Batch:T6

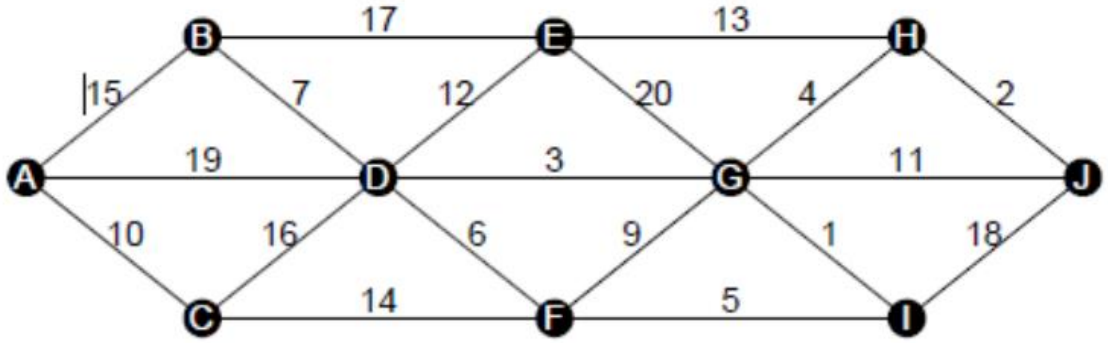
Practical No.7

Title of Assignment: Greedy approach

Student Name: Parshwa Herwade

Student PRN: 22510064

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?

Ans. A minimum spanning tree (MST) for a graph with V vertices always has V - 1 edges. In this graph, the number of vertices is 10 (A, B, C, D, E, F, G, H, I, J), so the MST will have:

V−1=10−1=9 edges.V - 1 = 10 - 1 = 9 \text{ edges}.V−1=10−1=9 edges.

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.

Ans. No, the edge with the least weight is not always part of the MST. While it is usually considered in MST algorithms like Kruskal's, it is not guaranteed to be included if adding that edge creates a cycle, as the MST must be acyclic.

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?

Ans. If the weight of an edge in the MST is decreased, the current MST remains valid. However, we need to check if this updated edge creates a more optimal MST by removing another heavier edge in any cycles formed. This can be efficiently done by replacing the heavier edge using algorithms like Kruskal's or Prim's.

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

Ans. For **Kruskal's algorithm**, the order is based on sorting all edges by their weights and then processing them one by one.

For **Prim's algorithm**, the order of edges will be based on starting from any vertex and greedily expanding the MST using the minimum-weight edge.

Pseudocode for Kruskal's Algorithm:

```
KRUSKAL(G, V):
  Sort all edges in non-decreasing order of their weights
  Initialize Disjoint Set (Union-Find)
  MST = []
  for each edge (u, v) in sorted edge list:
    if find(u) != find(v):
      MST.append((u, v))
      union(u, v)
  return MST
```

Pseudocode for Prim's Algorithm:

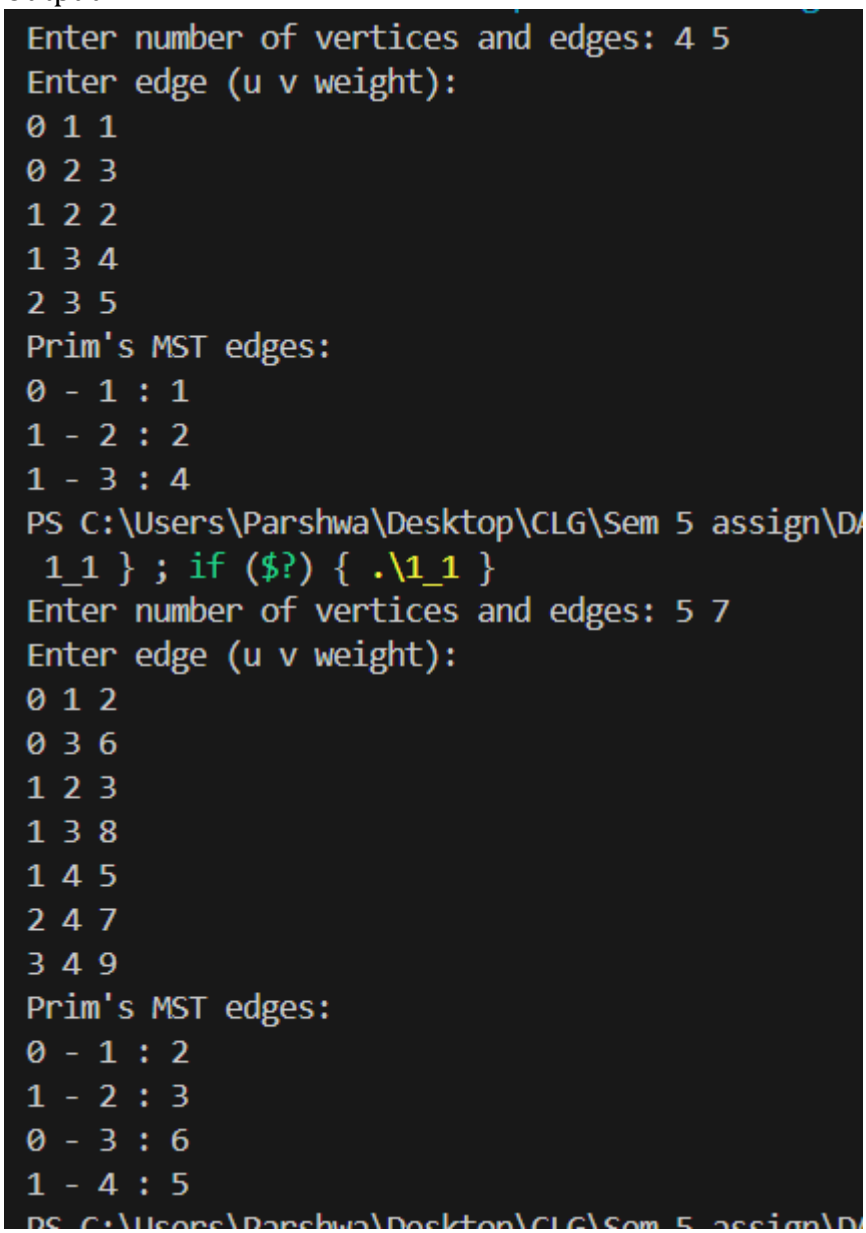
```
PRIM(G, V):
  Initialize key[] to infinity, parent[] to -1, inMST[] to false
  key[0] = 0
  for count = 0 to V-1:
    u = vertex with minimum key[] value not in MST
    inMST[u] = true
    for each v adjacent to u:
      if weight(u, v) < key[v] and v is not in MST:
        parent[v] = u
        key[v] = weight(u, v)
  return parent[]
```

CODE:

```
//Prim's algo

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

void prim(int V, vector<vector<pair<int, int>>>& adj) {
    vector<int> key(V, INT_MAX);
    vector<int> parent(V, -1);
    vector<bool> inMST(V, false);
    key[0] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = -1;
        for (int i = 0; i < V; i++) {
            if (!inMST[i] && (u == -1 || key[i] < key[u])) {
                u = i;
            }
        }
        inMST[u] = true;

        for (auto& neighbor : adj[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
            }
        }
    }

    cout << "Prim's MST edges:\n";
    for (int i = 1; i < V; i++) {
        cout << parent[i] << " - " << i << " : " << key[i] << endl;
    }
}

int main() {
    int V, E;
    cout << "Enter number of vertices and edges: ";
```

```
    cin >> V >> E;
    vector<vector<pair<int, int>>> adj(V);

    cout << "Enter edge (u v weight):\n";
    for (int i = 0; i < E; i++) {
        int u, v, weight;
        cin >> u >> v >> weight;
        adj[u].push_back({v, weight});
        adj[v].push_back({u, weight});
    }

    prim(V, adj);
    return 0;
}
```

Output:


