm

This algorithm is used to find the shortest distance between any two vertices in a weighted non-cyclic graph.

Here, we will be using a slight modification of the algorithm according to which we will be figuring out the minimum distance of all the vertices from the particular source vertex.

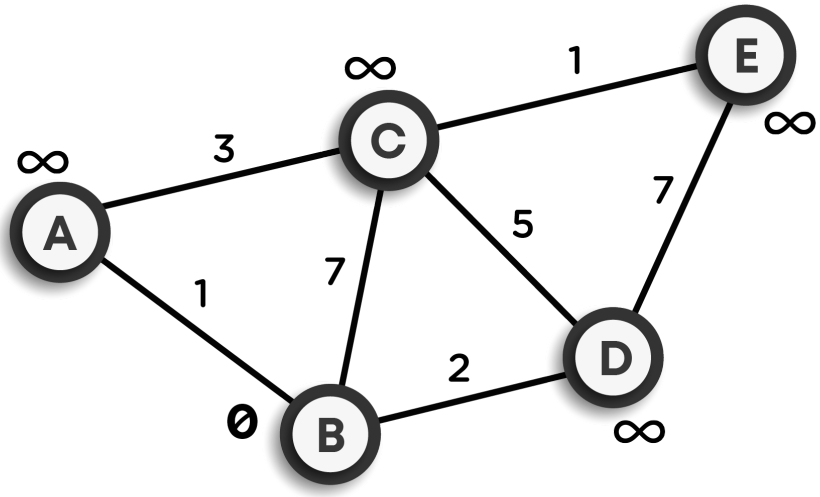
Let's consider the algorithm with an example:

1. We want to calculate the shortest path between the source vertex C and all other vertices in the following graph.

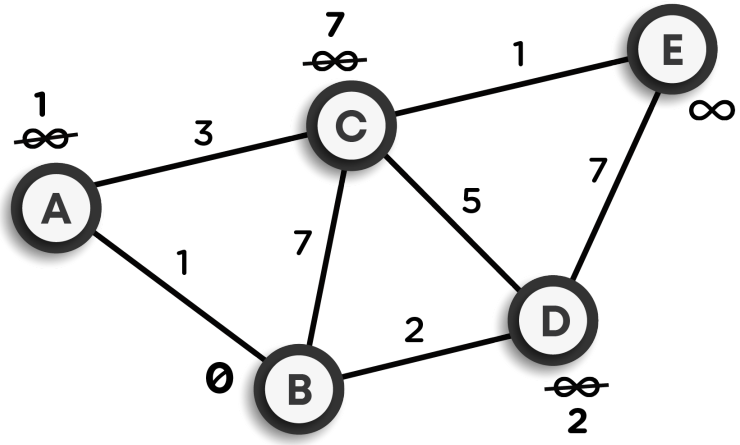


2. While executing the algorithm, we will mark every node with its **minimum distance** to the selected node, which is C in our case. Obviously, for node C

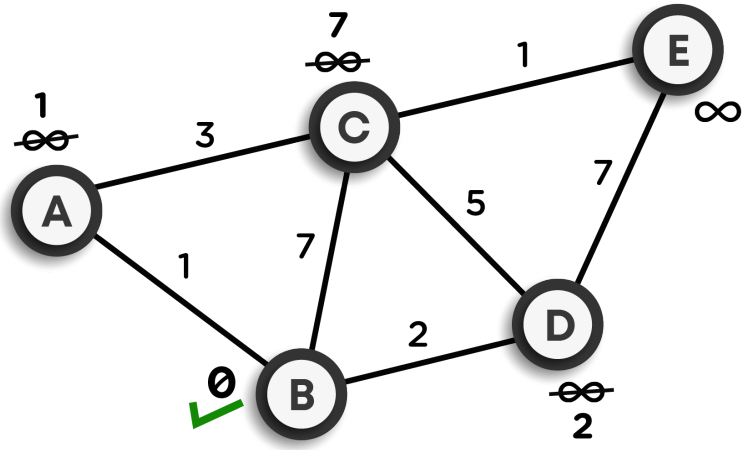
itself, this distance will be 0, and for the rest of the nodes, we will assume that the distance is infinity, which also denotes that these vertices have not been visited till now.



