**Batch : T7**

**Practical No. 7**

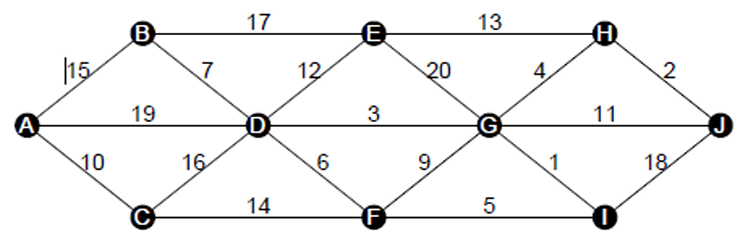
**Title of Assignment Greedy Method**

**Student Name: Om Kulkarni**

**Student PRN: 22510034**

**Problem Statement:**

1. **Implement Kruskal’s algorithm & Prim's algorithm to find Minimum Spanning Tree (MST) of the given an undirected, connected and weighted graph.**

****

Q) How many edges does a minimum spanning tree for above example?

Q) In a graph G. let the edge u v have the least weight. is it true that u v is always part of any minimum spanning tree of G?.Justify your answers.

Q) Let G be a graph and T be a minimum spanning tree of G. Suppose that the weight of an edge e is decreased. How can you find the minimum spanning tree of the modified graph? What is the runtime of your solution?

Q) Find order of edges for Kruskal's and Prim's?

1. Algorithm/Pseudocode

**Kruskal's Algorithm:**

1. **Sort all edges** of the graph in increasing order of their weights.
2. Start adding edges to the MST from the **smallest weight** edge, ensuring that no cycle is formed.
3. Stop when you have V−1V - 1V−1 edges in the MST, where VVV is the number of vertices.

**Prim's Algorithm:**

1. Start with any vertex and **add the least weight edge** connecting the current tree (or vertex) to a new vertex.
2. Continue adding the least weight edge from the **set of vertices already included** in the MST to a vertex not yet included.
3. Repeat until all vertices are included.
4. Program Code

Kruskal’s Algorithm :

#include <iostream>

#include <vector>

#include <algorithm>

#include <unordered\_map>

using namespace std;

struct Edge {

    int src, dest, weight;

};

struct DisjointSet {

    vector<int> parent, rank;

    DisjointSet(int *n*) {

        parent.resize(*n*);

        rank.resize(*n*, 0);

        for (int i = 0; i < *n*; i++)

            parent[i] = i;

    }

    int find(int *u*) {

        if (parent[*u*] != *u*)

            parent[*u*] = find(parent[*u*]);

        return parent[*u*];

    }

    void unionSet(int *u*, int *v*) {

        int rootU = find(*u*);

        int rootV = find(*v*);

        if (rootU != rootV) {

            if (rank[rootU] > rank[rootV]) {

                parent[rootV] = rootU;

            } else if (rank[rootU] < rank[rootV]) {

                parent[rootU] = rootV;

            } else {

                parent[rootV] = rootU;

                rank[rootU]++;

            }

        }

    }

};

bool compareEdges(Edge *a*, Edge *b*) {

    return *a*.weight < *b*.weight;

}

void kruskal(int *vertices*, vector<Edge>& *edges*, const vector<char>& *vertexChars*) {

    sort(*edges*.begin(), *edges*.end(), compareEdges);

    DisjointSet ds(*vertices*);

    vector<Edge> mst;

    int totalCost = 0; // Variable to store total cost of MST

    for (const auto& edge : *edges*) {

        if (ds.find(edge.src) != ds.find(edge.dest)) {

            ds.unionSet(edge.src, edge.dest);

            mst.push\_back(edge);

            totalCost += edge.weight; // Add weight to total cost

        }

    }

    cout << "Kruskal's MST edges:\n";

    for (const auto& edge : mst) {

        cout << *vertexChars*[edge.src] << " -- " << *vertexChars*[edge.dest] << " == " << edge.weight << endl;

