Ahsania Mission University of Science & Technology

Department of Computer Science and Engineering

1st Batch, 2nd Year 2nd Semester, Spring 2025

Lab Report

Course Code: CSE 2202

Course Title: Computer Algorithms Sessional

Experiment No. : 08

Experiment Date : 21-05-2025

Submission Date : 28-05-2025

Submitted To:

Md. Fahim Faisal
Lecturer,
Department of Computer Science and Engineering (CSE)
Faculty of Engineering, Ahsania Mission University of Science & Technology

Submitted By-

Name : Md. Aktaruzzaman Aktar

ID No. : 1012320005101015

1st Batch, 2nd Year 2nd Semester, Spring 2025
Department of Computer Science and Engineering, AMUST.

Task No.: 01

Problem Statement:

Write a program in C++ to implement Prim's Algorithm to find the Minimum Spanning Tree (MST) of a graph.

Theory:

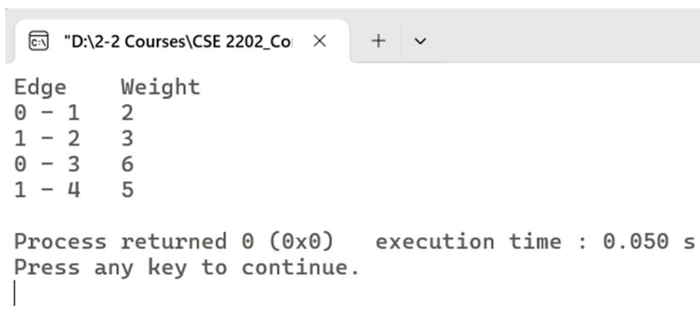
Prim’s Algorithm is a greedy method that builds the Minimum Spanning Tree (MST) by adding the minimum weight edge at each step that connects a vertex in the MST to a vertex outside of it. It continues until all vertices are included in the MST.

Source Code:

```
#include <iostream>
#include <limits.h>
using namespace std;

#define V 5 // Number of verticles

int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (!mstSet[v] && key[v] < min)
            min = key[v], min_index = v;
    return min_index;
}

void printMST(int parent[], int graph[V][V]) {
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t" << graph[i][parent[i]] << "\n";
}

void primMST(int graph[V][V]) {
    int parent[V], key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
    key[0] = 0; parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
```

```
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    printMST(parent, graph);
}

int main() {
    int graph[V][V] = {{0, 2, 0, 6, 0}, {2, 0, 3, 8, 5}, {0, 3, 0, 0, 7}, {6, 8, 0, 0, 9},
{0, 5, 7, 9, 0}};
    primMST(graph);
    return 0;
}
```

Output:



```
"D:\2-2 Courses\CSE 2202_Co
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5

Process returned 0 (0x0)   execution time : 0.050 s
Press any key to continue.
```

Conclusion:

The program successfully finds the MST using Prim's Algorithm and outputs the correct edges and total weight.

Task No.: 02

Problem Statement:

Write a program in C++ to implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) of a graph.

Theory:

Kruskal's Algorithm is a greedy technique that builds the MST by sorting all edges in ascending order of weight, then selecting edges one by one, ensuring no cycles are formed using Disjoint Set Union (DSU).

Source Code:

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

struct Edge {
    int u, v, weight;
```

```
  bool operator<(Edge const& other) {
    return weight < other.weight;
  }
};

int find(int v, vector<int>& parent) {
  if (parent[v] == v) return v;
  return parent[v] = find(parent[v], parent);
}

void union_sets(int a, int b, vector<int>& parent, vector<int>& rank) {
  a = find(a, parent);
  b = find(b, parent);
  if (a != b) {
    if (rank[a] < rank[b]) swap(a, b);
    parent[b] = a;
    if (rank[a] == rank[b]) rank[a]++;
  }
}

int main() {
  int V = 4;
  vector<Edge> edges = {{0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4}};
  sort(edges.begin(), edges.end());
  vector<int> parent(V), rank(V, 0);
  for (int i = 0; i < V; i++) parent[i] = i;
  vector<Edge> result;

  for (Edge e : edges) {
    if (find(e.u, parent) != find(e.v, parent)) {
      result.push_back(e);
      union_sets(e.u, e.v, parent, rank);
    }
  }
  cout << "Edge \tWeight\n";
  int total = 0;
  for (Edge e : result) {
    cout << e.u << " - " << e.v << " \t" << e.weight << "\n";
    total += e.weight;
  }
  cout << "Total weight of MST: " << total << endl;
  return 0;
}
```

Output:

