ROAD NETWORK DESIGN USING MINIMUM SPANNING TREE

A CAPSTONE PROJECT REPORT

Submitted in partial fulfilment of the requirement for the award of the Degree of

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

By

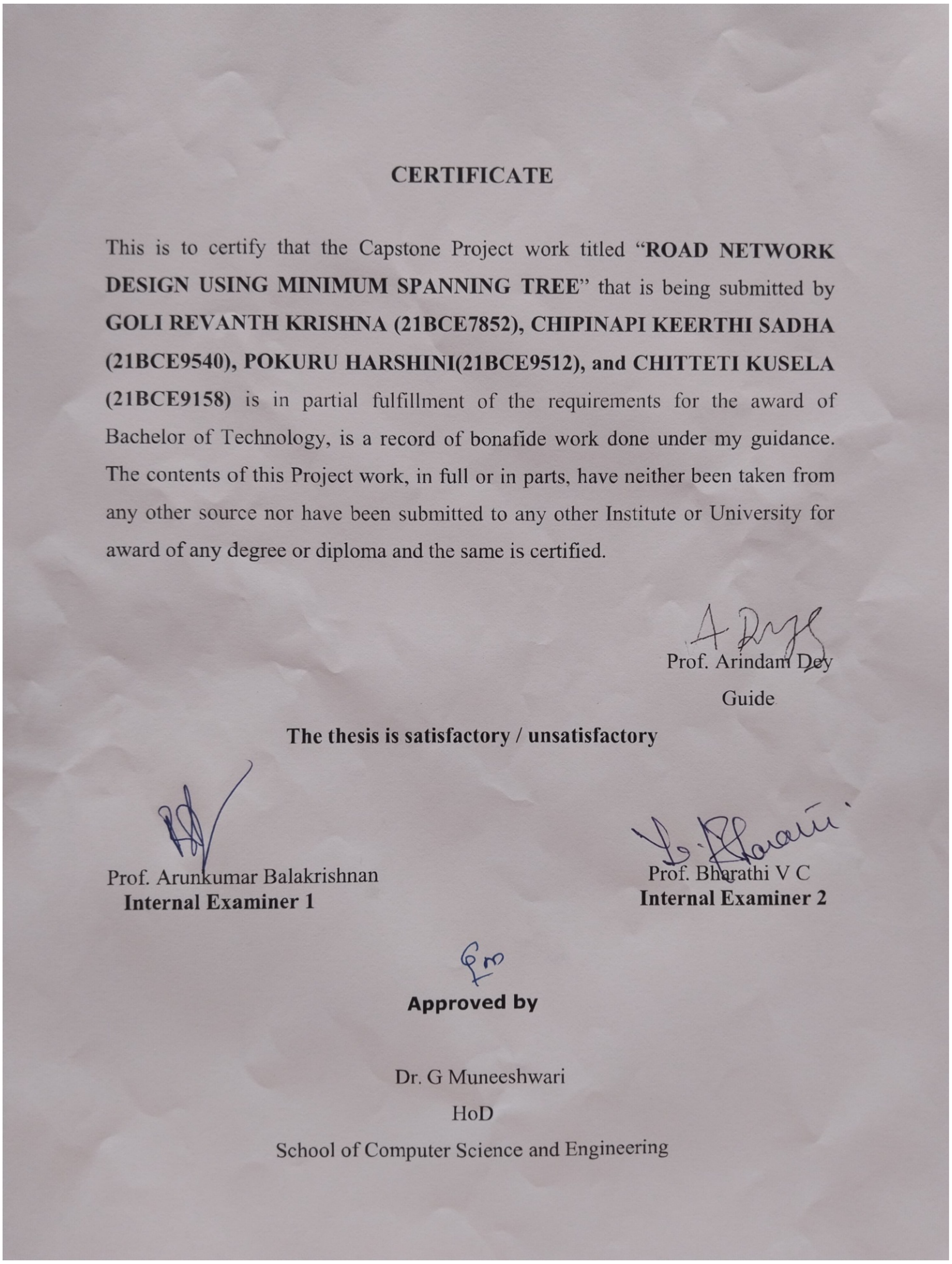
Goli Revanth Krishna – 21BCE7852
Chipinapi KeerthiSadha – 21BCE9540
Chitteti Kusela – 21BCE9158
Pokuru Harshini – 21BCE9512

Under the Guidance of

Prof. Arindam Dey

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
VIT-AP UNIVERSITY
AMARAVATI – 522237
ANDHRA PRADESH, INDIA

December 2024



CERTIFICATE

This is to certify that the Capstone Project work titled "**ROAD NETWORK DESIGN USING MINIMUM SPANNING TREE**" that is being submitted by **GOLI REVANTH KRISHNA (21BCE7852), CHIPINAPI KEERTHI SADHA (21BCE9540), POKURU HARSHINI(21BCE9512), and CHITTETI KUSELA (21BCE9158)** is in partial fulfillment of the requirements for the award of Bachelor of Technology, is a record of bonafide work done under my guidance. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted to any other Institute or University for award of any degree or diploma and the same is certified.

[illegible]

Prof. Arindam Dey

Guide

The thesis is satisfactory / unsatisfactory

[illegible]

Prof. Arunkumar Balakrishnan
Internal Examiner 1

[illegible]

Prof. Bharathi V C
Internal Examiner 2

[illegible]

Approved by

Dr. G Muneeshwari

HoD

School of Computer Science and Engineering

ACKNOWLEDGEMENTS

We thank VIT-AP University for giving us the opportunity to work on this project. We would like to express our deep and sincere gratitude to Prof. Arindam Dey**,** School of Computer Science and Engineering, for his guidance, providing us with relevant references, valuable support, motivation, and invaluable inputs throughout the process.

We express our heartfelt gratitude to Prof. Saroj Kumar Panigrahy sir, Associate Dean, Program Chair and Professor, School of Computer Science and Engineering. It is our pleasure to express our heartfelt gratitude to Dr. S Sudhakar Ilango Sir, Dean, School of Computer Science and Engineering, for providing an environment to work in for their support and guidance during the course's tenure.

It is a pleasure to thank all faculty members who work as limbs of our university for their non-self-centred enthusiasm and timely encouragement showered on us with zeal, prompting the acquisition of the necessary knowledge to successfully complete our course study. We would like to express our gratitude to our respective parents for their assistance.

Finally, we extend our heartfelt gratitude to our parents for their unwavering support and encouragement and to everyone who contributed directly or indirectly to the successful completion of this project.