    }

    cout << "Total cost of the MST: " << totalCost << endl; // Output the total cost

}

int main() {

    int vertices, edgesCount;

    cout << "Enter number of vertices: ";

    cin >> vertices;

    cout << "Enter number of edges: ";

    cin >> edgesCount;

    vector<char> vertexChars(vertices);

    vector<Edge> edges(edgesCount);

    cout << "Enter vertices (one at a time):\n";

    for (int i = 0; i < vertices; i++) {

        cin >> vertexChars[i];

    }

    cout << "Enter edges in the format (src dest weight):\n";

    for (int i = 0; i < edgesCount; i++) {

        char src, dest;

        int weight;

        cin >> src >> dest >> weight;

        edges[i].src = find(vertexChars.begin(), vertexChars.end(), src) - vertexChars.begin();

        edges[i].dest = find(vertexChars.begin(), vertexChars.end(), dest) - vertexChars.begin();

        edges[i].weight = weight;

    }

    kruskal(vertices, edges, vertexChars);

    return 0;

}

Prim’s Algorithm :

#include <iostream>

#include <vector>

#include <queue>

#include <utility>

#include <limits>

#include <algorithm>

using namespace std;

void prim(int *vertices*, vector<vector<pair<int, int>>>& *adj*, const vector<char>& *vertexChars*) {

    vector<bool> inMST(vertices, false);

    vector<int> key(vertices, INT\_MAX);

    vector<int> parent(vertices, -1);

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    key[0] = 0;

    pq.push({0, 0});

    int totalCost = 0;

    while (!pq.empty()) {

        int u = pq.top().second;

        pq.pop();

        if (inMST[u]) continue;

        inMST[u] = true;

        totalCost += key[u];

        for (const auto& [v, weight] : adj[u]) {

            if (!inMST[v] && weight < key[v]) {

                key[v] = weight;

                parent[v] = u;

                pq.push({key[v], v});

            }

        }

    }

    cout << "Prim's MST edges:\n";

    for (int i = 1; i < vertices; i++) {

        if (parent[i] != -1) {

            cout << vertexChars[parent[i]] << " -- " << vertexChars[i] << " == " << key[i] << endl;

        }

    }

    cout << "Total cost of the MST: " << totalCost << endl;

}

int main() {

    int vertices, edgesCount;

    cout << "Enter number of vertices: ";

    cin >> vertices;

    cout << "Enter number of edges: ";

    cin >> edgesCount;

    vector<char> vertexChars(vertices);

    vector<vector<pair<int, int>>> adj(vertices);

    cout << "Enter vertices (one at a time):\n";

    for (int i = 0; i < vertices; i++) {

        cin >> vertexChars[i];

    }

    cout << "Enter edges in the format (src dest weight):\n";

    for (int i = 0; i < edgesCount; i++) {

        char src, dest;

        int weight;

        cin >> src >> dest >> weight;

        int srcIndex = find(vertexChars.begin(), vertexChars.end(), src) - vertexChars.begin();

        int destIndex = find(vertexChars.begin(), vertexChars.end(), dest) - vertexChars.begin();

        adj[srcIndex].emplace\_back(destIndex, weight);

        adj[destIndex].emplace\_back(srcIndex, weight);

    }

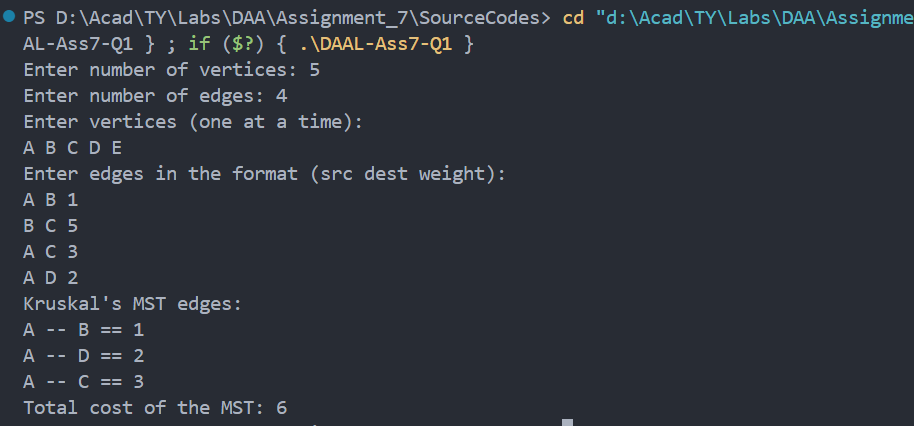
    prim(vertices, adj, vertexChars);