3. Now, we will check for the neighbors of the current node, which in our case is A, B, and D. Now, we will add the minimum cost of the current node to the weight of the edge connecting the current node and the particular neighbor node. For example, for node B, it's weight will become minimum(infinity, 0+7) = 7. This same process is repeated for other neighbor nodes.

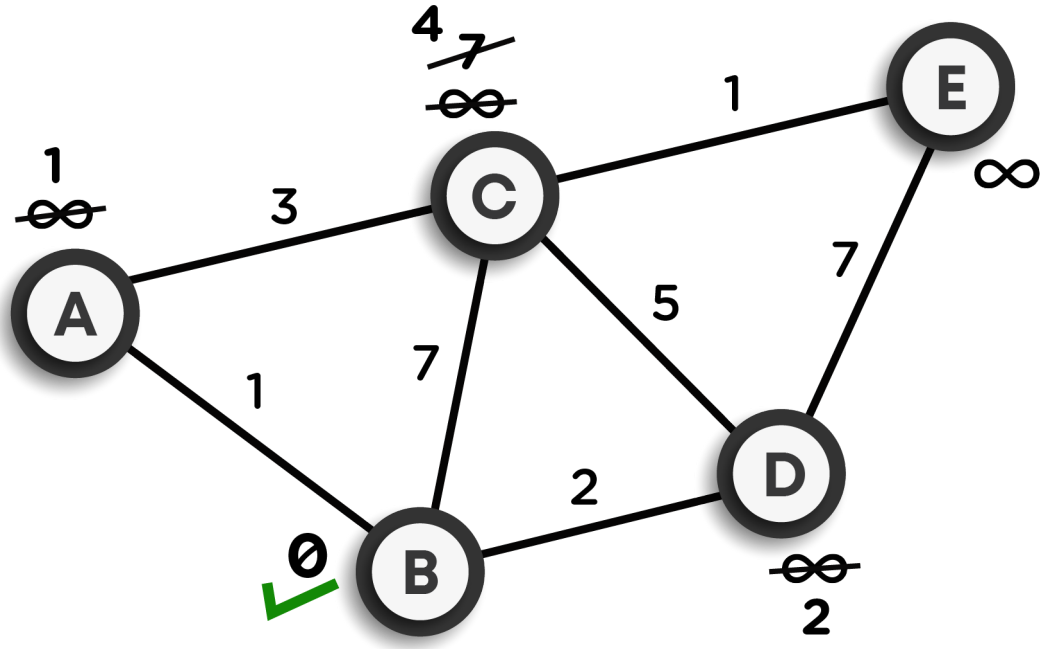


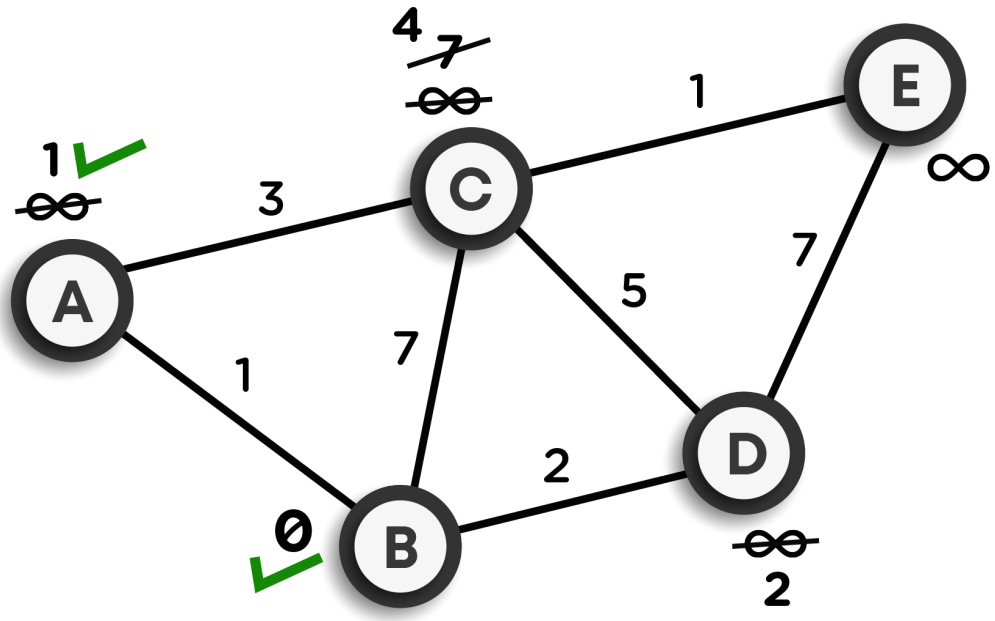
4. Now, as we have updated the distance of all the neighbor nodes of the current node, we will mark the current node as visited.