ABSTRACT

The Minimum Spanning Tree (MST) is a foundational concept in graph theory, widely used in various domains such as network design, clustering, and data analysis. This project explores the practical construction of MSTs using Kruskal's and Prim's algorithms, with a specific focus on optimizing city connections. A user-friendly web application built with Flask provides an interactive platform where users can input city names, compute MSTs dynamically, and visualize the results. Geopy is utilized for geocoding city coordinates, and the great-circle formula is applied to calculate edge weights, ensuring accurate distance-based graph modeling. For visualization and graph representation, Python libraries like NetworkX and Matplotlib are integrated to offer a seamless user experience.

The project was tested on a dataset of 13 cities, resulting in an MST with a total length of 4,913 kilometers. The visualization emphasizes the efficiency of the computed MST by clearly highlighting its edges, enhancing the user's understanding of how the cities are optimally connected. This practical demonstration of Kruskal's and Prim's algorithms reveals their effectiveness in solving real-world problems while providing insights into efficient network design. The integration of dynamic, interactive tools within the application ensures the results are not only accurate but also visually intuitive for diverse audiences.

In addition to showcasing the algorithms' computational strengths, the project highlights the significance of accessible and interactive platforms for graph-based analysis. By merging geospatial data with real-time computations, the system creates a valuable resource for understanding and solving connectivity issues. The visualizations make complex network structures easier to interpret, aiding in decision-making processes in fields such as urban planning and logistics. This combination of technical precision and user-focused design bridges the gap between theoretical knowledge and practical application.

Ultimately, this project illustrates the broader potential of MST algorithms beyond academic use. The incorporation of modern web technologies, real-time graph processing, and visualization tools transform traditional computations into an engaging, hands-on experience. By focusing on usability and clarity, the project provides a compelling example of how theoretical models can address practical challenges, paving the way for future advancements in network optimization and analysis.

TABLE OF CONTENTS

List of figures

CHAPTER 1

INTRODUCTION

The Minimum Spanning Tree (MST) is a fundamental concept in graph theory with significant real-world applications, particularly in network design, clustering, and data analysis. It focuses on connecting all nodes in a graph with the minimum total edge weight, optimizing resource allocation and reducing costs. This project investigates MST construction using Kruskal's and Prim's algorithms, two efficient methods with unique approaches to solving this problem. A Flask-based web application was developed to provide users with an interactive platform for dynamically computing and visualizing MSTs by entering city names. The application leverages Geopy for geocoding city coordinates and calculating distances through the great-circle formula, ensuring accuracy in edge weight computation. NetworkX and Matplotlib are utilized for graph modeling and visualization, offering intuitive representations of MST structures. Tested on 13 cities, the system produced an MST with a total weight of 4,913 kilometers, efficiently connecting the cities and demonstrating the algorithms' practicality. This project is particularly relevant for real-life applications such as road network design, where MSTs ensure cost-effective and optimized routes between cities, reducing construction and maintenance expenses while improving connectivity. The interactive visualization tools enhance user understanding and illustrate how these algorithms can address real-world challenges. By bridging theoretical graph concepts with practical applications, this project showcases the importance of MSTs in solving complex connectivity problems in an accessible and engaging way.

1.1 Objectives

The primary objective of this study is to analyze and compare two widely used algorithms, Prim's and Kruskal's, for solving the Minimum Spanning Tree problem in the context of network optimization. By examining these algorithms, the project aims to evaluate their computational efficiency, practicality, and suitability for real-world applications such as transportation network design, urban planning, and other optimization challenges. This comprehensive comparison will provide a deeper understanding of their strengths, limitations, and performance across different types of graph structures.

The system developed in this project is designed to identify and input the locations of clients or neighborhoods, which are represented as nodes in a graph. Using Prim’s and Kruskal’s algorithms, it computes the optimal network structure that connects all nodes while minimizing the total edge weight. The output of the system includes the edges of the MST along with their respective weights, clearly showcasing the least-cost network structure and its benefits in practical scenarios.

Additionally, the project aims to enhance understanding by visualizing the MST through graphical tools like NetworkX and Matplotlib, which highlight the optimal paths and their total cost. These visualizations not only improve the interpretability of the algorithms but also demonstrate their practical applicability in a visually engaging manner. By providing clear and intuitive representations, the project bridges the gap between theoretical concepts and real-world implementation.

Through this analysis, the study contributes significantly to the exploration of algorithmic efficiency and their application in solving complex problems. It offers valuable insights into how Prim's and Kruskal's algorithms can be effectively utilized in real-world scenarios, particularly for optimizing network-based systems in fields like transportation and urban development.

1.2 Scope and Constraints

Scope:

The project aims to explore the application of Minimum Spanning Tree (MST) algorithms in real-world scenarios, particularly focusing on network optimization problems such as transportation planning and urban development. It provides a platform for analyzing and comparing the performance of Prim's and Kruskal's algorithms, offering insights into their strengths and limitations. The development of a Flask-based web application extends the usability of the project by allowing users to input city names dynamically, compute MSTs, and visualize the results. Tools like Geopy, NetworkX, and Matplotlib ensure accurate computation of edge weights and clear visualization of the MST. The project's scope also includes testing on various datasets, demonstrating its adaptability to different graph structures. This implementation highlights the practical value of MST algorithms and their role in optimizing network connections for cost-effectiveness and efficiency.

Constraints:

Geographical Limitations: The accuracy of edge weights relies on geocoded city coordinates, which may vary due to errors in location data or inaccuracies in the great-circle distance calculations.

Scalability: The system's performance may degrade when handling a large number of nodes or edges, as both algorithms have computational complexities that can impact runtime in dense graphs.

Algorithm-Specific Limitations: Prim's algorithm may not be as efficient for sparse graphs compared to Kruskal's, while Kruskal's algorithm might struggle with edge sorting in dense graphs.

User Input Dependency: The accuracy and relevance of the MST depend on the correctness of the user-provided city names, as invalid or incomplete inputs may result in errors.

Visualization Constraints: While **NetworkX** and **Matplotlib** provide robust visualization, their graphical outputs may become cluttered or less interpretable for very large graphs.

Real-World Generalization: The project is limited to theoretical MSTs and assumes all edges are bidirectional and weighted based on distance, which might not fully account for real-world factors like road conditions, traffic, or terrain.

Static Testing Dataset: The project demonstrates results using a fixed dataset of 13 cities, which might not represent the complexity of larger or more diverse datasets.