    return 0;

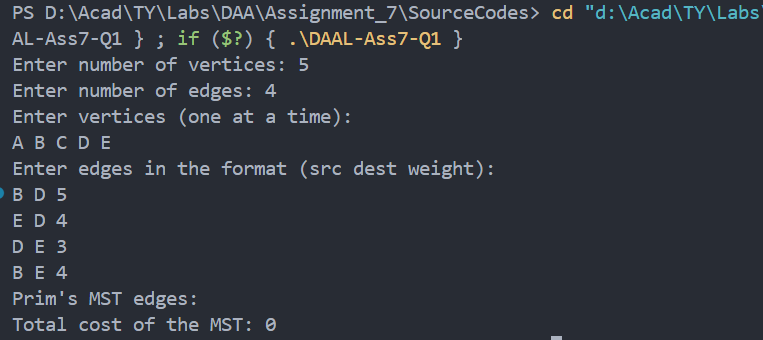
}

1. Output with verity of test cases

Kruskal’s Algorithm :



Prim’s



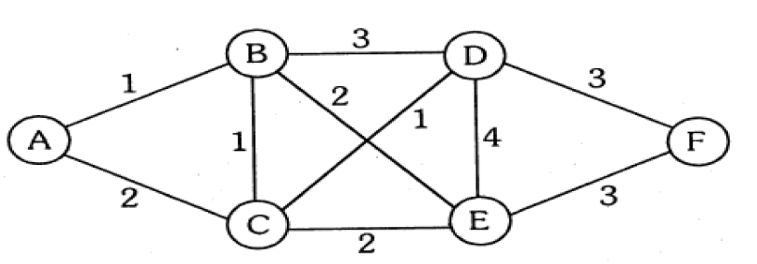
1. Analysis in terms of complexity wherever applicable.

**Time Complexity:**

* **Kruskal’s Algorithm**: O(ElogE) due to sorting of edges.
* **Prim’s Algorithm**: O(ElogV) using a priority queue.

**Problem Statement:**

1. **From a given vertex in a weighted connected graph, implement shortest path finding Dijkstra's algorithm.**

****

**Q) Show that Dijkstra’s algorithm doesn’t work for graphs with negative weight edges**

**Q) Modify the Dijkstra’s algorithm to find shortest path.**

1. Algorithm/Pseudocode

**Input**:

* A graph G=(V,E) where:
  + V is the set of vertices.
  + E is the set of edges with non-negative weights.
* A source vertex sr.

**Initialize**:

* Create a distance array dist[] where dist[i] represents the shortest distance from the source vertex src to vertex iii.
  + Set dist[src] = 0 (distance to the source is zero).
  + Set dist[v] = ∞ for all other vertices v.
* Create a priority queue (min-heap) to store vertices by their current shortest distance.
  + Push the source vertex with distance 0 into the queue.**Relaxation Loop**:
* While the priority queue is not empty:
  1. Extract the vertex u from the queue with the smallest distance.
  2. For each neighboring vertex v of u, check if there is a shorter path through u by evaluating the edge u→v with weight www:
     + If dist[u] + w < dist[v], update dist[v] = dist[u] + w.
     + Push v with its new distance into the priority queue.

**End of Algorithm**:

* When the priority queue is empty, all vertices have been processed and the dist[] array contains the shortest distances from src to all other vertices.

**Output**:

* Print the shortest distances stored in dist[] for each vertex.