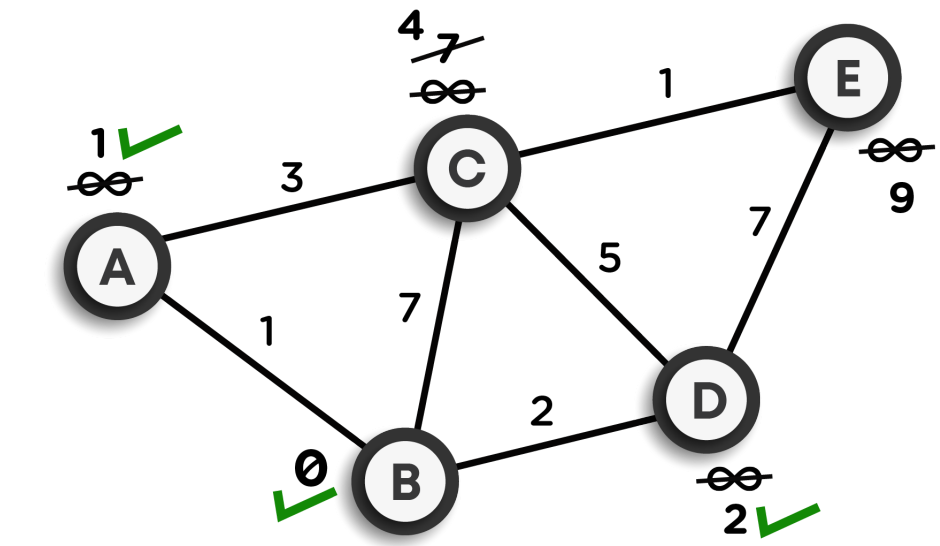


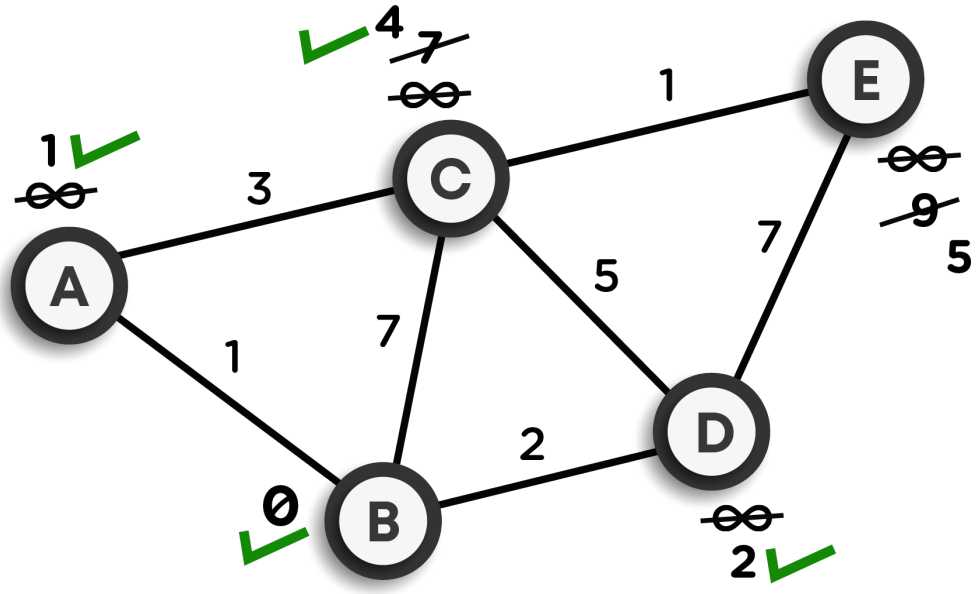
5. After this, we will be selecting the minimum weighted node among the remaining vertices. In this case, it is node A. Take this node as the current node.