1.3 Review and Literature

A comprehensive review of the existing literature related to Minimum Spanning Tree (MST) algorithms reveals a rich landscape of methodologies aimed at optimizing network designs across various fields. The research highlights the significance of MST in minimizing costs while ensuring efficient connectivity, and two of the most commonly studied algorithms in this area are Prim’s and Kruskal’s algorithms.

Prim's and Kruskal's Algorithms: The efficiency of Prim's and Kruskal's algorithms in resolving the MST problem is well known. In sparse graphs, where there are much fewer edges than vertices, Kruskal's technique performs especially well. Sorting every edge and adding the least expensive edge that doesn't form a cycle is how it operates. Prim's technique, on the other hand, works better with dense graphs since it starts at a random node and increases the MST by adding the minimal weight edge that connects the tree to a new node. The two methods are compared in a number of studies that concentrate on their efficiency, time complexity, and use in practical issues.

Applications in Network Design: The primary real-world application of MST algorithms is in network design, where they are used to minimize the cost of connecting various nodes, such as cities, communication networks, and utilities. Researchers have applied MST algorithms to optimize the design of road networks, railway systems, and even electrical grids. These algorithms help minimize the total connection cost while ensuring that all points in the network are interconnected efficiently. The importance of MST in real-life road network design, especially in urban planning, is a recurring theme in the literature.

Computational Efficiency and Performance: The literature emphasizes the computational efficiency of both Prim's and Kruskal's algorithms. While Kruskal's algorithm tends to be more efficient in sparse graphs with fewer edges, Prim's algorithm is favored in dense graphs due to its ability to handle large numbers of edges and vertices effectively. Researchers have also explored variations of these algorithms to improve their performance in specific use cases, such as parallel implementations and optimizations for specific graph types.

Visualization and Interactive Tools: Recent advancements in the visualization of MSTs have significantly enhanced the understanding of algorithmic solutions. Tools such as NetworkX and Matplotlib are commonly used to graphically represent MSTs, allowing users to visualize the optimal paths and their respective costs. The integration of visualization tools has proven to be an effective way of demonstrating the practical application of MST algorithms in fields like urban planning and transportation.

Cross-Disciplinary Applications: The application of MST algorithms has expanded beyond traditional network design. Studies have explored their use in clustering and data analysis, where MST is applied to group similar data points or create efficient data structures for machine learning algorithms. This broadens the utility of MST beyond network optimization and into more diverse scientific and engineering fields.

Challenges and Future Directions: Despite their wide applicability, the literature acknowledges several challenges in implementing MST algorithms in real-world scenarios. These include issues related to the dynamic nature of networks, such as the addition or removal of nodes, and the scalability of algorithms to handle very large datasets. Future research calls for hybrid approaches that combine MST with other optimization techniques and algorithms, as well as more user-friendly and interactive tools that can simplify the process of solving the MST problem for non-experts.

In conclusion, the literature on Minimum Spanning Trees and their algorithms highlights the versatility and efficiency of Kruskal’s and Prim’s algorithms in a wide range of applications. The comparison of these two algorithms, along with their visualization and implementation in network optimization, underscores their importance in real-world applications such as transportation network design, urban planning, and data analysis. Future research will likely focus on improving the scalability and adaptability of these algorithms, as well as enhancing their integration into practical, user-friendly tools.

1.4 Background and Literature Survey

The optimization of road networks has long been a critical area of study, with Minimum Spanning Tree algorithms playing a pivotal role in minimizing total construction costs and improving connectivity. Various real-world applications demonstrate the effectiveness of MST algorithms like Kruskal's and Prim's in solving complex infrastructure challenges. A study by Kotoky titled "Study of Kruskal's Algorithm in Nagaland's Road Network" explores how Kruskal’s algorithm successfully minimized distances between cities and enhanced infrastructure planning.

Similarly, research conducted on route planning in Lobels Bulawayo highlights Kruskal's utility in optimizing road networks, showcasing its practical impact in diverse geographic settings.

The comparative analysis of Kruskal’s and Prim’s algorithms sheds light on their strengths and suitability for different types of networks. An article titled "Comparative Analysis of Kruskal and Prim MST Algorithms" notes that Kruskal's algorithm excels in sparse networks by efficiently handling fewer edges, while Prim's algorithm is more effective in dense networks where a greater number of edges must be evaluated. Further insights from research at Beijing Normal University delve deeper into the unique advantages of both algorithms, providing valuable perspectives on their tailored applications for urban and rural road systems.

In rural settings, Prim’s algorithm has shown particular promise. A study conducted by Arogundade and Akinwale focuses on its application in Nigeria’s transportation systems, where it connects isolated villages and towns at minimal construction costs. By always selecting the nearest available edge, Prim's algorithm ensures cost-effective connectivity, making it especially suitable for rural and less-developed areas. This use case highlights the adaptability of MST algorithms to address varying infrastructural needs.

Beyond transportation, MST algorithms have been effectively applied in other domains, such as communication networks. Research conducted at Shenzhen MSU–BIT University explores the use of Kruskal's algorithm in optimizing large-scale communication systems, drawing parallels with its applications in road networks. Additionally, advancements in geospatial analysis tools, as outlined in studies on interactive visualization for network design, enhance the efficiency and accuracy of MST-based designs by enabling precise edge-weight calculations. These developments demonstrate the versatility of MST algorithms across disciplines, reaffirming their importance in infrastructure development.

1.5 Organization of the Report

The following is a description of the remaining project report chapters:

- Chapter 2 contains the proposed system, standards and software details.
- Chapter 3 covers the outcomes following the project's implementation.
- Chapter 4 concludes the report.
- Chapter 5 consists of codes.
- Chapter 6 gives references.

CHAPTER 2

RESEARCH METHODOLOGY

The research methodology for building a Minimum Spanning Tree (MST) involves several key steps to evaluate the effectiveness of Prim's and Kruskal's algorithms in network optimization. Initially, geographical data of cities, including their names and coordinates, are collected and preprocessed. Using the Geopy library, the system computes pairwise distances between cities via the great-circle distance formula. This distance data is then used to construct a graph where cities are represented as nodes, and the distances between them are the edges. The MST is generated by applying Prim's and Kruskal's algorithms, which are designed to find the minimum cost network connecting all cities. The algorithms are implemented and their results are compared to evaluate performance. Visualization of the MST is achieved using Python tools like NetworkX and Matplotlib, highlighting the optimal path and total cost of the network. The final output is a graphical representation of the MST, along with a list of edges and their respective weights, which showcase the most efficient network structure. The entire process is integrated into a web application, allowing users to interact with the system and input city names to dynamically generate and display MSTs.