Program Code

Dijkstra’s Algorithm :

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

#include <unordered\_map>

using namespace std;

struct Edge {

    int destination;

    int weight;

};

void dijkstra(const unordered\_map<string, int>& *vertexMap*, const vector<vector<Edge>>& *graph*, const string& *startVertex*) {

    int vertices = *graph*.size();

    vector<int> distances(vertices, numeric\_limits<int>::max());

    distances[*vertexMap*.at(*startVertex*)] = 0;

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    pq.push({0, *vertexMap*.at(*startVertex*)});

    while (!pq.empty()) {

        int currentDistance = pq.top().first;

        int currentVertex = pq.top().second;

        pq.pop();

        if (currentDistance > distances[currentVertex]) {

            continue;

        }

        for (const auto& edge : *graph*[currentVertex]) {

            int neighbor = edge.destination;

            int weight = edge.weight;

            int newDistance = currentDistance + weight;

            if (newDistance < distances[neighbor]) {

                distances[neighbor] = newDistance;

                pq.push({newDistance, neighbor});

            }

        }

    }

    cout << "Shortest distances from vertex " << *startVertex* << ":\n";

    for (const auto& [vertex, index] : *vertexMap*) {

        if (distances[index] == numeric\_limits<int>::max()) {

            cout << "Vertex " << vertex << " is unreachable." << endl;

        } else {

            cout << "Distance to vertex " << vertex << " is " << distances[index] << endl;

        }

    }

}

int main() {

    int vertices, edgesCount;

    cout << "Enter number of vertices: ";

    cin >> vertices;

    vector<vector<Edge>> graph(vertices);

    unordered\_map<string, int> vertexMap;

    cout << "Enter vertices (numbers or single-letter characters):\n";

    for (int i = 0; i < vertices; i++) {

        string vertex;

        cin >> vertex;

        vertexMap[vertex] = i;

    }

    cout << "Enter number of edges: ";

    cin >> edgesCount;

    cout << "Enter edges in the format (src dest weight):\n";

    for (int i = 0; i < edgesCount; i++) {

        string src, dest;

        int weight;

        cin >> src >> dest >> weight;

        graph[vertexMap[src]].push\_back({vertexMap[dest], weight});

        graph[vertexMap[dest]].push\_back({vertexMap[src], weight});

    }

    string startVertex;

    cout << "Enter starting vertex: ";

    cin >> startVertex;

    dijkstra(vertexMap, graph, startVertex);

    return 0;

}

Shortest Path :

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

#include <unordered\_map>

#include <string>

#include <algorithm>

using namespace std;

struct Edge {

    int destination;

    int weight;

};

void dijkstra(const unordered\_map<string, int>& *vertexMap*, const vector<vector<Edge>>& *graph*, const string& *startVertex*) {

    int vertices = *graph*.size();

    vector<int> distances(vertices, numeric\_limits<int>::max());

    vector<int> predecessors(vertices, -1);

    distances[*vertexMap*.at(*startVertex*)] = 0;

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    pq.push({0, *vertexMap*.at(*startVertex*)});

    while (!pq.empty()) {

        int currentDistance = pq.top().first;

        int currentVertex = pq.top().second;

        pq.pop();

        if (currentDistance > distances[currentVertex]) {

            continue;

        }

        for (const auto& edge : *graph*[currentVertex]) {

            int neighbor = edge.destination;

            int weight = edge.weight;

            int newDistance = currentDistance + weight;

            if (newDistance < distances[neighbor]) {

                distances[neighbor] = newDistance;

                predecessors[neighbor] = currentVertex;

                pq.push({newDistance, neighbor});

            }

        }

    }

    cout << "Shortest distances from vertex " << *startVertex* << ":\n";

    for (const auto& [vertex, index] : *vertexMap*) {

        if (distances[index] == numeric\_limits<int>::max()) {

            cout << "Vertex " << vertex << " is unreachable." << endl;

