Krushkal Algorithm for MST

```
#include <iostream>

#include <vector>

#include <algorithm>


using namespace std;


// Edge structure to represent a graph edge

struct Edge {

   int src, dest, weight;

};


// Graph structure to represent a graph with V vertices and E edges

struct Graph {

   int V, E;

   vector<Edge> edges;

};


// Subset structure for union-find

struct Subset {

   int parent;

   int rank;

};
```

```
// Function to create a graph with V vertices and E edges
Graph* createGraph(int V, int E) {
   Graph* graph = new Graph;
   graph->V = V;
   graph->E = E;
   return graph;
}

// Function to find the parent of a node i
int find(Subset subsets[], int i) {
   if (subsets[i].parent != i)
      subsets[i].parent = find(subsets, subsets[i].parent);
   return subsets[i].parent;
}
// Function to unite two subsets u and v
void Union(Subset subsets[], int u, int v) {
   int rootU = find(subsets, u);
   int rootV = find(subsets, v);

   if (subsets[rootU].rank < subsets[rootV].rank) {
      subsets[rootU].parent = rootV;
   } else if (subsets[rootU].rank > subsets[rootV].rank) {
      subsets[rootV].parent = rootU;
   } else {
      subsets[rootV].parent = rootU;
      subsets[rootU].rank++;
   }
}
```

```cpp
// Comparator function to sort edges by weight
bool compare(Edge a, Edge b) {
    return a.weight < b.weight;
}

// Kruskal's algorithm to find the MST of a given graph
void KruskalMST(Graph* graph) {
    int V = graph->V;
    vector<Edge> result; // Store the resultant MST
    int e = 0; // Index for result
    int i = 0; // Index for sorted edges

    // Sort all the edges in non-decreasing order of their weight
    sort(graph->edges.begin(), graph->edges.end(), compare);

    // Allocate memory for creating V subsets
    Subset* subsets = new Subset[V];
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }
```

```
    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->edges.size()) {
        // Pick the smallest edge and increment the index for next iteration
        Edge next_edge = graph->edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does not cause a cycle, include it in result
        // and increment the index of result for next edge
        if (x != y) {
            result.push_back(next_edge);
            Union(subsets, x, y);
            e++;
        }
    }

    // Print the resultant MST
    cout << "Edges in the MST:\n";
    for (i = 0; i < result.size(); ++i)
        cout << result[i].src << " -- " << result[i].dest << " == " << result[i].weight << endl;

    // Free allocated memory
    delete[] subsets;
}
```

```
//driver code
int main() {
   int V = 4; // Number of vertices in graph
   int E = 5; // Number of edges in graph
   Graph* graph = createGraph(V, E);

   // Add edges
   graph->edges.push_back({0, 1, 10});
   graph->edges.push_back({0, 2, 6});
   graph->edges.push_back({0, 3, 5});
   graph->edges.push_back({1, 3, 15});
   graph->edges.push_back({2, 3, 4});

   // Function call
   KruskalMST(graph);

   // Free allocated memory for the graph
   delete graph;

   return 0;
```

Dijkstra Algorithm for Shortest Path

```
#include <iostream>
#include <vector>
#include <queue>
#include <utility>
#include <limits>

using namespace std;

typedef pair<int, int> pii;

// Function to perform Dijkstra's algorithm
vector<int> dijkstra(int n, vector<vector<pii>>& adj, int src) {
    // Priority queue to select the vertex with the smallest distance
    priority_queue<pii, vector<pii>, greater<pii>> pq;
    // Vector to store the shortest distance from source to each vertex
    vector<int> dist(n, numeric_limits<int>::max());
    dist[src] = 0;
    pq.push(make_pair(0, src));
    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
```

```
        // Traverse all adjacent vertices of the dequeued vertex u
        for (const auto& neighbor : adj[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            // If there is a shorter path to v through u
            if (dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    return dist;
}

//driver code
int main() {
    int n, m, src;
    cout << "Enter the number of vertices and edges: ";
    cin >> n >> m;

    vector<vector<pii>> adj(n);

    cout << "Enter the edges (u v w) where u and v are vertices and w is weight:" << endl;
```

```
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].emplace_back(v, w);
        adj[v].emplace_back(u, w); // For undirected graph
    }

    cout << "Enter the source vertex: ";
    cin >> src;

    vector<int> dist = dijkstra(n, adj, src);

    cout << "Vertex Distance from Source" << endl;
    for (int i = 0; i < n; i++) {
        cout << i << " \t\t " << dist[i] << endl;
    }

    return 0;
}
```

Bellman Ford Algorithm

```
#include <iostream>

#include <vector>

#include <limits.h>


using namespace std;


// Structure to represent a weighted edge in a graph

struct Edge {

   int src, dest, weight;

};


// Function to find the shortest paths from source vertex to all other vertices using Bellman-Ford algorithm

void bellmanFord(vector<Edge>& edges, int V, int E, int src) {

   // Step 1: Initialize distances from src to all other vertices as INFINITE

   vector<int> dist(V, INT_MAX);

   dist[src] = 0;


   // Step 2: Relax all edges |V| - 1 times.

   for (int i = 1; i <= V - 1; i++) {

      for (int j = 0; j < E; j++) {

         int u = edges[j].src;

         int v = edges[j].dest;

         int weight = edges[j].weight;
```

```
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            dist[v] = dist[u] + weight;
        }
    }
}

// Step 3: Check for negative-weight cycles.
for (int i = 0; i < E; i++) {
    int u = edges[i].src;
    int v = edges[i].dest;
    int weight = edges[i].weight;
    if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
        cout << "Graph contains negative weight cycle" << endl;
        return;
    }
}

// Print all distances
cout << "Vertex\tDistance from Source" << endl;
for (int i = 0; i < V; ++i) {
    cout << i << "\t\t" << dist[i] << endl;
}
}
```

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

    vector<Edge> edges(E);

    cout << "Enter the edges in the format (source destination weight):" << endl;
    for (int i = 0; i < E; ++i) {
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    }

    cout << "Enter the source vertex: ";
    cin >> src;

    bellmanFord(edges, V, E, src);

    return 0;
}
```