6. Now, we will repeat the above steps for the rest of the vertices. The pictorial representation of the same is shown below:

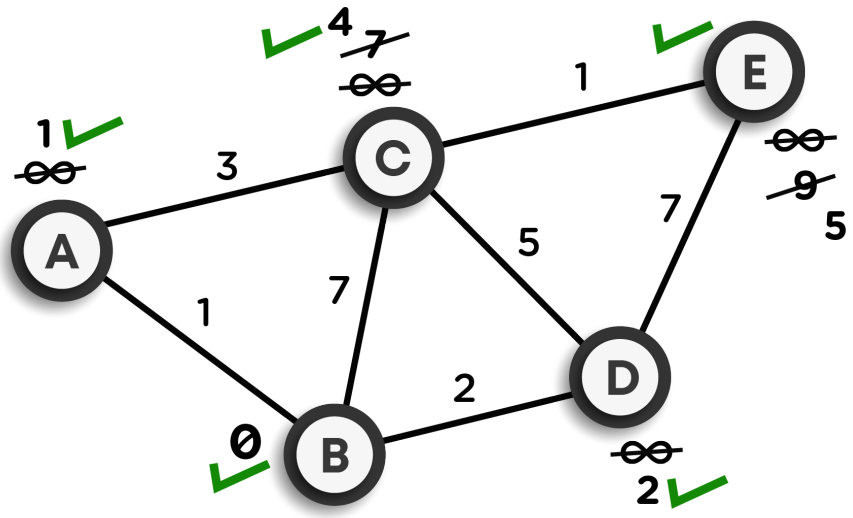








7. Finally, we will get the graph as follows:



The distances finally marked at each node are minimum from node C.

Implementation:

Let's look at the code below for a better explanation:(Code is nearly same as that of Prim's algorithm, just a change while updating the distance)

```
#include <iostream>
#include <climits>
using namespace std;

int findMinVertex(int *distance, bool *visited, int n) {

    int minVertex = -1;
    for (int  i = 0; i < n; i++) {
        if (!visited[i] && (minVertex == -1 || distance[i] <
distance[minVertex])) {
            minVertex = i;
        }
    }
    return minVertex;
}
```

```
void dijkstra(int **edges, int n) {

    int *distance = new int[n];
    bool *visited = new bool[n];

    for(int i = 0; i < n; i++) {
        visited[i] = false;
        distance[i] = INT_MAX;
    }

    distance[0] = 0;  // 0 is considered as the starting node.

    for (int i = 0; i < n-1; i++) {
        // Find min vertex
        int minVertex = findMinVertex(distance, visited, n);
        visited[minVertex] = true;
        // Explore unvisited neighbors
        for (int j = 0; j < n; j++) {
            if(edges[minVertex][j] != 0 && !visited[j]) {
                    // distance of any node will be the current node's
distance + the weight
                    // of the edge between them
                int dist = distance[minVertex] + edges[minVertex][j];
                if(dist < distance[j]) {     // If required, then updated.
                    distance[j] = dist;
                }
            }
        }
    }
    // Final output of distance of each node with respect to 0
    for (int i = 0; i < n; i++) {
        cout << i << " " << distance[i] << endl;
    }
}

int main() {

    int n;
    int e;
```

```
    cin >> n >> e;
    int **edges = new int*[n];
    for (int i = 0; i < n; i++) {
        edges[i] = new int[n];
        for (int j = 0; j < n; j++) {
            edges[i][j] = 0;
        }
    }

    for (int i = 0; i < e; i++) {
        int f, s, weight;
        cin >> f >> s >> weight;
        edges[f][s] = weight;
        edges[s][f] = weight;
    }

    dijkstra(edges, n);

    for(int i = 0; i < n; i++) {
        delete [] edges[i];
    }
    delete [] edges;
    return 0;
}
```

Time Complexity of Dijkstra's algorithm:

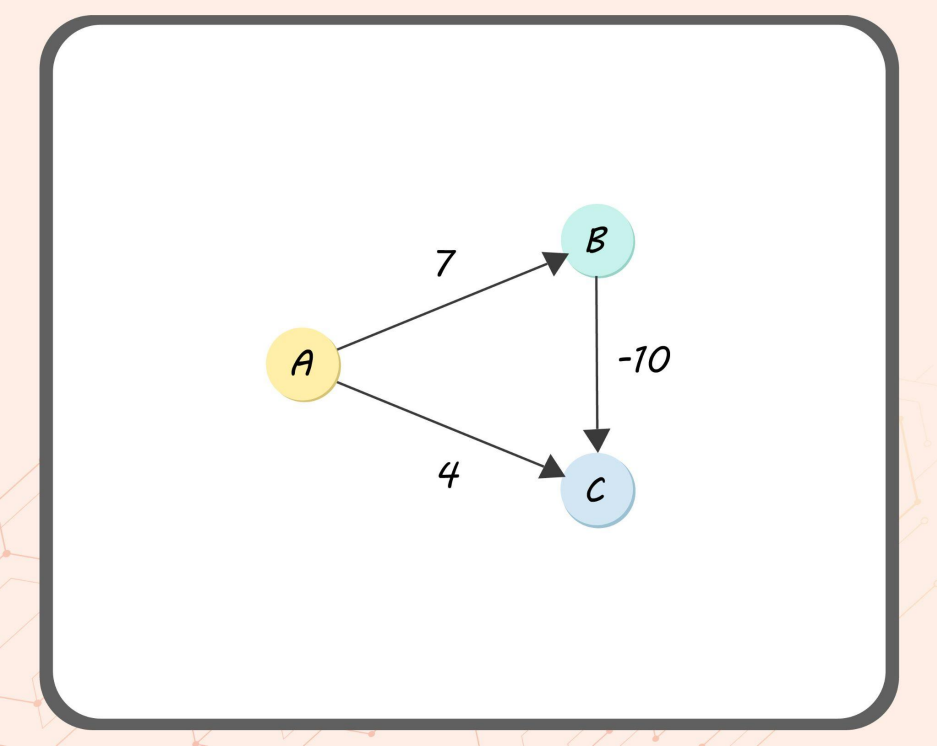
The time complexity is also the same as that of Prim's algorithm, i.e., $O(n^2)$. This can be reduced by using the same approaches as discussed in Prim's algorithm's content.

Bellman Ford Algorithm

This algorithm is used to find the shortest distance between any two vertices in a weighted non-cyclic graph.

It overcomes the shortcomings of Dijkstra's algorithm, which fails for a graph with negative weight.

Considering an example :



If we apply Dijkstra's algorithm in the above example it gives us the path A → C.

However in reality the path A → B → C is the correct answer with the distance of 7 - 10 = -3.

So Bellman Ford suggests that in order to take into account the negative edges, we must calculate the shortest path between any two vertices through different numbers of edges so as to find out the correct answer amongst them.

In the above example, if we find the shortest path with at most one edge then our distance is 4. But if we find the shortest path once more and this time we could take at most 2 edges to travel from our source vertex to the destination vertex then our total distance is -3. So amongst 4 and -3, -3 is shorter and hence our final path will be A→ B→ C.

Algorithm:

1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in a given graph.

a) Do following for each edge u-v

If dist[v] > dist[u] + weight of edge uv, then update dist[v]

