**EXPERIMENT – 5**

**AIM:** Implementing the PRIM’s algorithm to create MST (Minimum Spanning Tree).

**SOURCE CODE:**

#include <iostream>

#include <unordered\_map>

#include <vector>

#include <queue>

#include <chrono>

#include <limits.h>

using namespace std;

typedef pair<int, int> iPair;

void addEdge(unordered\_map<int, vector<iPair>> &graph, int u, int v, int w)

{

    graph[u].push\_back(make\_pair(v, w));

    graph[v].push\_back(make\_pair(u, w));

}

void PRIMsMST(unordered\_map<int, vector<iPair>> &graph, int start\_node)

{

    priority\_queue<iPair, vector<iPair>, greater<iPair>> pq;

    vector<int> key(graph.size(), INT\_MAX),

parent(graph.size(), -1),

weight(graph.size(), 0);

    vector<bool> inMST(graph.size(), false);

    pq.push(make\_pair(0, start\_node));

    key[start\_node] = 0;

    while (!pq.empty())

    {

        int u = pq.top().second;

        pq.pop();

        inMST[u] = true;

        for (auto &x : graph[u])

        {

            int v = x.first, w = x.second;

            if (inMST[v] == false && key[v] > w)

            {

                key[v] = w;

                pq.push(make\_pair(key[v], v));

                parent[v] = u;

                weight[v] = w;

            }

        }

    }

    cout << "Parent\tChild\tWeight\n";

    for (int i = 1; i < graph.size(); ++i)

        cout << parent[i] << "\t" << i << "\t" << weight[i] << "\n";

}

int main()

{

    unordered\_map<int, vector<iPair>> graph;

    vector<vector<int>> edges = {{0, 1, 2}, {0, 3, 6}, {1, 2, 3},

{1, 3, 8}, {1, 4, 5}, {2, 4, 7}, {3, 4, 9}};

    for (auto &edge : edges)

        addEdge(graph, edge[0], edge[1], edge[2]);

    auto start\_time = chrono::high\_resolution\_clock::now();

    PRIMsMST(graph, 0);

    auto duration = chrono::duration\_cast<chrono::microseconds>

(chrono::high\_resolution\_clock::now() - start\_time);

    cout << "\nExecution time: " << duration.count() << " microseconds\n";

    return 0;

}

**OUTPUT:**

Parent Child Weight

0 1 2

1 2 3

0 3 6

1 4 5

Execution time: 1238 microseconds

**VIVA QUESTIONS:**

**Q1.** **What is a Minimum Spanning Tree (MST)?**

**Solution:** A Minimum Spanning Tree is a subset of the edges of a connected, undirected graph that connects all the vertices together without any cycles and with the minimum possible total edge weight.

**Q2.** **Why are Minimum Spanning Trees important in graph theory and real-world applications?**

**Solution:** MSTs have applications in network design, circuit layout, transportation, and communication networks, where the goal is to connect all nodes while minimizing the total cost or distance.

**Q3.** **Explain the basic idea behind Prim's algorithm.**

**Solution:** Prim's algorithm starts with an arbitrary node and repeatedly adds the shortest edge that connects a vertex in the growing MST to a vertex outside the MST until all vertices are included.

**Q4. How does Prim's algorithm ensure that the result is a Minimum Spanning Tree?**

**Solution:** Prim's algorithm ensures a Minimum Spanning Tree by always selecting the shortest available edge at each step, which guarantees that the total weight of the selected edges is minimized.

**Q5.** **What is the time complexity of Prim's algorithm?**

**Solution:** The time complexity of Prim's algorithm is O(V^2) with an adjacency matrix representation and can be optimized to O(E + V log V) with a priority queue and an adjacency list representation, where V is the number of vertices and E is the number of edges**EXPERIMENT – 6**

**AIM:** Implementing the DIJKSTRA’s algorithm to find the shortest distance of a node from the source node.

**SOURCE CODE:**

#include <iostream>

#include <unordered\_map>

#include <vector>

#include <queue>

#include <limits.h>

#include <chrono>

using namespace std;

typedef pair<int, int> iPair;

void addEdge(unordered\_map<int, vector<iPair>> &graph, int u, int v, int w)

{

    graph[u].push\_back(make\_pair(v, w));

    graph[v].push\_back(make\_pair(u, w));

}

void Dijkstras(unordered\_map<int, vector<iPair>> &graph, int start\_node)

{

    priority\_queue<iPair, vector<iPair>, greater<iPair>> pq;

    vector<int> dist(graph.size(), INT\_MAX);

    pq.push(make\_pair(0, start\_node));

    dist[start\_node] = 0;

    while (!pq.empty())

    {

        int u = pq.top().second;

        pq.pop();

        for (auto &x : graph[u])

        {

            int v = x.first, weight = x.second;

            if (dist[v] > dist[u] + weight)

            {

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

                pq.push(make\_pair(dist[v], v));

