Bangladesh University of Business and Technology

Faculty of Engineering & Applied Sciences
Department of Computer Science and Engineering

Lab Assignment 07

CSE 242: Algorithms Lab

Submitted To:

Sumona Yeasmin
Lecturer
Department of Computer Science and Engineering

Submitted By

Md. Touhidur Rahman
ID: 22234103412
Riyadul Islam
ID: 22234103007
Md. Habibur Rahman
ID: 19201203032 (34 evening)

Kruskal's Algorithm:
Kruskal's Algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected graph. The minimum spanning tree is a subgraph that connects all the vertices without any cycles and with the minimum possible total edge weight. Kruskal's Algorithm achieves this by iteratively adding the shortest edge that does not form a cycle until all vertices are connected.

Our implementation of Kruskal's Algorithm:

Here is a brief overview of our implementation of Kruskal's Algorithm.

1. Disjoint Set Union (DSU) Data Structure :
- It's defined to manage sets of vertices efficiently and detect cycles in the graph.
- The `find` function implements path compression to find the root of a vertex in the disjoint-set forest.
- The `unite` function merges two sets in the disjoint-set forest while considering their ranks to maintain balanced trees.

2. Edge Struct:
- Represents an edge in the graph, containing source, destination, and cost.

3. Prims Function:
- Implements Kruskal's Algorithm to find the minimum spanning tree.
- Sorts the edges vector by cost.
- Iterates through the sorted edges:
  - Checks if adding the current edge to the MST would create a cycle.
  - If not, adds the edge to the MST and merges the sets of the vertices connected by the edge.
- Returns the MST.

4. Main Function:
- Reads user input for the number of cities and their names.
- Reads user input for the number of edges and their details (source city, destination city, and cost).
- Constructs the edges vector.
- Calls the Prims function to find the MST.
- Prints the solution using Kruskal's Algorithm, showing the selected edges and their costs, as well as the total cost of the MST.

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

// Disjoint Set Union Data Structure
// if find(a) == find(b) then a and b forms a cycle
// unite(a, b) will connect a and b
class DSU {
private:
    int* parent;
    int* rank;

public:
    DSU(int n) {
        parent = new int[n];
        rank = new int[n];

        for (int i = 0; i < n; i++) {
            parent[i] = -1;
            rank[i] = 1;
        }
    }

    // Find function
    int find(int i) {
        if (parent[i] == -1)
            return i;

        return parent[i] = find(parent[i]);
    }

    // Union function
    void unite(int x, int y) {
        int s1 = find(x);
        int s2 = find(y);

        if (s1 != s2) {
            if (rank[s1] < rank[s2]) {
                parent[s1] = s2;
            }
            else if (rank[s1] > rank[s2]) {
                parent[s2] = s1;
            }
            else {
                parent[s2] = s1;
                rank[s1] += 1;
            }
        }
    }
};
```

```cpp
struct Edge {
  int src, dest, cost;
};

vector<Edge> Kruskal(vector<Edge>& edges, int n) {
    DSU dsu(n);
    vector<Edge> mst;

    sort(edges.begin(), edges.end(), [](Edge a, Edge b) {
        return a.cost < b.cost;
    });

    for (Edge &e : edges) {
        if (dsu.find(e.src) != dsu.find(e.dest)) {
            mst.push_back(e);
            dsu.unite(e.src, e.dest);
        }
        if (mst.size() == n - 1) break;
    }

    return mst;
}


int main() {
    int cityCount, edgeCount;
    vector<Edge> edges;
    vector<string> cities;
    map<string, int> cityIndex;

    cout << "Enter number of cities: "; cin >> cityCount;
    cout << "Enter city names....\n";
    string cityName;
    for (int i = 1; i <= cityCount; ++i) {
        cout << '[' << i << "]: ";
        cin >> cityName;
        cities.push_back(cityName);
        cityIndex[cityName] = i - 1;
    } cout << endl;

    cout << "Enter number of edges: "; cin >> edgeCount;

    cout << "Enter edges....\n";
    cout << "Serial\tSrc\tDest\tCost\n";
    string src, dest; int cost;
    for (int i = 1; i <= edgeCount; ++i) {
        cout << '[' << i << "]:\t";
        cin >> src >> dest >> cost;
```

```
            Edge e;
            e.cost = cost;
            e.src = cityIndex[src];
            e.dest = cityIndex[dest];
            edges.push_back(e);
        } cout << endl;

        vector<Edge> mst = Kruskal(edges, cityCount);

        cout << "Solution using Krushkal's Algorithm:\n";
        int totalCost = 0;
        for (Edge &e : mst) {
            cout << cities[e.src] << " -> " << cities[e.dest]
                 << " (cost " << e.cost << ")\n";
            totalCost += e.cost;
        } cout << endl;

        cout << "Total cost: " << totalCost << endl;
        return 0;
    }
```

Input / Output:

```
Enter number of cities: 5
Enter city names....
[1]: Dhaka
[2]: Bhola
[3]: Khulna
[4]: Sylhet
[5]: Pabna