dist[v] = dist[u] + weight of edge uv

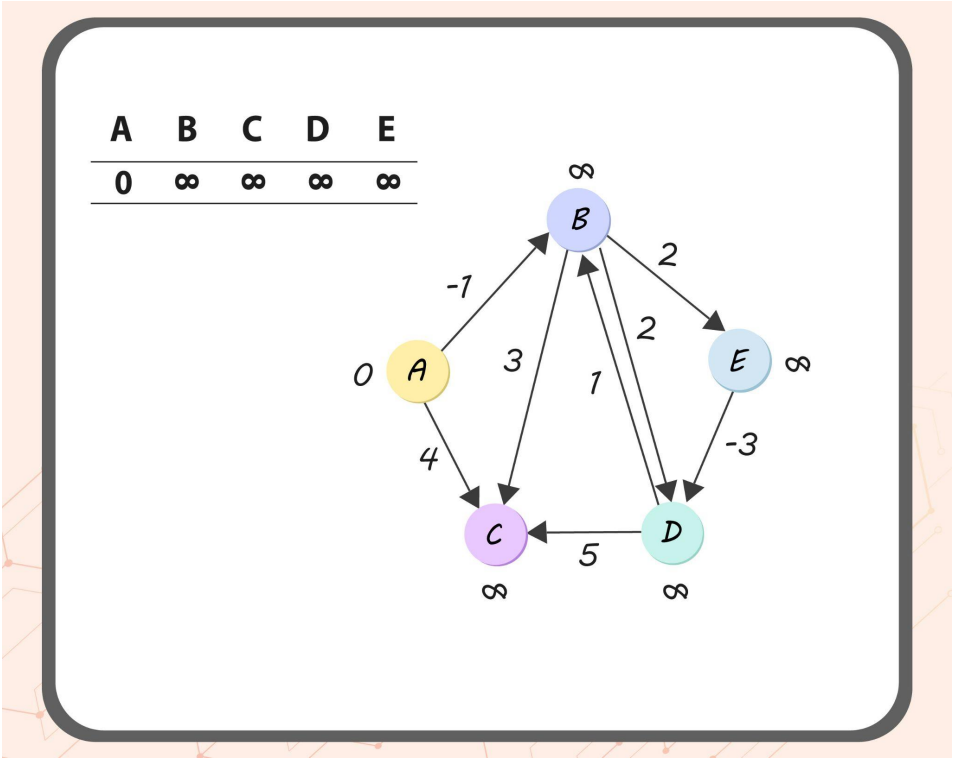
3) This step reports if there is a negative weight cycle in the graph. Do following for each edge u-v

If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"

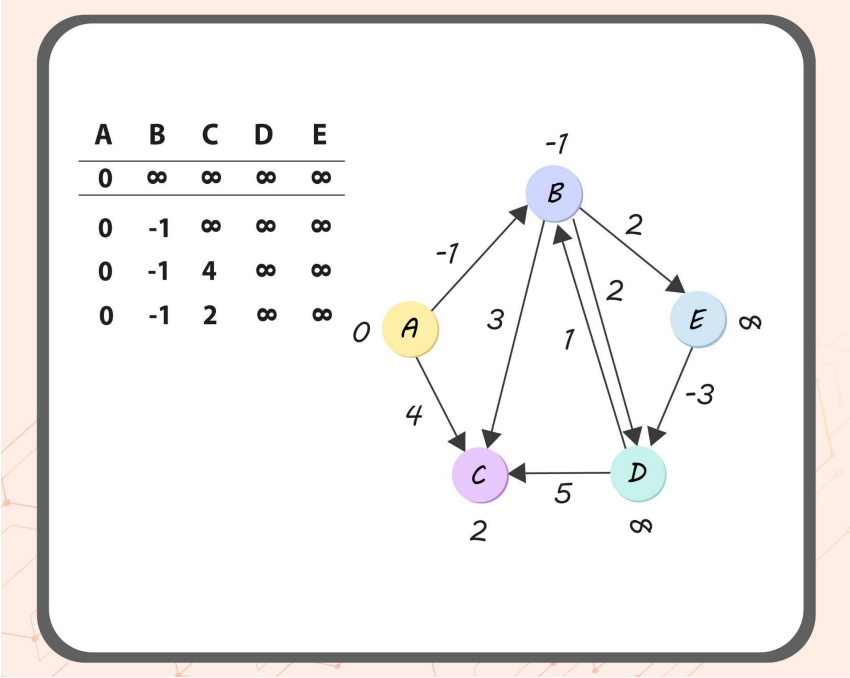
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one

more time and get a shorter path for any vertex, then there is a negative weight cycle.