            }

        }

    }

    cout << "Vertex\tDistance from Source\n";

    for (int i = 0; i < graph.size(); ++i)

        cout << i << "\t" << dist[i] << "\n";

}

int main()

{

    unordered\_map<int, vector<iPair>> graph;

    vector<vector<int>> edges = {{0, 1, 2}, {0, 3, 6}, {1, 2, 3},

{1, 3, 8}, {1, 4, 5}, {2, 4, 7}, {3, 4, 9}};

    for (auto &edge : edges)

        addEdge(graph, edge[0], edge[1], edge[2]);

    auto start\_time = chrono::high\_resolution\_clock::now();

    Dijkstras(graph, 0);

    auto duration = chrono::duration\_cast<chrono::microseconds>

(chrono::high\_resolution\_clock::now() - start\_time);

    cout << "\nExecution time: " << duration.count() << " microseconds\n";

    return 0;

}

**OUTPUT:**

Vertex Distance from Source

0 0

1 2

2 5

3 6

4 7

Execution time: 1999 microseconds

**VIVA QUESTIONS:**

**Q1. Explain the basic idea behind Dijkstra's algorithm.**

**Solution:** Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a source node to all other nodes in a weighted graph. It maintains a set of vertices whose shortest distance from the source is known and repeatedly selects the vertex with the minimum distance, updating the distances of its neighbours.

**Q2. How does Dijkstra's algorithm handle weighted edges and find the shortest path?**

**Solution:** Dijkstra's algorithm maintains a distance array, initially set to infinity for all vertices except the source, and updates the distances based on the sum of the current distance and the weight of the edge. It always selects the vertex with the minimum distance in each iteration.

**Q3. Can Dijkstra's algorithm handle graphs with negative edge weights?**

**Solution:** No, Dijkstra's algorithm is designed for graphs with non-negative edge weights. Negative edge weights can lead to incorrect results because the algorithm assumes that each step taken is the optimal choice.

**Q4. What is the priority queue used for in Dijkstra's algorithm?**

**Solution:** Dijkstra's algorithm relies on a priority queue to efficiently select the vertex with the minimum distance in each iteration. The priority queue allows quick access to the vertex with the smallest known distance.

**Q5. What is the time complexity of Dijkstra's algorithm?**

**Solution:** The time complexity of Dijkstra's algorithm is O((V + E) log V) with a binary heap-based priority queue, where V is the number of vertices and E is the number of edges. It is efficient for sparse graphs.

**EXPERIMENT – 7**

**AIM:** An algorithm to perform the matrix multiplication.

**SOURCE CODE:**

#include <iostream>

#include <vector>

#include <sstream>

#include <chrono>

using namespace std;

vector<vector<int>> inputMatrix()

{

vector<vector<int>> matrix;

string line;

while (getline(cin, line))

{

if (line.empty())

break;

istringstream stream(line);

vector<int> row;

int n;

while (stream >> n)

row.push\_back(n);

matrix.push\_back(row);

}

return matrix;

}

vector<vector<int>> multiplyMatrices

(const vector<vector<int>> &a, const vector<vector<int>> &b)

{

int rows = a.size(), cols = b[0].size(), inner = b.size();

vector<vector<int>> product(rows, vector<int>(cols, 0));

for (int i = 0; i < rows; ++i)

for (int j = 0; j < cols; ++j)

for (int k = 0; k < inner; ++k)

product[i][j] += a[i][k] \* b[k][j];

return product;

}

void printMatrix(const vector<vector<int>> &matrix)

{

cout << "The product is:\n";

for (const auto &row : matrix)

{

for (const auto &elem : row)

cout << elem << '\t';

cout << '\n';

}

}

int main()

{

cout << "Enter elements of first matrix (empty line to end):\n";

auto matrix1 = inputMatrix();

cout << "Enter elements of second matrix (empty line to end):\n";

auto matrix2 = inputMatrix();

if (matrix1[0].size() != matrix2.size())

{

cout << "Matrix multiplication not possible.";

return 0;

}

auto start\_time = chrono::high\_resolution\_clock::now();

auto product = multiplyMatrices(matrix1, matrix2);

auto duration = chrono::duration\_cast<chrono::nanoseconds>

(chrono::high\_resolution\_clock::now() - start\_time);

printMatrix(product);

cout << "\nExecution time: " << duration.count() << " nanoseconds\n";

return 0;

}

**OUTPUT:**

Enter elements of first matrix (empty line to end):

10 11 12 13 14 15

20 21 22 23 24 25

30 31 32 33 34 35

40 41 42 43 44 45

Enter elements of second matrix (empty line to end):

10 11 12 13

20 21 22 23

30 31 32 33

40 41 42 43

50 51 52 53

60 61 62 63

The product is:

2800 2875 2950 3025

4900 5035 5170 5305

7000 7195 7390 7585

9100 9355 9610 9865

Execution time: 179999 nanoseconds