NED UNIVERSITY OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & IT

BSCS (with Specialization in Cyber Security)

CT-636 Design and Analysis of Algorithms

Sir Usman Amjad

Minimum Cost Spanning Tree

MUNEEZA BADAR	CR-22022
MUHAIB SHAMSHER	CR-22029
SYED GUFRAN RAZA	CR-22028

Introduction

A **spanning tree** is a critical concept in graph theory, representing a subset of a connected, undirected graph that includes all vertices with the fewest edges and no cycles. Among possible spanning trees, a **Minimum Spanning Tree (MST)** has the smallest total edge weight, making it ideal for optimizing connection costs in network designs. To find MSTs efficiently, two primary algorithms are widely used: **Prim's Algorithm** and **Kruskal's Algorithm**.

1. **Prim's Algorithm**
   Prim's algorithm starts from a chosen vertex and expands the MST by repeatedly adding the shortest edge connected to the current tree, which is particularly efficient for dense graphs.
2. **Kruskal's Algorithm**
   Kruskal's algorithm sorts all edges by weight and adds them one by one, preventing cycles and making it well-suited for sparse graphs. Both algorithms offer valuable methods for constructing MSTs depending on the structure of the graph in question.

Understanding Algorithms

Prim's Algorithm

```
selectMinVertex(value[], setMST[], V)
    minimum = INT_MAX and vertex = -1

    for i=0 to V
        if setMST[i] == false and value[i] < minimum
            minimum = value[i]
            vertex = i

    return vertex

primsMST(graph[][], int V)
    Initialize parent[] = -1, value[] = INT_MAX, setMST[] = false

    set value[0] = 0

    for i=0 to V-1
        vertex U = selectMinVertex(value, setMST, V)
        setMST[U] = true

        for j=0 to V
            if graph[U][j] != 0 and setMST[j] == false and graph[U][j] < value[j]
                value[j] = graph[U][j]
                parent[j] = U

    printMST()
```

Kruskal’s Algorithm

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

class DSU
    Initialize parent[], rank[]

    DSU(V)
        Resize parent[] with V elements where each element is -1
        Resize rank[] with V elements where each element is 0

    find(int u)
        if parent[u] == -1
            return u

        return parent[u] = find(parent[u])

    unite(u, v)
        rootU = find(u)
        rootV = find(v)

        if rank[rootU] < rank[rootV]
            parent[rootU] = rootV
        else if rank[rootU] > rank[rootV]
            parent[rootV] = rootU
        else
            parent[rootV] = rootU
            rank[rootU]++

kruskalMST(edges[], V)
    DSU dsu(V)
    Initialize mst[]

    Sort edges[] by weight in ascending order.

    for each edge in edges
        dsu.find(edge.u) != dsu.find(edge.v)
            dsu.unite(edge.u, edge.v)
            Add edge to mst

    printMST mst
```

Prim’s Complexity

```
selectMinVertex(value[], setMST[], V)
    minimum = INT_MAX and vertex = -1

    for i=0 to V
        if setMST[i] == false and value[i] < minimum
            minimum = value[i]
            vertex = i

    return vertex

primsMST(graph[][], int V)
    Initialize parent[] = -1, value[] = INT_MAX, setMST[] = false

    set value[0] = 0

    for i=0 to V-1
        vertex U = selectMinVertex(value, setMST, V) }  O(V)
        setMST[U] = true

        for j=0 to V
            if graph[U][j] != 0 and setMST[j] == false and graph[U][j] < value[j] }  O(V)
                value[j] = graph[U][j]
                parent[j] = U

    printMST()
```

The code iterates over all vertices and all edges connecting them, regardless of edge weights. In the best and worst cases:

1. **Selecting Minimum Vertex**: This takes O(V) in each of the V−1 iterations, contributing O(V^2) in total.
2. **Updating Adjacent Vertices**: The inner loop also takes O(V) in each iteration, contributing another O(V^2).

So, the total time complexity in the general case is:

O(V^2)

- Worst case: O(V^2)
- Best case: O(V^2)

Kruskal's Complexity

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

class DSU
    Initialize parent[], rank[]

    DSU(V)
        Resize parent[] with V elements where each element is -1
        Resize rank[] with V elements where each element is 0

    find(u)
        if parent[u] == -1
            return u

        return parent[u] = find(parent[u])

    unite(u, v)
        rootU = find(u)
        rootV = find(v)

        if rank[rootU] < rank[rootV]
            parent[rootU] = rootV
        else if rank[rootU] > rank[rootV]
            parent[rootV] = rootU
        else
            parent[rootV] = rootU
            rank[rootU]++

kruskalMST(edges[], V)
    DSU dsu(V)
    Initialize mst[]

    Sort edges[] by weight in ascending order.  }  O(ElogE)

    for each edge in edges                    ⎫
        dsu.find(edge.u) != dsu.find(edge.v)  ⎬   O(ElogV)
            dsu.unite(edge.u, edge.v)         ⎪
            Add edge to mst                   ⎭

    printMST mst
```

Kruskal's algorithm efficiently finds the Minimum Spanning Tree (MST) by sorting edges and using a Disjoint Set Union (DSU) structure to avoid cycles.