Work flow of the model

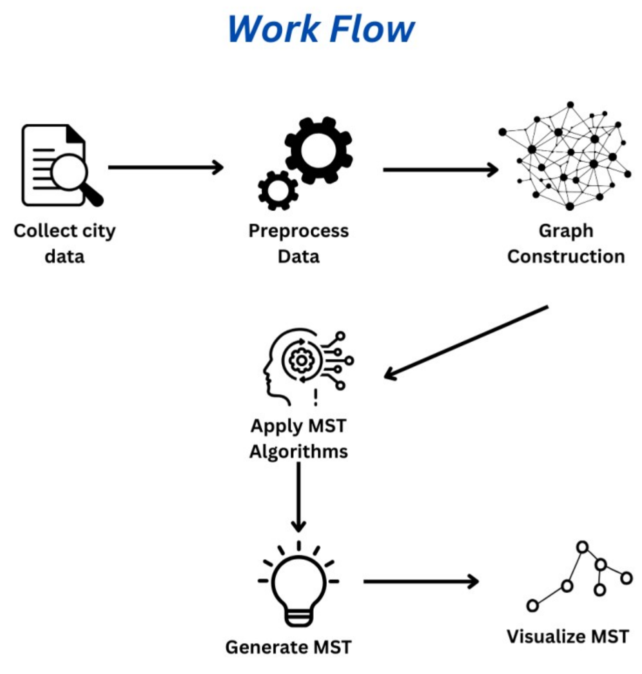
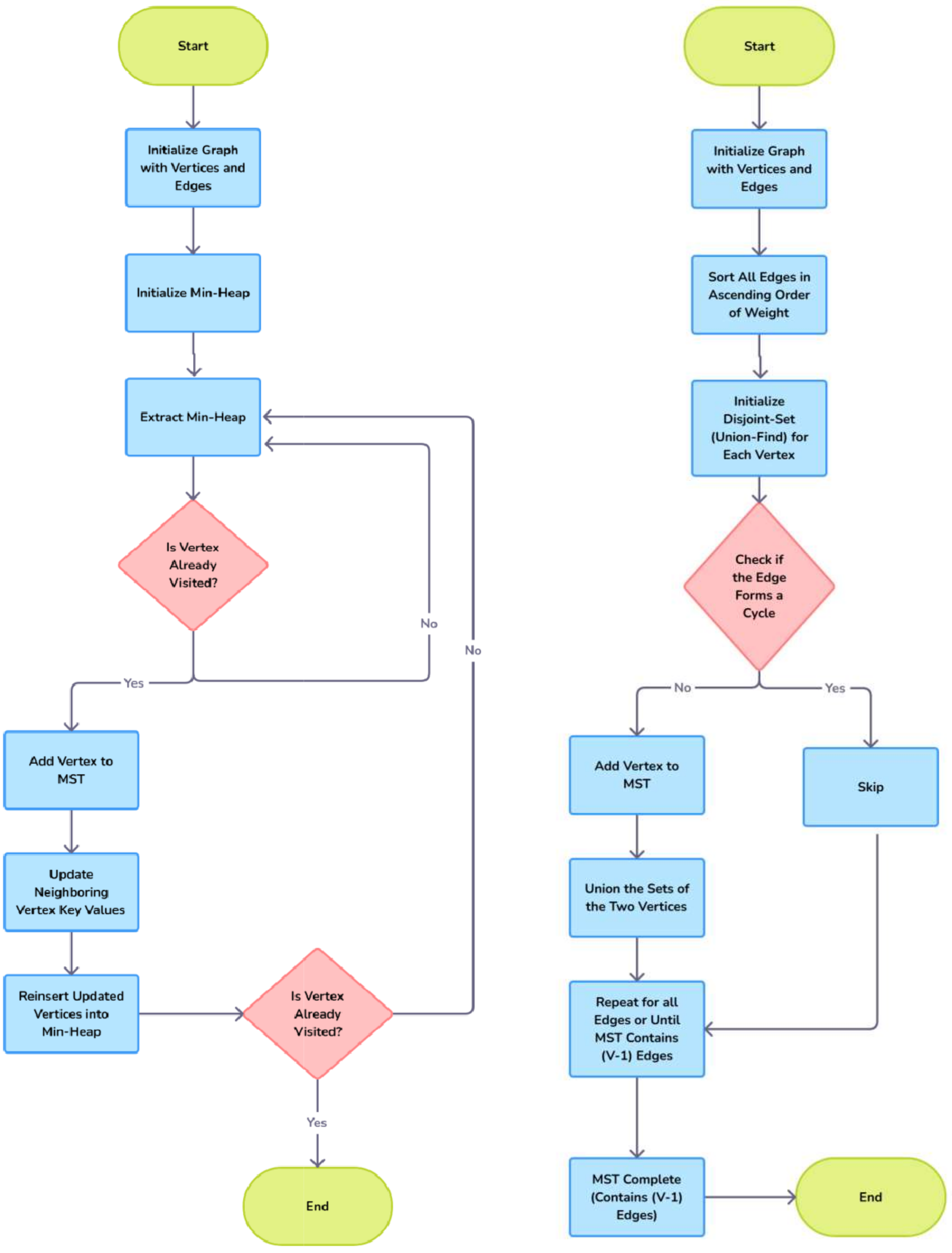


Figure 1 Work flow of the model



Start
Initialize Graph with Vertices and Edges
Initialize Min-Heap
Extract Min-Heap
Is Vertex Already Visited?
No
No
Yes
Add Vertex to MST
Update Neighboring Vertex Key Values
Reinsert Updated Vertices into Min-Heap
Is Vertex Already Visited?
Yes
End
Start
Initialize Graph with Vertices and Edges
Sort All Edges in Ascending Order of Weight
Initialize Disjoint-Set (Union-Find) for Each Vertex
Check if the Edge Forms a Cycle
No
Yes
Add Vertex to MST
Skip
Union the Sets of the Two Vertices
Repeat for all Edges or Until MST Contains (V-1) Edges
MST Complete (Contains (V-1) Edges)
End

Figure 2 Working of the model

2.1 Model building

Data Collection and Preprocessing: Gathering geographic information about cities, such as their names and coordinates, is the first step. The Geopy module is used to convert city names into geographic coordinates (latitude and longitude). The great-circle distance formula is then applied to get the pairwise distances between cities using these coordinates.