        } else {

            cout << "Distance to vertex " << vertex << " is " << distances[index] << ". Path: ";

            vector<string> path;

            for (int v = index; v != -1; v = predecessors[v]) {

                for (const auto& pair : *vertexMap*) {

                    if (pair.second == v) {

                        path.push\_back(pair.first);

                        break;

                    }

                }

            }

            reverse(path.begin(), path.end());

            for (const string& p : path) {

                cout << p << " ";

            }

            cout << endl;

        }

    }

}

int main() {

    int vertices, edgesCount;

    cout << "Enter number of vertices: ";

    cin >> vertices;

    vector<vector<Edge>> graph(vertices);

    unordered\_map<string, int> vertexMap;

    cout << "Enter vertices (numbers or single-letter characters):\n";

    for (int i = 0; i < vertices; i++) {

        string vertex;

        cin >> vertex;

        vertexMap[vertex] = i;

    }

    cout << "Enter number of edges: ";

    cin >> edgesCount;

    cout << "Enter edges in the format (src dest weight):\n";

    for (int i = 0; i < edgesCount; i++) {

        string src, dest;

        int weight;

        cin >> src >> dest >> weight;

        graph[vertexMap[src]].push\_back({vertexMap[dest], weight});

        graph[vertexMap[dest]].push\_back({vertexMap[src], weight});

    }

    string startVertex;

    cout << "Enter starting vertex: ";

    cin >> startVertex;

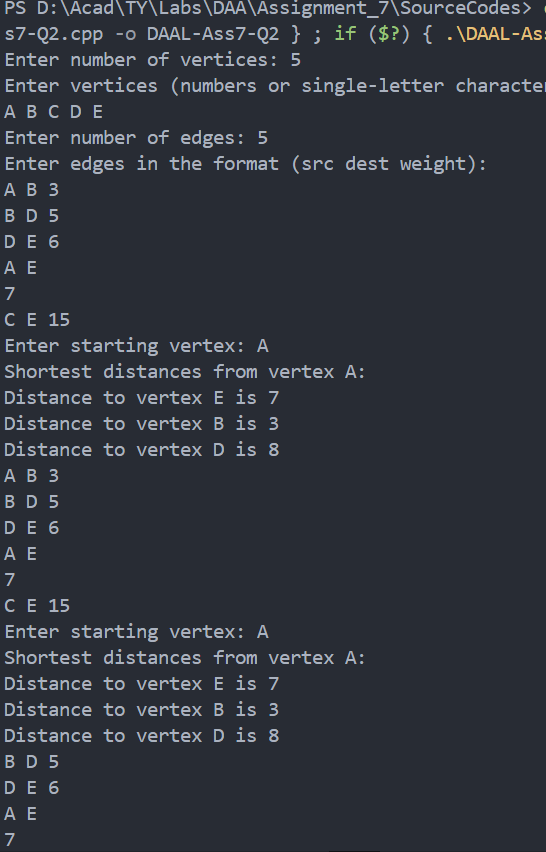
    dijkstra(vertexMap, graph, startVertex);

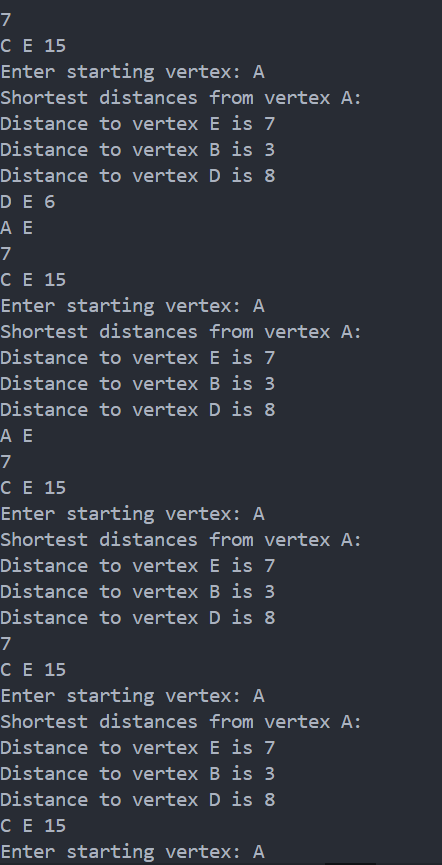
    return 0;