Enter number of vertices and edges: 4 5
Enter edge (u v weight):
0 1 1
0 2 3
1 2 2
1 3 4
2 3 5
Prim's MST edges:
0 - 1 : 1
1 - 2 : 2
1 - 3 : 4
PS C:\Users\Parshwa\Desktop\CLG\Sem 5 assign\DA
1_1 } ; if ($?) { .\1_1 }
Enter number of vertices and edges: 5 7
Enter edge (u v weight):
0 1 2
0 3 6
1 2 3
1 3 8
1 4 5
2 4 7
3 4 9
Prim's MST edges:
0 - 1 : 2
1 - 2 : 3
0 - 3 : 6
1 - 4 : 5

```
Enter number of vertices and edges: 6 9
Enter edge (u v weight):
0 1 4
0 2 2
1 2 1
1 3 7
2 4 3
3 4 8
3 5 5
4 5 6
1 5 9
Prim's MST edges:
2 - 1 : 1
0 - 2 : 2
5 - 3 : 5
2 - 4 : 3
4 - 5 : 6
PS C:\Users\Parshwa\Desktop\CLG\Sem 5 assign
```

Code:

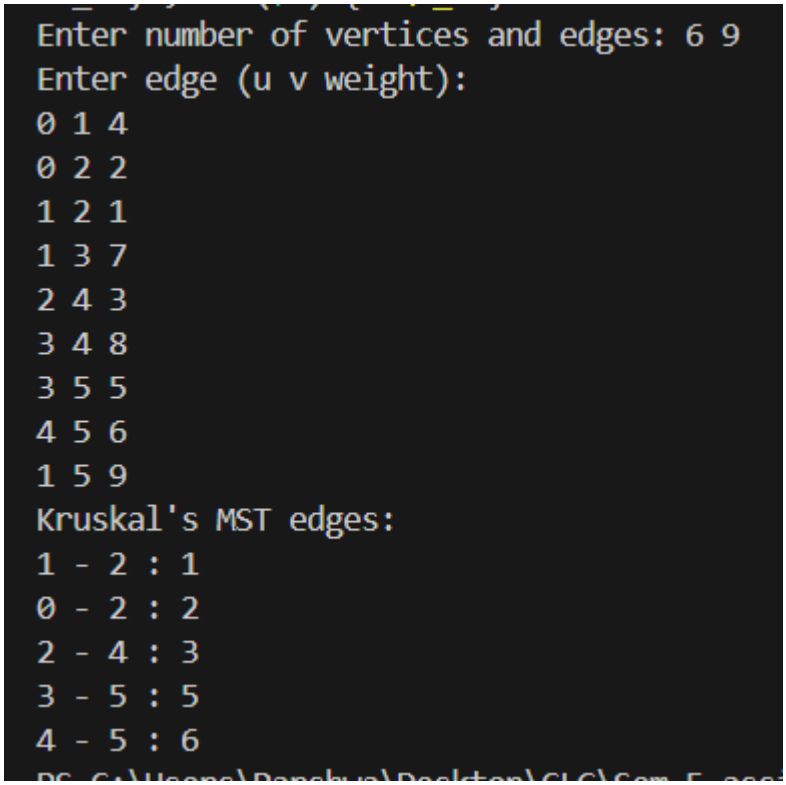
```cpp
//Kruskal's algo

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

struct Edge {
    int u, v, weight;
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

class DisjointSet {
public:
    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 unite(int u, int v) {
        int root_u = find(u);
        int root_v = find(v);

        if (root_u != root_v) {
            if (rank[root_u] > rank[root_v]) {
                parent[root_v] = root_u;
            } else if (rank[root_u] < rank[root_v]) {
                parent[root_u] = root_v;
            } else {
                parent[root_v] = root_u;
                rank[root_u]++;
            }
        }
    }
};

void kruskal(int V, vector<Edge>& edges) {
    sort(edges.begin(), edges.end());
    DisjointSet ds(V);
    vector<Edge> mst;

    for (auto& edge : edges) {
        if (ds.find(edge.u) != ds.find(edge.v)) {
            mst.push_back(edge);
            ds.unite(edge.u, edge.v);
        }
    }

    cout << "Kruskal's MST edges:\n";
    for (auto& edge : mst) {
```

```
        cout << edge.u << " - " << edge.v << " : " << edge.weight <<
endl;
    }
}

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

    vector<Edge> edges;
    cout << "Enter edge (u v weight):\n";
    for (int i = 0; i < E; i++) {
        int u, v, weight;
        cin >> u >> v >> weight;
        edges.push_back({u, v, weight});
    }

    kruskal(V, edges);
    return 0;
}
```

