Ahsania Mission University of Science & Technology

Lab Report

Course Code: CSE 2202

Course Title: Computer Algorithm Sessional

Experiment No: 08

Experiment Date: 21.05.25

Submitted By:

Mehrin Nusrat Chowdhury

Roll: 1012320005101026

1st Batch, 2nd Year, 2nd Semester

Department of Computer Science and Engineering

Ahsania Mission University of Science & Technology

Submitted To:

Md. Fahim Faisal

Lecturer

Department of Computer Science and Engineering

Faculty of Engineering, Ahsania Mission University of Science & Technology

Submission Date: 28.05.25

Task 01: A C++ program that will implement Prim's Algorithm.

Theory:

Prim's Algorithm is a greedy approach used to find the Minimum Spanning Tree (MST) of a weighted, undirected graph. A spanning tree connects all vertices with the minimum total edge weight, ensuring no cycles. The algorithm begins with an arbitrary starting vertex and iteratively adds the smallest possible edge that connects a new vertex to the growing MST. This process continues until all vertices are included, guaranteeing an optimal solution.

The algorithm follows these steps:

1. Initialize a key[] array and set all values to ∞ (infinity). Set the key value of the starting vertex to 0.
2. Use a mstSet[] array to track vertices included in the MST, initializing all values as false.
3. Repeat for V-1 vertices:

- Pick the vertex u not in MST with the smallest key value.
- Include u in the MST.
- For all adjacent vertices v of u, if v is not in MST and the weight of (u,v) is less than key[v], update key[v] and set parent[v] as u.

By maintaining a set of vertices that are part of the MST and continuously selecting the least costly edge, Prim's Algorithm ensures that the final tree has the smallest total edge weight. This makes it an efficient and widely used approach for network optimization problems.

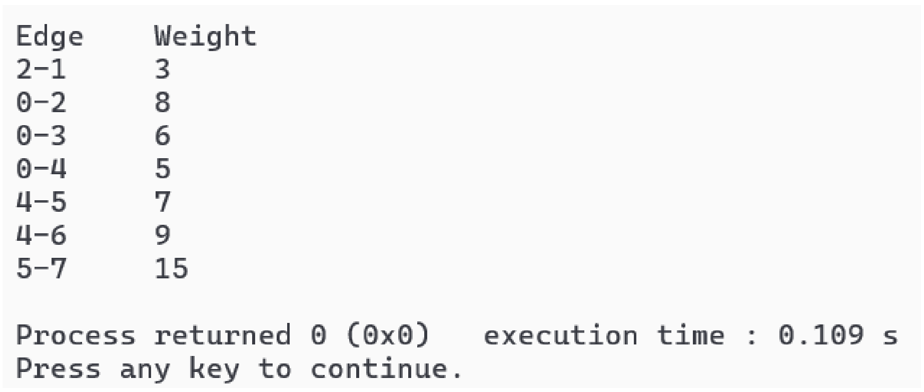
Code:

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

#define V 8  //Number of vertices

// Function to find the vertex with minimum key value
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;
}

// Function to print MST
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";
    }
}

// Prim's Algorithm
void primMST(int graph[V][V])
{
    int parent[V];
    int 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, 14, 8, 6, 5, 0, 0, 0},   //A
        {14, 0, 3, 0, 0, 0, 0 , 0},  //B
        {8, 3 , 0, 0, 0, 10, 0, 0},  //C
        {6, 0, 0, 0, 12, 0, 0, 0},   //D
        {5, 0, 0, 12, 0, 7, 9, 0},   //E
        {0, 0, 10, 0, 7, 0, 0, 15}, //F
        {0, 0, 0, 0, 9, 0, 0, 0},     //G
        {0, 0, 0, 0, 0, 15, 0, 0}    //H
    };

    primMST(graph);

    return 0;
}
```

Output:



```
Edge    Weight
2-1     3
0-2     8
0-3     6
0-4     5
4-5     7
4-6     9
5-7     15