1. **Sorting the Edges:** Sorting (E) edges takes O(E log E).
2. **Union by Rank and Path Compression**: Checking each edge to see if it should be added to the MST requires DSU operations. These operations (find and union) have an amortized complexity of O(log V) per operation. For all (E) edges, this is O(E log V).
3. **Total Time Complexity:**

$$T(n) = O(E \log E) + O(E \log V)$$

4. **Simplification for Dense Graphs**: For dense graphs (like complete graphs), we can assume log E approx log V, so O(E log E) simplifies to O(E log V).

So, The time complexity of Kruskal's algorithm can be represented as either O(E log V) or O(E log E).

- Worst case: O(E log E)
- Best case: O(E log E)

Comparison

S.No.	Prim's Algorithm	Kruskal's Algorithm
1	This algorithm begins to construct the shortest spanning tree from any vertex in the graph.	This algorithm begins to construct the shortest spanning tree from the vertex having the lowest weight in the graph.
2	To obtain the minimum distance, it traverses one node more than one time.	It crosses one node only one time.
3	The time complexity of Prim's algorithm is O(V^2).	The time complexity of Kruskal's algorithm is O(E log V).
4	In Prim's algorithm, all the graph elements must be connected.	Kruskal's algorithm may have disconnected graphs.
5	When it comes to dense graphs, the Prim's algorithm runs faster.	When it comes to sparse graphs, Kruskal's algorithm runs faster.
6	It prefers list data structure.	It prefers the heap data structure.

Strengths and Weaknesses of Prim’s & Kruskal’s Algorithms

➢ Prim’s Algorithm Strengths:

1. Efficient on Dense Graphs:

Prim’s algorithm is efficient on dense graphs (those with many edges) due to its (O(V^2)) complexity with adjacency matrix implementations.

It considers one vertex at a time, adding the shortest adjacent edge, making it ideal when you can easily access the list of adjacent nodes, such as in network design problems.

2. Simple Implementation:

Prim's algorithm is relatively straightforward to implement, especially in adjacency matrix-based graphs, where you only need to find the minimum-cost edges for connected nodes.

3. Connected Graph Requirement:

Prim’s algorithm naturally assumes a single, connected component, making it suitable for scenarios where the entire network is guaranteed to be connected, such as connecting cities in a transportation or telecommunications network.

Weaknesses:

1. Less Efficient on Sparse Graphs:

In sparse graphs (those with fewer edges), Prim's (O (V^2)) complexity can be costly since it must check all edges regardless of the number of connections. This may lead to inefficiencies in terms of memory and processing.

2. Not Suitable for Edge List Representation:

Prim’s algorithm is inefficient with edge list representations, where finding adjacent nodes would require additional data structures, making it less adaptable to dynamically changing or sparse graphs.

3. Requires Starting Vertex:

Prim’s algorithm requires a starting point, which may not be intuitive in scenarios with multiple disconnected components.

➢ Kruskal’s Algorithm Strengths:

1. Efficient on Sparse Graphs:

Kruskal’s algorithm is efficient for sparse graphs, as its complexity of \(O(E \log E)\) focuses on the number of edges. This makes it ideal for networks or graphs with fewer connections, where sorting edges is faster.

2. Edge List Friendly:

Since it works by sorting edges, Kruskal’s algorithm is well-suited for edge list representation and for graphs where edges are dynamically updated or need to be sorted by weight (cost).

3. Finds MST for Each Component:

Kruskal’s algorithm doesn’t require a connected graph; it can work on multiple disconnected components and create a minimum spanning tree for each, making it versatile in scenarios where graphs may have isolated clusters.

Weaknesses:

1. Sorting Overhead for Dense Graphs:

In dense graphs, Kruskal's algorithm requires sorting a large number of edges, which can be computationally expensive, making it slower compared to Prim's algorithm.

2. Requires Union-Find Structure:

To efficiently check for cycles, Kruskal's algorithm needs a union-find data structure, which can be complex to implement and might require additional memory and computation.

3. Less Efficient with Adjacency Matrix:

Kruskal's algorithm doesn't benefit much from adjacency matrices, which store all edge information. Instead, it works better with edge lists, making it less compatible with matrix-based representations.

Applications

Prim's Algorithm Applications:

1. Network Design:

Used in designing network connections such as telecommunications, computer networks, and electrical grid systems to connect all nodes with minimal wiring or cabling costs.