Enter number of edges: 7
Enter edges....
Serial  Src     Dest    Cost
[1]:    Dhaka   Bhola   2
[2]:    Dhaka   Sylhet  6
[3]:    Bhola   Khulna  3
[4]:    Sylhet  Bhola   5
[5]:    Pabna   Bhola   8
[6]:    Khulna  Pabna   7
[7]:    Pabna   Sylhet  9

Solution using Krushkal's Algorithm:
Dhaka -> Bhola (cost 2)
Bhola -> Khulna (cost 3)
Sylhet -> Bhola (cost 5)
Khulna -> Pabna (cost 7)

Total cost: 17
```

Prim's Algorithm:

Prim's Algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected graph. The algorithm starts from an arbitrary vertex and repeatedly grows the MST by adding the shortest edge that connects a vertex in the MST to a vertex outside the MST. This process continues until all vertices are included in the MST, resulting in a tree that spans all vertices with the minimum possible total edge weight.

Our implementation of Prim's Algorithm:

1. Edge Struct:

- Defines a structure to represent an edge in the graph.
- Contains two fields: `w` for the weight of the edge and `to` for the index of the destination vertex.

2. Prims Function:

- Implements Prim's Algorithm to find the minimum spanning tree.
- Initializes a vector `selected` to keep track of selected vertices.
- Initializes a vector `min_e` to store the minimum edge weight to connect each vertex to the MST.
- Iterates through each vertex:
  - Selects the vertex with the minimum edge weight to add to the MST.
  - Adds the selected vertex to the MST and updates the `selected` vector.
  - Updates the `min_e` vector with the minimum edge weight to connect each remaining vertex to the MST.
- Returns the MST as a vector of vectors, where each inner vector represents an edge (source vertex, destination vertex, and edge weight).

3. Main Function:

- Reads user input for the number of cities.
- Constructs an adjacency matrix to represent the graph.
- Reads user input for city names and edge details (source city, destination city, and cost).
- Constructs the adjacency matrix based on the input.
- Calls the `Prims` function to find the MST.
- Prints the solution using Prim's Algorithm, showing the selected edges and their costs, as well as the total cost of the MST.

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

const int INF = 1000000000; // weight INF means there is no edge

struct Edge {
    int w = INF, to = -1;
};

vector<vector<int>> Prims(vector<vector<int>>& adj, int n) {
  vector<vector<int>> mst;

  vector<bool> selected(n, false);
  vector<Edge> min_e(n);
  min_e[0].w = 0;

  for (int i=0; i<n; ++i) {
    int v = -1;
    for (int j = 0; j < n; ++j) {
      if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
        v = j;
    }

    if (min_e[v].w == INF) {
      cout << "No MST!" << endl;
      break;
    }

    selected[v] = true;
    if (min_e[v].to != -1) {
      mst.push_back({v, min_e[v].to, min_e[v].w});
    }

    for (int to = 0; to < n; ++to) {
      if (adj[v][to] < min_e[to].w)
        min_e[to] = {adj[v][to], v};
    }
  }

  return mst;
}
```

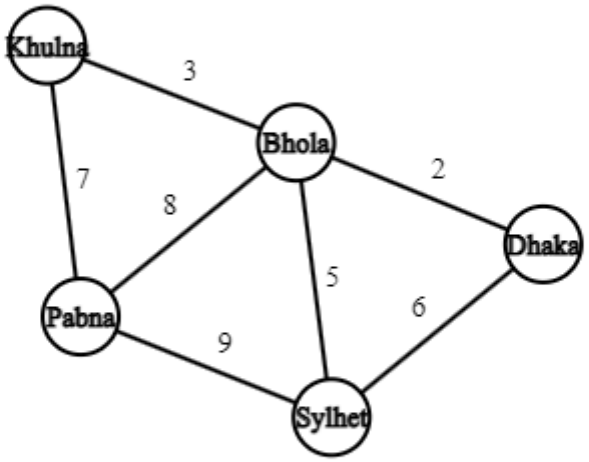
```cpp
int main() {
    int cityCount, edgeCount;
    vector<string> cities;
    map<string, int> cityIndex;

    cout << "Enter number of cities: "; cin >> cityCount;

    // adjacency matrix
    vector<vector<int>> adj(cityCount);
    for (int i = 0; i < cityCount; ++i) {
        vector<int> v(cityCount, INF);
        adj[i] = v;
    }

    cout << "Enter city names....\n";
    string cityName;
    for (int i = 1; i <= cityCount; ++i) {
        cout << '[' << i << "]: ";
        cin >> cityName;
        cities.push_back(cityName);
        cityIndex[cityName] = i - 1;
    } cout << endl;

    cout << "Enter number of edges: "; cin >> edgeCount;
    cout << "Enter edges....\n";
    cout << "Serial\tSrc\tDest\tCost\n";
    string src, dest;
    int srcIdx, destIdx, cost;
    for (int i = 1; i <= edgeCount; ++i) {
        cout << '[' << i << "]:\t";
        cin >> src >> dest >> cost;

        srcIdx = cityIndex[src];
        destIdx = cityIndex[dest];

        // add edge to adjacency list
        adj[srcIdx][destIdx] = cost;
        adj[destIdx][srcIdx] = cost;
    } cout << endl;

    auto mst = Prims(adj, cityCount);

    cout << "Solution using Prims's Algorithm:\n";
    int totalCost = 0;
```