Output:

```
Enter number of vertices and edges: 4 5
Enter edge (u v weight):
0 1 1
0 2 3
1 2 2
1 3 4
2 3 5
Kruskal's MST edges:
0 - 1 : 1
1 - 2 : 2
1 - 3 : 4
PS C:\Users\Parshwa\Desktop\CLG\Sem 5 assign
 1_2 } ; if ($?) { .\1_2 }
Enter number of vertices and edges: 5 7
Enter edge (u v weight):
0 1 2
0 3 6
1 2 3
1 3 8
1 4 5
2 4 7
3 4 9
Kruskal's MST edges:
0 - 1 : 2
1 - 2 : 3
1 - 4 : 5
0 - 3 : 6
```



Time and Space Complexities:

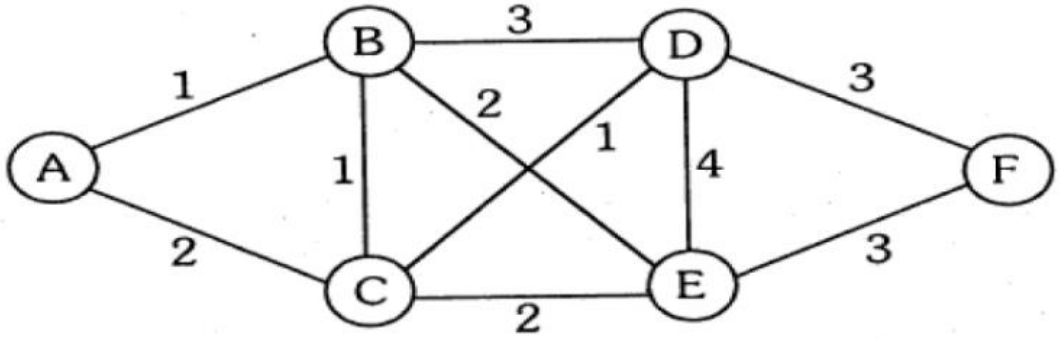
Kruskal's Algorithm:

- **Time Complexity**: O(E log E) (sorting the edges), where E is the number of edges.
- **Space Complexity**: O(V + E) (for storing edges and union-find data structure).

Prim's Algorithm:

- **Time Complexity**: O(V^2) using adjacency matrix, O(E log V) with min-heap.
- **Space Complexity**: O(V^2) (adjacency matrix) or O(V + E) (adjacency list with heap).

2) 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.

Pseudocode for Dijkstra's Algorithm:

```
DIJKSTRA(G, V, src):
   dist[] = {∞, ∞, ..., ∞}  // Initialize all distances to infinity
   dist[src] = 0            // Distance to source is 0
   Set S = empty            // Set of processed vertices
   Priority Queue PQ        // Priority queue to store {distance, vertex}

   PQ.insert({0, src})      // Insert source vertex with distance 0

   while PQ is not empty:
      u = vertex with smallest dist[] value from PQ
      S.insert(u)           // Mark u as processed
      for each neighbor v of u:
         if v is not in S and dist[u] + weight(u, v) < dist[v]:
            dist[v] = dist[u] + weight(u, v)
            PQ.update(v, dist[v])

   return dist[]
```