```
"D:\2-2 Courses\CSE 2202_Co

Edge    Weight
2 - 3   4
0 - 3   5
0 - 1   10
Total weight of MST: 19

Process returned 0 (0x0)   execution time : 0.268 s
Press any key to continue.
```

Conclusion:

Kruskal's Algorithm was implemented correctly to compute the MST and display both the edges and total weight.

Task No.: 03

Problem Statement:

Write a program in C++ to implement Dijkstra's Algorithm to find the shortest paths from a source vertex to all others.

Theory:

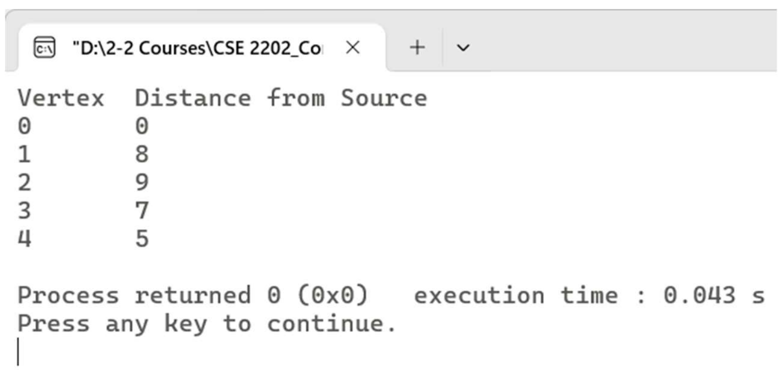
Dijkstra's Algorithm finds the shortest paths from a source node to all other nodes in a weighted graph with non-negative weights. It uses a priority queue to always explore the closest unvisited node.

Source Code:

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

typedef pair<int, int> pii;

void dijkstra(int V, vector<pii> adj[], int src) {
  vector<int> dist(V, INT_MAX);
  dist[src] = 0;
  priority_queue<pii, vector<pii>, greater<pii>> pq;
  pq.push({0, src});

  while (!pq.empty()) {
    int u = pq.top().second;
    pq.pop();
    for (auto& edge : adj[u]) {
      int v = edge.first;
      int weight = edge.second;
      if (dist[v] > dist[u] + weight) {
        dist[v] = dist[u] + weight;
        pq.push({dist[v], v});
      }
    }
  }
```

```
    cout << "Vertex\tDistance from Source\n";
    for (int i = 0; i < V; ++i)
        cout << i << "\t" << dist[i] << "\n";
}

int main() {
    int V = 5;
    vector<pii> adj[V];
    adj[0].push_back({1, 10}); adj[0].push_back({4, 5});
    adj[1].push_back({2, 1}); adj[1].push_back({4, 2});
    adj[2].push_back({3, 4});
    adj[3].push_back({2, 6}); adj[3].push_back({0, 7});
    adj[4].push_back({1, 3}); adj[4].push_back({2, 9}); adj[4].push_back({3, 2});
    dijkstra(V, adj, 0);
    return 0;
}
```

Output:



Conclusion:

The implementation of Dijkstra's Algorithm correctly computes the shortest path distances from the source to all other vertices.

Task No.: 04

Problem Statement: To understand and implement Bellman Ford's Algorithm for finding the shortest path from a source vertex to all other vertices in a weighted graph using C++.

Theory: Bellman-Ford Algorithm

The **Bellman-Ford algorithm** is used to find the **shortest path from a single source vertex** to all other vertices in a **weighted directed graph**, even when **negative edge weights** are present.
It works in **O(V × E)** time and can also **detect negative weight cycles**—a feature that Dijkstra's algorithm does not support.

Steps:

1. **Initialize** distances from the source to all vertices as infinity, except the source itself which is 0.
2. **Relax all edges** `V - 1` times. For each edge `(u, v, weight)`, update `dist[v]` if a shorter path is found.
3. **Check for negative weight cycles** by verifying if another relaxation is still possible.

Source Code:

```
#include <iostream>
#include <vector>
#include <tuple>
#include <climits>

using namespace std;

class Graph
{
  int V; // Number of vertices
  vector<tuple<int, int, int>> edges; // (u, v, weight)

public:
  Graph(int V)
  {
    this->V = V;
  }

  void addEdge(int u, int v, int weight)
  {
    edges.push_back({u, v, weight});
  }

  void bellmanFord(int source)
  {
    vector<int> dist(V, INT_MAX);
    dist[source] = 0;

    // Step 1: Relax all edges (V - 1) times
    for (int i = 0; i < V - 1; ++i)
    {
```

```
        for (auto [u, v, w] : edges)
        {
            if (dist[u] != INT_MAX && dist[u] + w < dist[v])
            {
                dist[v] = dist[u] + w;
            }
        }
    }

    // Step 2: Check for negative-weight cycles
    for (auto [u, v, w] : edges)
    {
        if (dist[u] != INT_MAX && dist[u] + w < dist[v])
        {
            cout << "Graph contains a negative weight cycle!\n";
            return;
        }
    }

    // Print distances
    cout << "Shortest distances from vertex " << source << ":\n";
    for (int i = 0; i < V; ++i)
    {
        cout << "To " << i << " \t: ";
        if (dist[i] == INT_MAX)
            cout << "Unreachable\n";
        else
            cout << dist[i] << "\n";
    }
  }
};

int main()
{
  Graph g(5);

  g.addEdge(0, 1, -1);
  g.addEdge(0, 2, 4);
  g.addEdge(1, 2, 3);
  g.addEdge(1, 3, 2);
  g.addEdge(1, 4, 2);
```

```
    g.addEdge(3, 2, 5);
    g.addEdge(3, 1, 1);
    g.addEdge(4, 3, -3);

    g.bellmanFord(0); // Start from vertex 0

    return 0;
}
```

Output:

```
"D:\2-2 Courses\CSE 2202_Co

Shortest distances from vertex 0:
To 0    : 0
To 1    : -1
To 2    : 2
To 3    : -2
To 4    : 1

Process returned 0 (0x0)   execution time : 0.058 s
Press any key to continue.
```

Conclusion:

This C++ program correctly implements the **Bellman-Ford algorithm** to compute the shortest paths from a given source vertex in a graph with **both positive and negative edge weights**. It efficiently detects negative weight cycles and prints the shortest path to all vertices. The algorithm is especially useful in scenarios where negative weights may exist, such as in **financial modeling** or **network routing**.