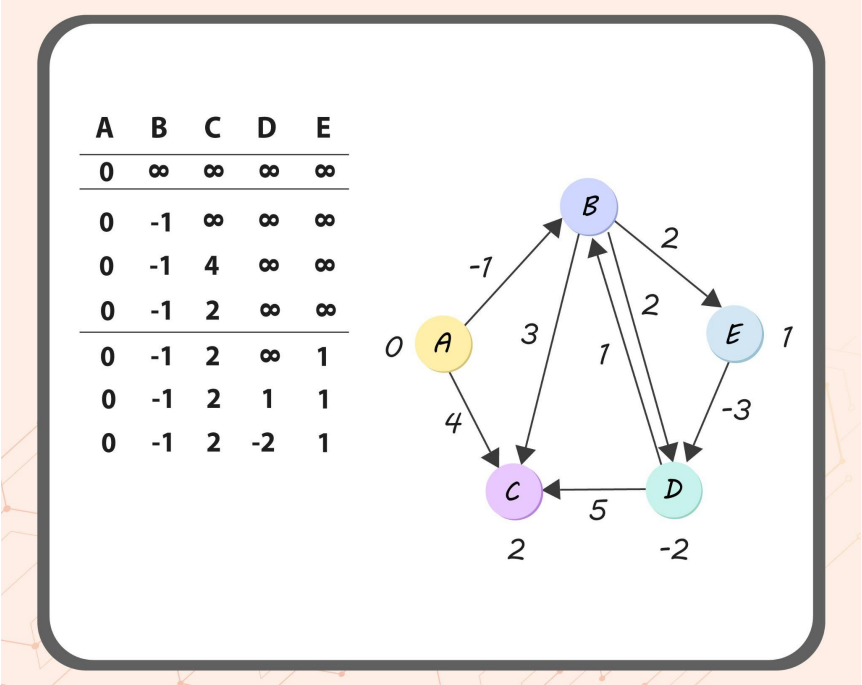
Let's take a look at another example and see how this works step by step.



Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D).



The first iteration guarantees to give all shortest paths which are at most 1 edge long.



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

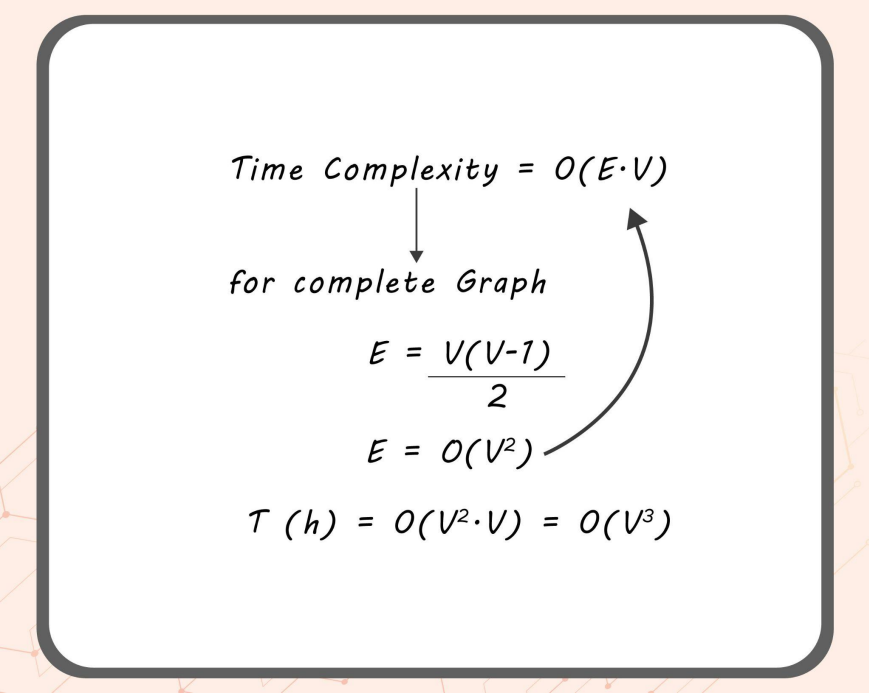
Implementation:

```
#include <bits/stdc++.h>
using namespace std;

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

void printAns(int dist[], int n)
{
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
```

```
    int E = graph->E;
    int dist[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle");
            return; // If negative cycle is detected, simply return
        }
    }

    printAns(dist, V);
    return;
}

int main()
{
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    struct Graph* graph = createGraph(V, E);
```

```
    for (int i = 0; i < E; i++)
    {
        int u, v, w;
        cout << "Enter the u, v and w(weight) for " << i + 1 << "th edge:
";
        cin >> u >> v >> w;
        graph->edge[i].src = u;
        graph->edge[i].dest = v;
        graph->edge[i].weight = w;
    }
    BellmanFord(graph, 0);
    return 0;
}
```