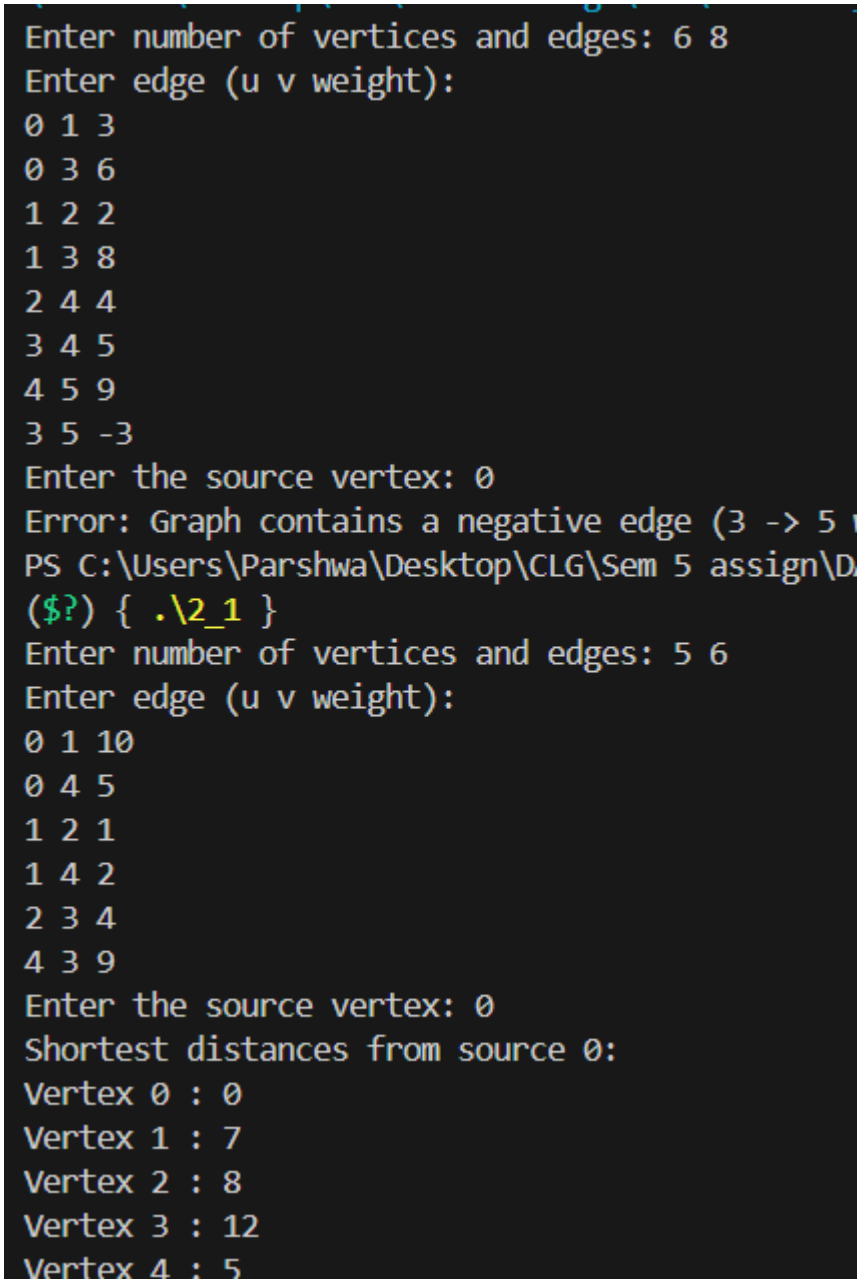
Code:

```
//Dijkstra's algo --> shortest path
```

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

void dijkstra(int V, vector<vector<pair<int, int>>>& adj, int src) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        int u = pq.top().second;
        int d = pq.top().first;
        pq.pop();

        if (d > dist[u]) continue;

        for (auto& neighbor : adj[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            // Check for negative edge weights
            if (weight < 0) {
                cout << "Error: Graph contains a negative edge (" <<
u << " -> " << v
                     << " with weight " << weight << "). Dijkstra's
algorithm cannot handle negative weights.\n";
                return;  // Exit the function early
            }

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
```

```
    cout << "Shortest distances from source " << src << ":\n";
    for (int i = 0; i < V; i++) {
        if (dist[i] == INT_MAX) {
            cout << "Vertex " << i << " : Unreachable\n";
        } else {
            cout << "Vertex " << i << " : " << dist[i] << endl;
        }
    }
}

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

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

    cout << "Enter edge (u v weight):\n";
    for (int i = 0; i < E; i++) {
        int u, v, weight;
        cin >> u >> v >> weight;
        adj[u].push_back({v, weight});
        adj[v].push_back({u, weight});
    }