Graph Construction: Each city is represented as a node in a graph, and the edges are the distances between the cities. Because there is no clear directionality in the connections connecting the cities, the graph is undirected. Each edge's weight reflects the distance between the two cities it connects.

Algorithm Implementation: The MST problem is solved by implementing both Kruskal's and Prim's algorithms. In order to prevent cycles, Kruskal's algorithm adds the smallest edges to the MST after sorting all the edges in ascending order of their weights. Prim's algorithm, on the other hand, begins with a single node and gradually adds the closest disconnected node, making sure that the smallest weight edge is selected at every stage.

MST Generation: The MST is produced by picking the edges that link every city with the smallest total distance using the selected algorithm (Kruskal's or Prim's). A network with the shortest possible distance between each city is the ultimate product.

Visualization: The final step in the model building involves visualizing the MST using Python libraries such as NetworkX and Matplotlib. The MST is displayed with clear markings for the edges and their respective weights, allowing users to visually interpret the structure and efficiency of the network.

2.2 Proposed System

In this project, the primary objective is to optimize the network design by utilizing two well-established algorithms: Kruskal’s and Prim’s, for solving the Minimum Spanning Tree (MST) problem. The classification of methods refers to the choice of algorithms that will be implemented to compute the optimal connection of cities based on their geographical coordinates. Below are the two proposed classification techniques, each with their own characteristics and approach.

Kruskal’s Algorithm:
Kruskal’s algorithm is a classical approach to finding the MST of a graph, specifically suited for edge-list representations. It follows a greedy strategy where edges are sorted in non-decreasing order of their weights (distances between cities), and the algorithm proceeds by adding the smallest edge to the MST, ensuring no cycles are formed. The algorithm employs a Union-Find data structure to detect cycles and manage connected

components efficiently. It is particularly useful when the graph is sparse, as sorting the edges first allow for an efficient process in finding the MST. Kruskal's approach is very effective for graphs with fewer edges since its time complexity is O(E log E), where E is the number of edges in the graph.

Prim’s Algorithm:
Prim’s algorithm is another greedy algorithm that operates by expanding the MST one vertex at a time. Prim's algorithm starts with an arbitrary node, chooses the minimal weight edge from a vertex inside the MST to a vertex outside of it, and then adds the vertex to the MST. Until every vertex is included, this process is repeated. Prim's algorithm is more efficient for dense graphs and is typically implemented using priority queues or binary heaps for optimal performance. The time complexity is O(E log V)**,** where V is the number of vertices, making it a better choice for graphs where many edges are present.

2.3 Standards

Various standards used in this project are:

Graph Theory Standards: Graph theory forms the foundation of the Minimum Spanning Tree algorithms, and its application to road network design follows established mathematical principles. MST algorithms like Prim’s and Kruskal’s are designed to optimize the construction of road networks by minimizing total construction cost while maintaining connectivity. The use of these algorithms adheres to the graph theory standards for edge-weighted, undirected graphs, where the goal is to find the subset of edges that connects all vertices with the minimum possible total edge weight.

2.4 Software details

Code Editor: Visual Studio Code
Languages Used: Python, HTML, CSS, JavaScript
Libraries and Frameworks: Flask, NetworkX, Matplotlib, Geopy
Tools and Technologies:
- Flask (Web Framework)
- Matplotlib (Graph Visualization)
- Geopy (Geocoding and Distance Calculation)
- NetworkX (Graph Algorithms)

Operating System: Windows

2.5 Procedure

Data Collection

The first step in the Road Network Expansion Project involves collecting data on the cities that will form the nodes of the road network. The cities can be selected based on either real-world geographic data or hypothetical scenarios. For each city, data such as the coordinates (latitude and longitude) and potential road connections with other cities are gathered. This data is essential for calculating distances between cities, which will later serve as the weights for the edges in the graph.

Data Preprocessing

Once the city data is collected, the next step is data preprocessing. In this stage, the raw data is cleaned and organized for use in the graph construction process. This includes calculating the geographic distances between pairs of cities. In case of real-world data, these distances could be determined from road networks or directly through GPS coordinates. The data is formatted into a list of pairs, where each pair consists of two cities and the associated distance (edge weight). The preprocessing step ensures that the data is ready to be translated into a graph representation for the next steps.

Graph Construction

With the preprocessed data, a graph is constructed to represent the road network. Cities are represented as vertices (nodes), and the roads between cities are represented as edges with weights corresponding to the calculated distances. The graph can either be undirected (if roads are bidirectional) or directed (if roads have specific travel directions). This graph serves as the foundation for applying the Minimum Spanning Tree (MST) algorithms. At this stage, the graph includes all potential roads between cities, which may later be optimized using MST techniques.

Apply MST Algorithms (Prim's and Kruskal's)

The heart of the optimization process involves applying MST algorithms to the constructed graph. Two common MST algorithms, Prim's Algorithm and Kruskal's Algorithm**,** are used to identify the optimal set of roads that connect all cities with the least total distance.

- **Prim's Algorithm**: This algorithm begins with a starting node and progressively adds the shortest edge connecting a new node to the growing MST until all cities are connected.
- **Kruskal's Algorithm**: This algorithm sorts all the edges by weight (distance) and adds them to the MST one by one, ensuring no cycles are formed, until the tree includes all cities.

Both algorithms are applied to determine the optimal road network that minimizes the total construction cost while connecting all cities.

Generate MST

After applying either Prim's or Kruskal’s algorithm, the result is the Minimum Spanning Tree. The MST is a subgraph of the original road network that includes only the roads that minimize the total distance and connect all the cities without forming any cycles. This represents the most cost-effective road network for expanding the city connections. The generated MST is crucial as it identifies the optimal solution to the road network expansion problem.

Visualize MST

The final step involves visualizing the optimized road network represented by the MST. A graphical representation is created, where the cities are shown as nodes, and the roads (edges) are drawn connecting them. The visualization helps to clearly illustrate the road network before and after the application of the MST algorithm, highlighting the reduced total distance and the most efficient paths. The visual comparison enables stakeholders to understand the improvements made through the optimization process, providing a tangible representation of the cost savings and efficiency gains achieved.