Process returned 0 (0x0)   execution time : 0.109 s
Press any key to continue.
```

Conclusion:

Through the implementation of Prim’s Algorithm, we efficiently construct a Minimum Spanning Tree, optimizing connectivity in weighted graphs. The algorithm’s greedy approach ensures that the final tree has the smallest possible total edge weight, making it ideal for network design, computer clustering, and other applications requiring cost-effective connectivity. The experiment successfully demonstrates the effectiveness of Prim’s Algorithm in achieving an optimized network structure.

Task 02: A C++ program that will implement Kruskal's Algorithm.

Theory:

Kruskal's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a weighted, undirected graph. Unlike Prim’s Algorithm, which grows the MST vertex by vertex, Kruskal’s Algorithm builds the MST by considering edges in increasing order of weight, ensuring minimal cost. The algorithm utilizes Disjoint Set Union (DSU) to efficiently track connected components and prevent cycles.

Algorithm Steps

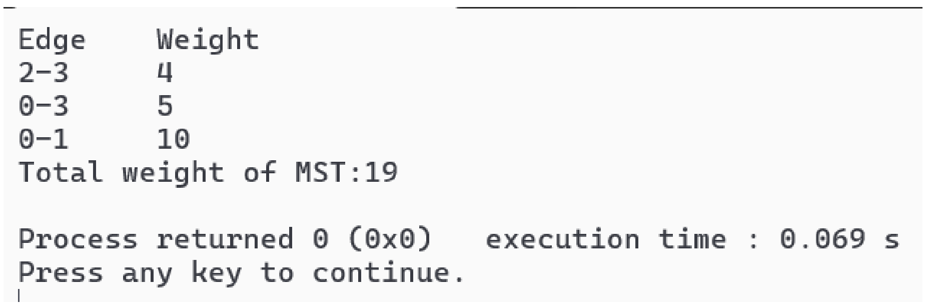
1. Sort all the edges of the graph in increasing order of their weight.
2. Initialize an empty MST and set each vertex as its own set using DSU.
3. Traverse through the sorted edge list:
    - If the edge connects two different sets (i.e., doesn’t form a cycle), include it in the MST and merge the sets.
4. Stop when the MST has (V – 1) edges.

By consistently selecting the smallest edge that maintains connectivity without forming cycles, Kruskal’s Algorithm ensures that the resulting spanning tree has the minimum possible total weight. This approach makes it particularly effective for network optimization problems.

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);
    vector<int> 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 << "\n";

    return 0;
}
```

Output:



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

Process returned 0 (0x0)   execution time : 0.069 s
Press any key to continue.
```

Conclusion:

The implementation of Kruskal's Algorithm successfully demonstrates its efficiency in constructing the Minimum Spanning Tree while ensuring cost-effective connectivity. By sorting edges and utilizing the Disjoint Set Union data structure, the algorithm maintains an optimal and structured network without redundant connections. Due to its simplicity and efficiency, Kruskal’s Algorithm finds practical applications in network design, clustering, and graph-based optimization problems.

Task 03: A C++ program that will implement Dijkstra's Algorithm.

Theory:

Dijkstra's Algorithm is a graph traversal algorithm designed to find the shortest path from a single source vertex to all other vertices in a weighted graph. The algorithm ensures optimal path selection by iteratively choosing the vertex with the minimum distance, making it ideal for applications in network routing, GPS navigation, and transportation planning.

Algorithm Steps

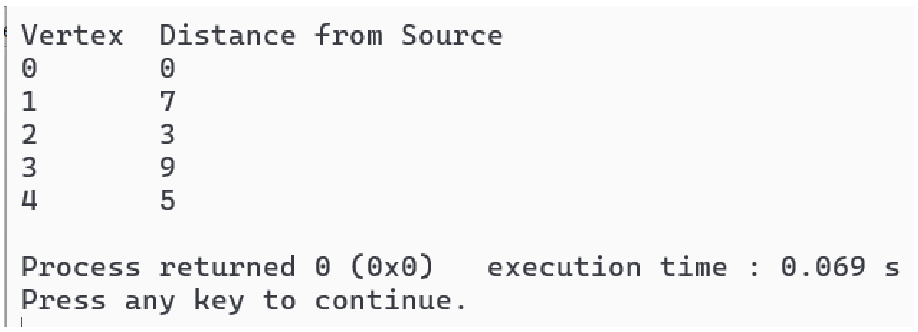
1. Initialize distances from the source to all vertices as infinite, except the source, which is set to 0.
2. Insert the source into a priority queue.
3. While the queue is not empty:
    - Extract the vertex with the minimum distance.
    - For each adjacent vertex, if the path through the current vertex is shorter, update the distance and insert it into the queue.
4. Repeat until all vertices are processed.

The algorithm effectively minimizes path costs using a greedy approach, ensuring that each vertex's shortest path is calculated efficiently.

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({2, 3});
    adj[1].push_back({3, 2});
    adj[1].push_back({2, 1});
    adj[2].push_back({1, 4});
    adj[2].push_back({3, 8});
    adj[2].push_back({4, 2});
    adj[3].push_back({4, 7});
    adj[4].push_back({3, 9});

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

Output:



```
Vertex  Distance from Source
0       0
1       7
2       3
3       9
4       5

Process returned 0 (0x0)   execution time : 0.069 s
Press any key to continue.
```

Conclusion:

The implementation of Dijkstra’s Algorithm successfully demonstrates its ability to determine the shortest path in a weighted graph. By utilizing a priority queue, the algorithm ensures that the paths are computed optimally while avoiding unnecessary computations. Given its efficiency, Dijkstra’s Algorithm is widely used in real-world scenarios, including network routing, urban transportation planning, and AI-based decision-making systems.

Task 04: A C++ program that will implement Bellman-Ford Algorithm.

Theory:

The Bellman-Ford Algorithm is used to find the shortest path from a single source to all vertices in a weighted graph. Unlike Dijkstra’s Algorithm, Bellman-Ford can handle negative weight edges, making it particularly useful for scenarios where negative weights exist. However, if a negative weight cycle is detected, the algorithm reports it as an issue since shortest paths become undefined.

Algorithm Steps

1. Initialize distances from the source to all vertices as infinite, except the source, which is set to zero.
2. Relax all edges for V-1 iterations:
    - For each edge (u, v, weight), if dist[u] + weight is smaller than dist[v], update dist[v].
3. Final iteration:

- If any distance can still be improved, a negative weight cycle exists, and the algorithm reports it.

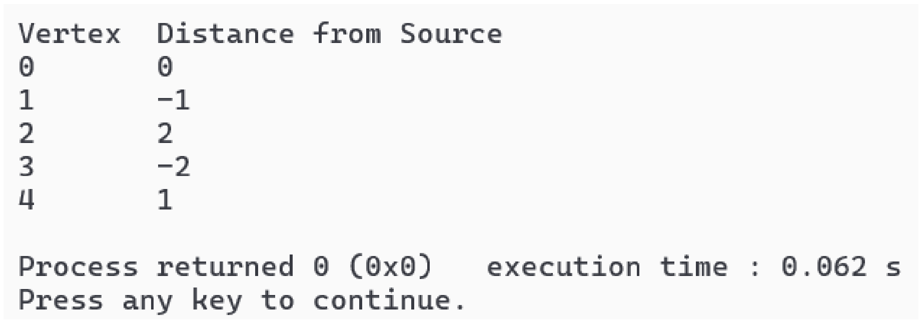
Bellman-Ford efficiently detects negative cycles, making it useful in applications such as currency exchange rate optimization, network routing, and decision-making systems involving cost analysis.

Code:

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

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

   for(int i = 0 ; i < V ; i++)
   {
      for(vector<int> edge : edges)
      {
         int u = edge[0];
```

```
            int v = edge[1];
            int wt = edge[2];

            if(dist[u] != INT_MAX && dist[u] + wt < dist[v])
            {
                if(i == V - 1)
                {
                    return {-1};
                }
                dist[v] = dist[u] + wt;
            }
        }
    }

    return dist;
}

int main()
{
    int V = 5;
    vector<vector<int>> edges = {
        {0 , 1 , -1},
        {0 , 2 , 4},
        {1 , 2 , 3},
        {1 , 3 , 2},
        {1 , 4 , 2},
        {3 , 1 , 1},
        {3 , 2 , 5},
        {4 , 3 , -3}
    };

    int src = 0;

    vector<int> ans = bellmanFord(V , edges , src);

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

    return 0;
}
```

Output:



```
Vertex  Distance from Source
0       0
1       -1
2       2
3       -2
4       1

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

Conclusion:

The implementation of Bellman-Ford Algorithm successfully demonstrates its ability to compute shortest paths in weighted graphs while handling negative weight edges. By systematically relaxing edges over multiple iterations, the algorithm ensures an accurate shortest-path calculation. Additionally, its capability to detect negative weight cycles makes it invaluable in scenarios where standard shortest-path algorithms like Dijkstra’s Algorithm may fail. This experiment showcases its practicality in network optimization, financial modeling, and distributed systems.