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

    dijkstra(V, adj, src);
    return 0;
}
```

Output:


Enter number of vertices and edges: 6 8
Enter edge (u v weight):
0 1 3
0 3 6
1 2 2
1 3 8
2 4 4
3 4 5
4 5 9
3 5 -3
Enter the source vertex: 0
Error: Graph contains a negative edge (3 -> 5
PS C:\Users\Parshwa\Desktop\CLG\Sem 5 assign\D
($?) { .\2_1 }
Enter number of vertices and edges: 5 6
Enter edge (u v weight):
0 1 10
0 4 5
1 2 1
1 4 2
2 3 4
4 3 9
Enter the source vertex: 0
Shortest distances from source 0:
Vertex 0 : 0
Vertex 1 : 7
Vertex 2 : 8
Vertex 3 : 12
Vertex 4 : 5

```
Enter number of vertices and edges: 7 9
Enter edge (u v weight):
0 1 2
0 2 4
1 3 7
1 4 2
2 4 1
3 5 3
4 5 2
4 6 5
5 6 4
Enter the source vertex: 0
Shortest distances from source 0:
Vertex 0 : 0
Vertex 1 : 2
Vertex 2 : 4
Vertex 3 : 9
Vertex 4 : 4
Vertex 5 : 6
Vertex 6 : 9
PS C:\Users\Parshwa\Desktop\CLG\Sem 5 ass
```

Code:

```cpp
//Modified dijkstra-->Bellman-Ford

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

bool bellmanFord(int V, int E, vector<vector<int>> &edges, int
source) {
    vector<int> dist(V, INT_MAX);
    dist[source] = 0;

    for (int i = 0; i < V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = edges[j][0];
            int v = edges[j][1];
            int weight = edges[j][2];
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
```

```
                dist[v] = dist[u] + weight;
            }
        }
    }

    for (int i = 0; i < E; i++) {
        int u = edges[i][0];
        int v = edges[i][1];
        int weight = edges[i][2];
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            cout << "Graph contains a negative weight cycle.\n";
            return false;
        }
    }

    cout << "Vertex\tDistance from Source\n";
    for (int i = 0; i < V; i++) {
        if (dist[i] == INT_MAX) {
            cout << i << "\t\t" << "INF" << endl;
        } else {
            cout << i << "\t\t" << dist[i] << endl;
        }
    }
    return true;
}

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

    vector<vector<int>> edges(E, vector<int>(3));

    cout << "Enter each edge (u v weight):\n";
    for (int i = 0; i < E; i++) {
        cin >> edges[i][0] >> edges[i][1] >> edges[i][2];
    }

    cout << "Enter source vertex: ";
    cin >> source;
```

```
    bool noNegativeCycle = bellmanFord(V, E, edges, source);
    if (noNegativeCycle) {
        cout << "Shortest path calculated successfully.\n";
    } else {
        cout << "Failed due to negative weight cycle.\n";
    }
    return 0;
}
```

Output:

```
Enter the number of vertices and edges: 5 8
Enter each edge (u v weight):
0 1 -1
0 2 4
1 2 3
1 3 2
1 4 2
3 2 5
3 1 1
4 3 -3
Enter source vertex: 0
Vertex  Distance from Source
0               0
1               -1
2               2
3               -2
4               1
Shortest path calculated successfully.
PS C:\Users\Parshwa\Desktop\CLG\Sem 5 assign\DAA\225100
($?) { .\2_2 }
Enter the number of vertices and edges: 4 4
Enter each edge (u v weight):
0 1 1
1 2 -1
2 3 -1
3 1 -1
Enter source vertex: 0
Graph contains a negative weight cycle.
Failed due to negative weight cycle.
PS C:\Users\Parshwa\Desktop\CLG\Sem 5 assign\DAA\225100
```

Time and Space Complexities:

Dijkstra's Algorithm:

- **Time Complexity**: O((V + E) log V) using a priority queue (min-heap), where V is the number of vertices and E is the number of edges.
- **Space Complexity**: O(V + E) for storing the adjacency list and distance array.