Prim's algorithm helps find the minimum-cost way to connect all points without loops.

2. Transportation and Logistics:

In designing road networks where cities or points need to be connected with the minimum distance or cost, ensuring all locations are reachable with the least expenditure.

3. Cluster Analysis:

Used in data clustering, specifically when constructing minimum spanning trees for data points. Prim's algorithm helps find clusters by identifying connections that minimize distances.

4. Infrastructure Development:

Applied in pipeline and railroad construction where all stations or endpoints need to be connected with the least amount of resources while avoiding unnecessary paths

Kruskal's Algorithm Applications:

1. Network Design:

Similar to Prim's, it is also used in designing network systems like telephone networks and computer networks but is especially beneficial when the graph has a smaller number of edges.

2. Social Networking Sites:

Applied to analyze and create clusters in social networks, where users are represented as nodes, and their relationships as edges. Kruskal's algorithm can help find clusters of closely connected users by determining a minimum spanning tree.

3. mage Segmentation:

Used in computer vision and image processing for image segmentation. By treating pixels or regions as nodes and connecting them based on similarity, Kruskal's algorithm can help minimize the total difference and segment the image effectively.

4. Electric Grid Layouts:

In the planning of electric power grids, Kruskal's algorithm helps to minimize the cost of power lines needed to connect substations while ensuring no closed loops are formed.

5. Geographic Information Systems (GIS):

Useful in geographic mapping and in creating MSTs that represent minimal distances between cities or geographic points, assisting in route planning and map layout.

IMPLEMENTATION OF MST

"AquaConnect: Efficient Water Distribution with MST Algorithms"

Project Statement

Optimal Water Pipeline Network Design for a Residential Society Using Minimum Spanning Tree Algorithms.

Project Description

In this project, we aim to design a water pipeline network for a residential society, represented as a collection of houses on a 2D plane. Each house is a unique point on this plane, defined by its coordinates. The goal is to construct a water pipeline system that connects all the houses with the minimum possible cost. The cost to lay a pipeline between two houses is based on the Manhattan distance between them, calculated as:

$$\text{Cost} = |x1 - x2| + |y1 - y2|$$

To achieve this, we need to determine the Minimum Spanning Tree (MST) for the network, which will connect all houses with the least total pipeline cost, ensuring that the water reaches every house without any redundant connections.

Methodology

We will use **two MST algorithms**—**Prim's Algorithm** and **Kruskal's Algorithm**—to find the optimal pipeline layout. Each algorithm has its unique approach and efficiency characteristics:

1. **Prim's Algorithm**: This algorithm constructs the MST by starting from an arbitrary node and expanding the tree by repeatedly adding the nearest vertex not yet in the tree. It is generally well-suited for dense graphs.
2. **Kruskal's Algorithm**: This algorithm builds the MST by sorting all edges by their weights and adding edges one by one to the MST, ensuring no cycles form. Kruskal's algorithm is efficient for sparse graphs and uses a Disjoint Set Union (DSU) structure for cycle detection.

Objectives

1. **Calculate MST Using Prim's and Kruskal's Algorithms**:
   - ✓ Implement both algorithms to find the MST of the residential society's graph.
2. **Compare Efficiency and Cost**:
   - ✓ Measure and compare the total cost of the pipeline network generated by each algorithm.
   - ✓ Compare the execution times of both algorithms to determine which performs better for this problem scenario.

3. **Select Optimal Solution**:

   - ✓ Based on the comparison, select the MST with the minimum cost and better efficiency to be the final solution for the pipeline layout.

Expected Outcomes

The project will output the minimum cost to connect all houses in the society, using the MST with the least pipeline cost. Additionally, we will analyze which algorithm (Prim's or Kruskal's) provides a more efficient solution based on cost and performance, guiding the choice of the best algorithm for similar future applications.

Real-World Application

This project simulates a practical scenario where infrastructure needs to be laid in an optimized manner. By reducing the total pipeline cost, we minimize both construction expenses and resource consumption, leading to a cost-effective and sustainable solution for infrastructure planning in residential areas.

Project Contributions

SYED GUFRAN RAZA (CR-22028)

- ✓ Applications.
- ✓ MST Implementation.
- ✓ Algorithms Analysis.
- ✓ Algorithms Strength & Weaknesses.
- ✓ Presentation Slides.

MUHAIB SHAMSHER (CR-22029)

- ✓ Project Idea.
- ✓ MST Implementation.
- ✓ Web Assembly.
- ✓ Algorithms.
- ✓ Algorithms Strength & Weaknesses.

MUNEEZA BADAR (CR-22022)

- ✓ Algorithms Complexities.
- ✓ Algorithms Comparisons.
- ✓ MST Implementation.
- ✓ Project report.
- ✓ MST Implementation report.