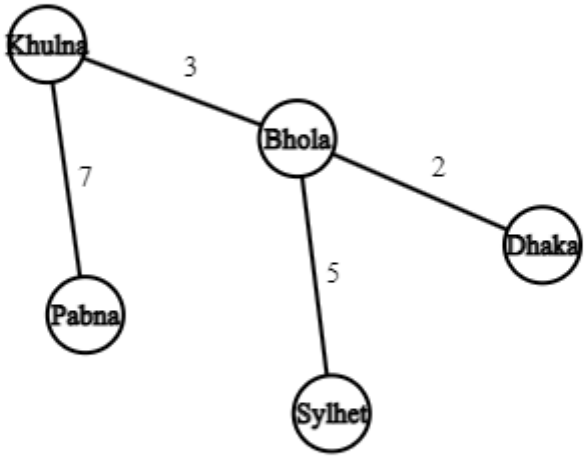
```
    for (auto &edge : mst) {
        int src = edge[0], dest = edge[1], cost = edge[2];
        cout << cities[src] << " -> " << cities[dest]
             << " (cost " << cost << ")\n";
        totalCost += cost;
    } cout << endl;

    cout << "Total cost: " << totalCost << endl;
    return 0;
}
```

Input / Output:

```
Enter number of cities: 5
Enter city names....
[1]: Dhaka
[2]: Bhola
[3]: Khulna
[4]: Sylhet
[5]: Pabna

Enter number of edges: 7
Enter edges....
Serial  Src     Dest    Cost
[1]:    Dhaka   Bhola   2
[2]:    Dhaka   Sylhet  6
[3]:    Bhola   Khulna  3
[4]:    Sylhet  Bhola   5
[5]:    Pabna   Bhola   8
[6]:    Khulna  Pabna   7
[7]:    Pabna   Sylhet  9

Solution using Prims's Algorithm:
Bhola -> Dhaka (cost 2)
Khulna -> Bhola (cost 3)
Sylhet -> Bhola (cost 5)
Pabna -> Khulna (cost 7)

Total cost: 17
```

Visualization:





The rationale behind choosing Prim's or Kruskal's algorithm for this problem :

Based on our problems, we can consider the characteristics of the graphs and the efficiency of the algorithms to determine which algorithm might be more suitable:

1. Kruskal's Algorithm:

Characteristics: Kruskal's Algorithm is efficient for graphs with relatively few edges, especially if the graph is sparse.

Rationale for Selection: If the input graph for the problem described in your code tends to have relatively few connections between vertices (i.e., a sparse graph), Kruskal's Algorithm might be a good choice. Since Kruskal's Algorithm sorts the edges and performs union-find operations, it can efficiently find the minimum spanning tree in such cases.

2. Prim's Algorithm:

Characteristics: Prim's Algorithm is efficient for graphs with many vertices and edges, especially if the graph is dense.

Rationale for Selection: If the input graph for the problem tends to have many connections between vertices (i.e., a dense graph), Prim's Algorithm might be a better choice. Prim's Algorithm's time complexity depends on the number of vertices and edges, making it suitable for dense graphs where Kruskal's Algorithm might be less efficient due to the overhead of sorting edges.

We can choose between Prim's and Kruskal's Algorithm accordingly:

If the graph tends to be sparse, with relatively few connections between vertices, then Kruskal's Algorithm is the best choice due to its good efficiency.

If the graph tends to be dense, with many connections between vertices, then Prim's Algorithm is the best option due to its good efficiency.

Time complexity analysis of both algorithms to understand their computational efficiency :

1. Kruskal's Algorithm:

- Sorting the edges: $O(E\ log\ E)$, where E is the number of edges.
- Union-find operations: $O(E\ \alpha(V))$, where $\alpha(V)$ is the inverse Ackermann function, which grows very slowly.
- Total time complexity: $O(E\ log\ E\ +\ E\ *\ \alpha(V))$

Kruskal's Algorithm is efficient for sparse graphs (graphs with relatively few edges), as the time complexity is dominated by the sorting step. In dense graphs (graphs with many edges), the union-find operations become more significant.

2. Prim's Algorithm:

- Initializing min_e vector: $O(V)$, where V is the number of vertices.
- Finding the minimum edge in each iteration: $O(v^2)$ if using an adjacency matrix or $O(v\ log(v))$ if using a priority queue.

  Total time complexity: $O(v^2)$ or $O(VlogV\ +\ ElogV)\ =\ O(ElogV)$

Prim's Algorithm is efficient for dense graphs (graphs with many vertices and edges), especially when using a priority queue to find the minimum edge efficiently. However, in sparse graphs, Kruskal's Algorithm may be more efficient due to its lower time complexity.

Both algorithms have their strengths depending on the characteristics of the input graph.

Kruskal's Algorithm is typically preferred for sparse graphs, while Prim's Algorithm is suitable for dense graphs