Lab Manual

Course: Computer Algorithms Sessional
Department of CSE, AMUST

Task 1: Prim's Algorithm

1. Objective

To understand and implement Prim's Algorithm for finding the Minimum Spanning Tree (MST) of a connected, weighted, undirected graph using C++.

2. Theory

Prim's Algorithm is a **greedy algorithm** that finds the MST by starting from a single vertex and growing the MST one vertex at a time.
At each step, it adds the **minimum weight edge** that connects a vertex in the MST to a vertex outside it.

Key Properties:

- Applicable only for **connected**, **undirected**, and **weighted** graphs.
- Ensures no cycles are formed.
- MST has **(V – 1)** edges for a graph with **V** vertices.

3. Algorithm Steps

1. Initialize a key[] array and set all values to ∞. Set the key value of the starting vertex to 0.
2. Use a mstSet[] to track vertices included in the MST (initially all 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 of v as u.

.

4. Sample Code in C++

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

#define V 5  // 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, 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;
}
```

5. Sample Input/Output

Input: Adjacency Matrix for 5 nodes.

Output:

```
Edge   Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

6. Observation Table

Students should test different sets of input and record the results.

7. Viva Questions

- What is a Minimum Spanning Tree?
- How does Prim's Algorithm work?
- How does it differ from Kruskal's Algorithm?
- Can Prim's Algorithm be applied to directed graphs?
- What is the time complexity of Prim's Algorithm using an adjacency matrix?

Task 2: Kruskal's Algorithm

1. Objective

To understand and implement Kruskal's Algorithm for finding the Minimum Spanning Tree (MST) of a connected, weighted, undirected graph using C++.

2. Theory

Kruskal's Algorithm is a greedy algorithm that finds the MST by sorting all edges in non-decreasing order of weight and adding them one by one, avoiding cycles.

Key Properties:

- Works for connected, undirected, and weighted graphs.

- Uses Disjoint Set Union (DSU) to detect cycles.

- Builds MST edge by edge.

- The MST has exactly (V – 1) edges.

3. 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:
   a. 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.

4. Sample Code in C++

```
#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 << endl;

  return 0;
}
```

5. Sample Input/Output

Input: 5 edges among 4 nodes with respective weights.

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

6. Observation Table

Record the MST for different input graphs.

7. Viva Questions

- What is the key idea behind Kruskal's Algorithm?
- What is the purpose of using Disjoint Set Union (DSU)?
- What is the time complexity of Kruskal's Algorithm?
- How does Kruskal's compare with Prim's?
- Can Kruskal's Algorithm handle disconnected graphs?

Task 3: Dijkstra's Algorithm

1. Objective

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

2. Theory

Dijkstra's Algorithm is a greedy algorithm used to find the shortest path from a single source node to all other nodes in a weighted graph with non-negative edge weights.

Key Properties:

- Works with directed and undirected graphs.

- Does not work with negative edge weights.

- Uses a priority queue or min-heap to optimize performance.

- Time complexity: O((V + E) log V) with a min-heap.

3. 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:
   a. Extract the vertex with the minimum distance.
   b. 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.

4. Sample Code in C++

```
#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;
```

}

5. Sample Input/Output

Input: A graph with 5 vertices and weighted edges from source vertex 0.

Output:

Vertex	Distance from Source
0	0
1	8
2	9
3	7
4	5

6. Observation Table

Record shortest paths from the source for different graphs.

7. Viva Questions

- What is the purpose of Dijkstra's Algorithm?
- Why doesn't Dijkstra's Algorithm work with negative weights?
- What data structure is commonly used to implement Dijkstra efficiently?
- Compare Dijkstra's and Bellman-Ford algorithms.
- Can Dijkstra's Algorithm be used for directed graphs?