CHAPTER 3

RESULTS AND DISCUSSIONS

Results

By constructing Minimum Spanning Trees (MST) for the road network graph using Prim's and Kruskal's methods, the overall cost of road building was effectively optimized, guaranteeing connectivity between all cities. By successfully reducing the overall amount of road length needed, both algorithms significantly decreased the expenses associated with developing new infrastructure. Prim's Algorithm created a minimal spanning tree with a shorter overall road length by growing from a chosen starting city and adding the shortest edges. Similarly, a minimal road network at a low cost was also accomplished using Kruskal's Algorithm, which processes edges in increasing order of distance and constructs the MST by joining non-cyclic edges. Comparing both algorithms to the original road network, the overall road length and cost were much decreased.

Despite their distinct methodologies, the two algorithms demonstrated their effectiveness in road network optimization by producing comparable decreases in the total road length. The ability of these algorithms to eliminate unneeded infrastructure was demonstrated by the more effective road construction plan that the MSTs produced by both algorithms offered. Because fewer roads were needed to connect all communities, which resulted in cheaper construction costs, the MST visualization validated the effectiveness of both methods. This demonstrates that the MST algorithms may be used successfully to maximize road network infrastructure planning.
According to the comparisons we have done in the research with different metrics, the following are the best possible results.

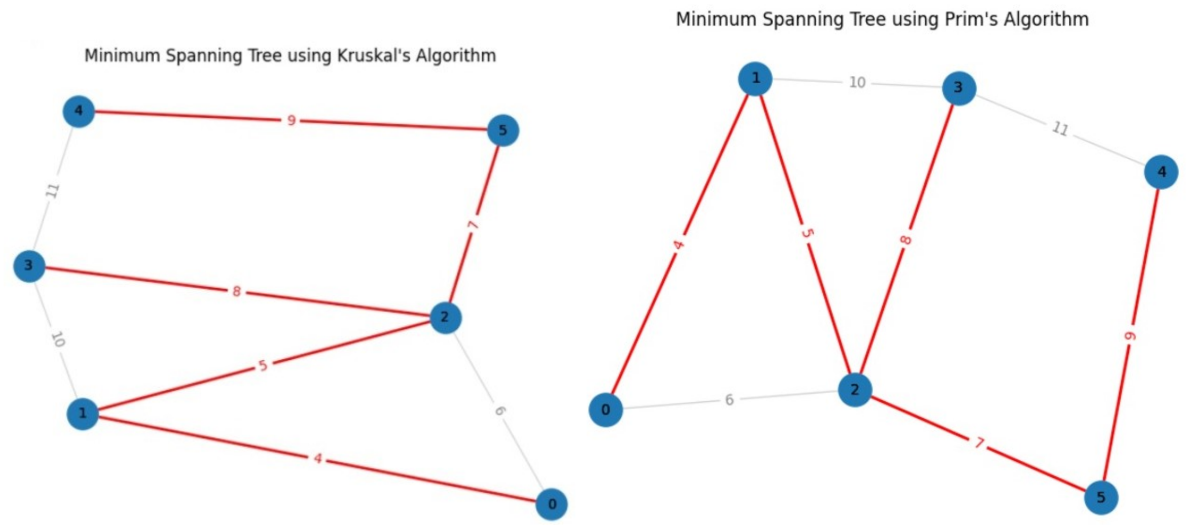

Minimum Spanning Tree using Kruskal's Algorithm
Minimum Spanning Tree using Prim's Algorithm

Figure 3 Kruskal’s and Prim’s MST for 6-node graph.