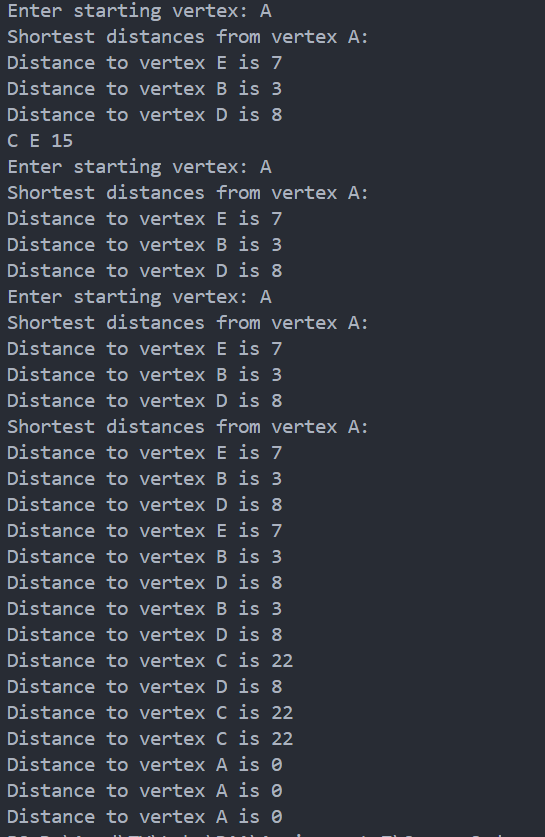
}

1. Output with verity of test cases

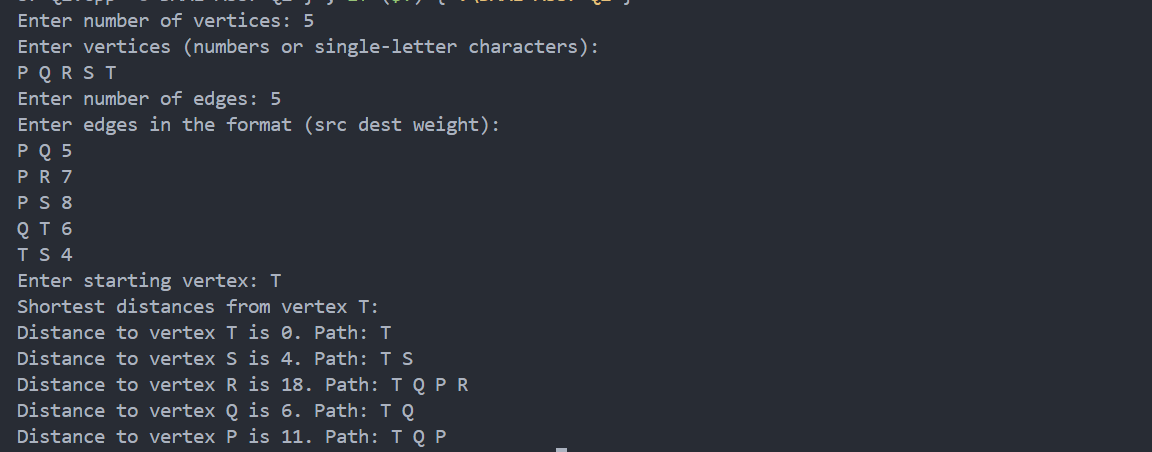
Dijkstra :







Shortest path :



1. Analysis in terms of complexity wherever applicable.

| **Algorithm** | **Graph Type** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- | --- |
| **Dijkstra** | Non-negative edges | O((V + E) | O(V+E) |
| **Bellman-Ford** | Negative edges allowed | O(V×E) | O(V) |