Time Complexity of Bellman Ford algorithm:

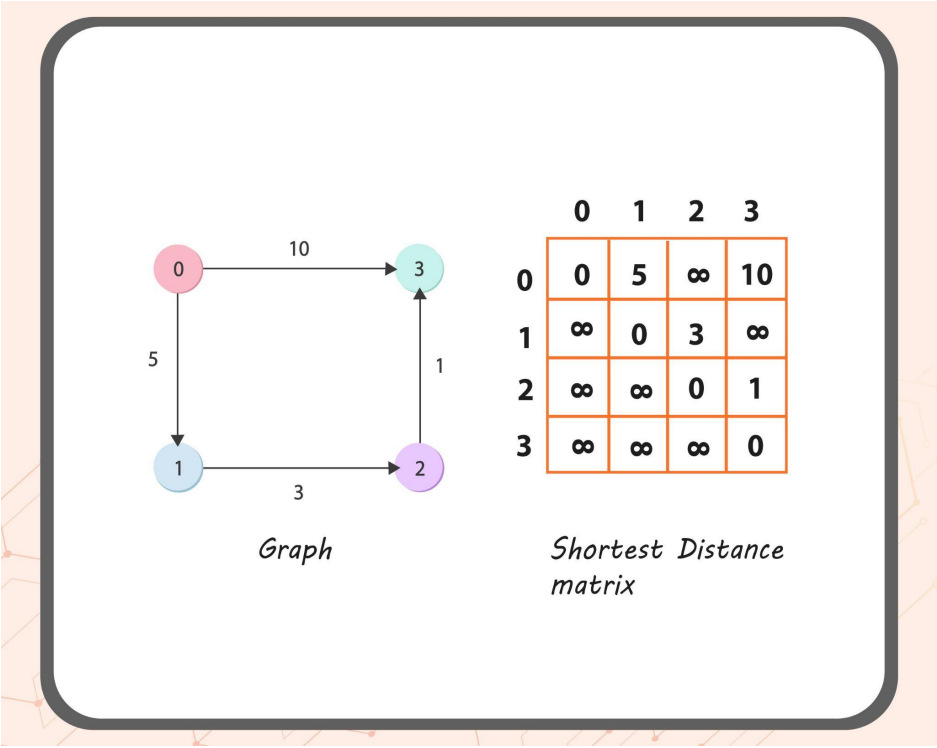


Floyd Warshall Algorithm

This algorithm is used to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

We store the distances in a matrix of N X N in which if the value at any cell [i][j] is infinity, then that means there exists no path between vertex i and vertex j.

Let's look at an example for better understanding:



	0	1	2	3
0	0	5	∞	10
1	∞	0	3	∞
2	∞	∞	0	1
3	∞	∞	∞	0

The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered

vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1. k is not an intermediate vertex in the shortest path from i to j. We keep the value of dist[i][j] as it is.
2. k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

Code :

```
/*
Time complexity: O(N^3)
Space complexity: O(N^2)
where N is the number of vertex in the graph
*/
#include <bits/stdc++.h>
using namespace std;
int INF = 1e9;
int main(){
    int n, m;
    cout << "Enter the number of vertices: ";
    cin >> n;
    cout << "Enter the number of edges: ";
    cin >> m;
    int mat[n][n];
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            mat[i][j] = INF;
            if (i == j){
                mat[i][j] = 0;
            }
        }
    }
    for (int i = 0; i < m; i++){
        int a, b, c;cout << "Enter the u, v and w(weight) for " << i + 1 <<
"th edge: ";
        cin >> a >> b >> c;
        mat[a][b] = c;
```

```
    }
    int i, j, k;
    for (k = 0; k < n; k++){
        for (i = 0; i < n; i++){
            for (j = 0; j < n; j++){
                if (mat[i][j] > (mat[i][k] + mat[k][j]) && (mat[k][j] !=
INF && mat[i][k] != INF))
                mat[i][j] = mat[i][k] + mat[k][j];
            }
        }
    }
    cout << "\nFinal Matrix A" << n << ": " << endl;
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            cout << mat[i][j] << " ";
    }
    cout << endl;
    }
}
```