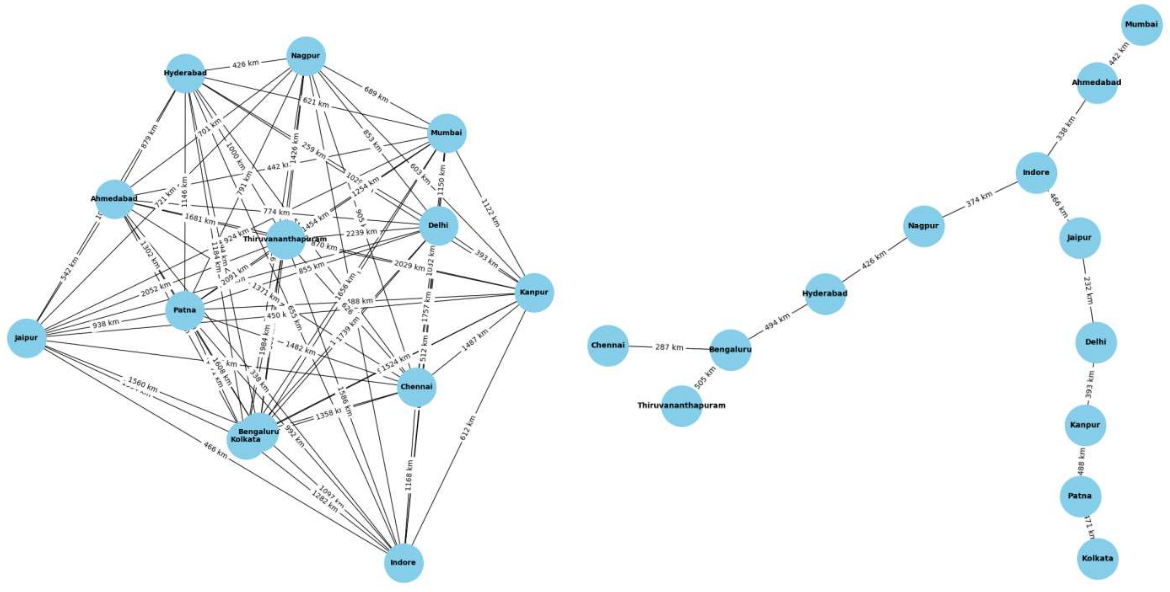


Figure 4 Kruskal’s Before and After MST - 13 cities

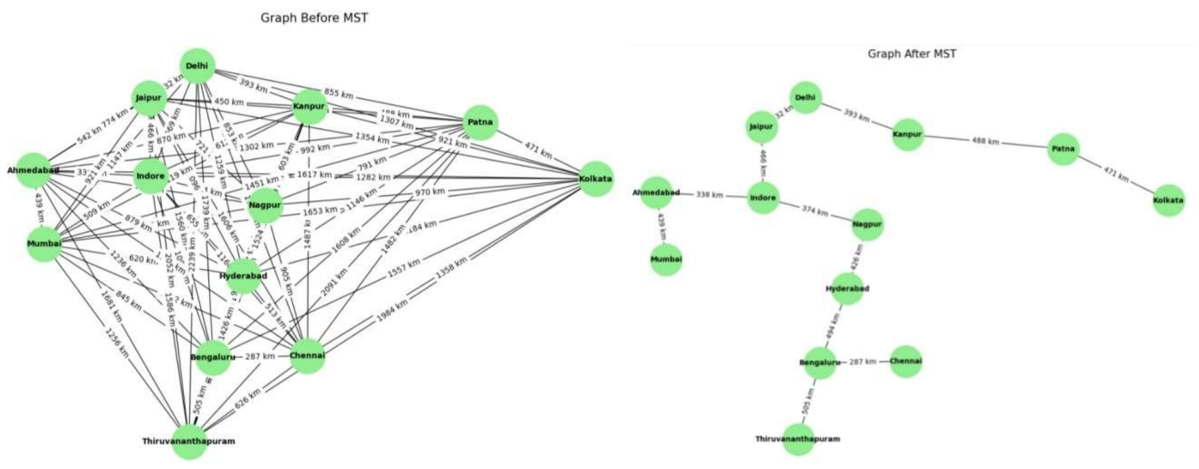


Figure 5 Prim’s Before and Afer MST- 13 cities

Discussions

The results obtained from applying Prim’s and Kruskal’s algorithms were consistent with the expectations of MST algorithms, where both successfully minimized the total length of roads, thus leading to lower costs. However, the performance of each algorithm varied based on the graph's characteristics. Prim’s Algorithm proved to be more efficient in dense networks, where many cities were interconnected, as it efficiently expanded the tree by choosing the nearest city to the already-visited ones. In contrast, Kruskal’s Algorithm, which processes edges in increasing order of weight, performed better in sparse graphs, where fewer connections were available between cities. Despite these differences in performance, both algorithms successfully achieved the goal of minimizing road length and cost.

In the real-world context, applying these algorithms to road network planning would result in significant cost savings for infrastructure development. The reduced number of roads in the MST means fewer resources are required for construction, maintenance, and land use, leading to lower overall expenditures. Moreover, fewer roads also have environmental benefits, as less land would need to be cleared, and fewer materials would be consumed, contributing to more sustainable urban development.

While the results demonstrated the optimization of road networks, the model can be further improved by incorporating real-world complexities. For example, the cost of road construction could vary based on factors such as terrain, population density, and geographical challenges, which could be represented as varying edge weights. Additionally, aspects such as traffic flow, road safety, and environmental impacts could be integrated into the model to make it more realistic and applicable to actual urban planning scenarios.

The project provides a solid basis for future work in infrastructure planning, with potential improvements that can reflect dynamic, evolving road networks or include additional factors beyond mere distance. The application of MST algorithms in large-scale infrastructure projects could lead to more cost-effective, sustainable, and optimized urban development strategies. By offering a more efficient layout of roads, these algorithms could significantly enhance the planning of transportation routes, ultimately benefiting city planners, developers, and the community as a whole.

CHAPTER 4

CONCLUSION AND FUTURE WORKS

Conclusion

Using the Minimum Spanning Tree technique, the project effectively applied and evaluated Kruskal's and Prim's algorithms to create an ideal road network. Prim's algorithm proved to be more successful in denser networks, whereas Kruskal's approach proved to more efficient in sparse road networks. The experiment demonstrated these algorithms advantages in reducing overall construction costs while maintaining connectedness among urban neighborhoods by comparing their performance.

With its insights into how MST algorithms might be applied to improve road network designs for economical infrastructure development, this project is a useful resource for engineers and urban planners. It also lays the groundwork for future research, including the incorporation of further optimization criteria and the application of these algorithms to bigger and more intricate real-world datasets. With the possibility for future improvements like dynamic network updates and comparisons with other network optimization techniques, the project's modularity and scalability make it a crucial contribution to network design research.

Future Works

This project showcases a solid implementation of Prim’s and Kruskal’s Algorithms for constructing the Minimum Spanning Tree (MST), laying the groundwork for future advancements. One promising direction is developing mobile apps that provide real-time traffic updates, road closures, and route optimization, with GPS and user-generated data to keep the network current. This would enhance travel planning and improve road network adaptability.

Additionally, expanding the algorithms to manage other urban infrastructure, like water pipelines, electricity grids, and communication networks, can optimize service delivery and reduce operational costs. Real-time updates, such as traffic changes or disruptions, would make the system more responsive. Integrating AI for traffic prediction could further optimize routes and support better urban planning by forecasting long-term traffic trends.

CHAPTER 5

APPENDIX

Code for Home Page Layout and Structure

(index.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>MST Visualization</title>
   <style>
      body {
         background-image: url('/static/hpbg.jpg');
         background-size: cover;
         color: white;
         text-align: center;
         font-family: Arial, sans-serif;
      }
      textarea, button {
         font-size: 1.2em;
         margin: 10px;
      }
   </style>
</head>
<body>
   <h1>MST Visualization Tool</h1>
   <form action="/submit" method="POST">
      <textarea name="cities" rows="10" cols="50" placeholder="Enter city names
        separated by commas"></textarea><br>
      <button type="submit">Generate MST</button>
   </form>

</body>
</html>
```

Code for form submission

(submit.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>MST Results</title>
    <link rel="stylesheet" href="/static/style.css"> <!-- Link to external CSS -->
    <style>
        /* Styling for the container */
        body {
            background-image: url('/static/hpbg.jpg');
            background-size: cover;
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 0;
            color: white;
            text-align: center;
        }

        .container {
            max-width: 90%;
            margin: 20px auto;
            padding: 20px;
            background: rgba(0, 0, 0, 0.7);
            border-radius: 10px;
        }
```

```
h1, h2, h3 {
   margin: 10px 0;
}

img {
   max-width: 100%;
   border: 2px solid white;
   border-radius: 10px;
   margin: 20px 0;
}


/* Adjusting the image styles dynamically */
.graph-image.small {
   width: 40%;
}
.graph-image.medium {
   width: 70%;
}
.graph-image.large {
   width: 90%;
}
ul {
   text-align: left;
   display: inline-block;
   margin-top: 20px;
   padding: 0;
}
ul li {
   margin: 5px 0;
```

```
        }
        .error-message {
            color: red;
            font-size: 1.2em;
            margin-top: 20px;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>MST Results</h1>
        <div id="results">
            <h2>Graph Before MST</h2>
            <img id="before-mst" src="{{ before_mst_image }}" alt="Graph Before
MST" class="graph-image"><br>

            <h2>Graph After MST</h2>
            <img id="after-mst" src="{{ after_mst_image }}" alt="Graph After MST"
class="graph-image"><br>

            <h3>MST Edges:</h3>
            <ul id="mst-edges"></ul>
        </div>
    </div>

    <script>
        // Step 1: Pass MST edges from Flask backend as a JavaScript object
        const mstEdges = {{ mst_edges|tojson }};

        // Step 2: Dynamically adjust graph image sizes based on content and ensure
visibility
```

```
function adjustImageSize(imageId) {
    const img = document.getElementById(imageId);

    // Wait for the image to load to access its dimensions
    img.onload = () => {
        const width = img.naturalWidth;
        const height = img.naturalHeight;

        // Dynamically adjust size based on dimensions
        if (width < 500 && height < 500) {
            img.classList.add("small");
        } else if (width < 1000 && height < 1000) {
            img.classList.add("medium");
        } else {
            img.classList.add("large");
        }
    };
}

// Step 3: Populate the MST edges dynamically
if (mstEdges && mstEdges.length > 0) {
    const edgesList = document.getElementById("mst-edges");
    mstEdges.forEach(edge => {
        const li = document.createElement("li");
        li.textContent = `${edge.from} -> ${edge.to}: ${edge.weight} km`;
        edgesList.appendChild(li);
    });
    // Adjust image sizes
    adjustImageSize("before-mst");
    adjustImageSize("after-mst");
```

```
        } else {
            // If no data is found, show an error message
            document.body.innerHTML = "<h1>No data available. Please generate MST first.</h1>";
        }
    </script>
</body>
</html>
```

Code for Flask Backend

(app.py)

```
from flask import Flask, render_template, request
from geopy.distance import great_circle
from geopy.geocoders import Nominatim
import os

import matplotlib
matplotlib.use('Agg')  # Fixes matplotlib GUI warnings
import matplotlib.pyplot as plt
import networkx as nx
from geopy.exc import GeocoderTimedOut, GeocoderUnavailable

app = Flask(__name__)

# Fetch city coordinates
def fetch_coordinates(cities):
    geolocator = Nominatim(user_agent="city_locator")
    location_dict = {}
    for city in cities:
        try:
            location = geolocator.geocode(city, timeout=10)  # Increase timeout
            if location:
                location_dict[city] = (location.latitude, location.longitude)
            else:
                print(f"Could not find location for city: {city}")
                raise ValueError(f"Invalid city: {city}")
        except (GeocoderTimedOut, GeocoderUnavailable) as e:
            print(f"Geocoding error for city '{city}': {e}")
            raise ValueError(f"Geocoding service is unavailable for city: {city}")
    return location_dict

# Graph Implementation (Kruskal's)
```

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    def KruskalMST(self):
        result = []
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []

        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        e = 0
        i = 0
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i += 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                result.append([u, v, w])
                self.union(parent, rank, x, y)
                e += 1
        return result

# Visualization
def visualize_graph(cities, location_dict, edges, filename, title):
    try:
        # Create the graph and add edges
        G = nx.Graph()
        positions = {i: location_dict[city] for i, city in enumerate(cities)}

        for edge in edges:
            u, v, w = edge
            G.add_edge(u, v, weight=w)

        # Use spring layout for uniform edge lengths
        pos = nx.spring_layout(G, scale=2, seed=42)  # `scale` controls overall spacing

        # Adjust figure size dynamically based on the number of nodes
        num_nodes = len(cities)
        if num_nodes <= 3:
            figsize = (5, 5)  # Small graph
        elif num_nodes <= 6:
            figsize = (8, 8)  # Medium graph
        else:
            figsize = (12, 12)  # Large graph

        # Plot the graph
        plt.figure(figsize=figsize)
        nx.draw(G, pos, with_labels=True, labels={i: cities[i] for i in range(len(cities))},
                node_color="skyblue", node_size=3000, font_size=10, font_weight="bold")

        # Add edge labels with weights
        edge_labels = nx.get_edge_attributes(G, 'weight')
        nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): f"{w} km" for u, v,
w in edges})

        plt.title(title, fontsize=15)
        plt.tight_layout()  # Adjust padding
        plt.savefig(filename)
        plt.close()

    except Exception as e:
        print(f"Error visualizing graph: {e}")



# Flask Routes
@app.route("/")
def index():
    return render_template("index.html")

@app.route("/submit", methods=["POST"])
def submit():
    try:
        cities = request.form.get("cities", "").split(",")
        cities = [city.strip() for city in cities if city.strip()]

        if not cities:
            return render_template("submit.html", error="No cities provided.")

        location_dict = fetch_coordinates(cities)

        g = Graph(len(cities))
        for i, city1 in enumerate(cities):
            for j, city2 in enumerate(cities):
                if i < j:
                    dist = int(great_circle(location_dict[city1], location_dict[city2]).km)
                    g.addEdge(i, j, dist)

        mst_result = g.KruskalMST()

        os.makedirs("static", exist_ok=True)
        graph_before_path = "static/Graph_Before_MST.png"
        graph_after_path = "static/Graph_After_MST.png"
        visualize_graph(cities, location_dict, g.graph, graph_before_path, "Graph Before
MST")
        visualize_graph(cities, location_dict, mst_result, graph_after_path, "Graph After
MST")

        mst_edges = [{"from": cities[u], "to": cities[v], "weight": w} for u, v, w in
mst_result]

        return render_template(
            "submit.html",
            mst_edges=mst_edges,
            before_mst_image="/" + graph_before_path,
            after_mst_image="/" + graph_after_path,
        )
    except Exception as e:
        print(f"Error in /submit: {e}")
        return render_template("submit.html", error=str(e), mst_edges=[])

@app.route("/generate_mst", methods=["POST"])
def generate_mst():
    try:
        cities = request.json.get("cities", [])
        if not cities:
            return jsonify({"error": "No cities provided."}), 400

        # Fetch coordinates
        location_dict = fetch_coordinates(cities)

        # Build the graph
        g = Graph(len(cities))
        for i, city1 in enumerate(cities):
            for j, city2 in enumerate(cities):
                if i < j:
                    dist = int(great_circle(location_dict[city1], location_dict[city2]).km)
                    g.addEdge(i, j, dist)

        # Generate MST
        mst_result = g.KruskalMST()

        # Save the graphs
        os.makedirs("static", exist_ok=True)
        graph_before_path = "static/Graph_Before_MST.png"
        graph_after_path = "static/Graph_After_MST.png"
        visualize_graph(cities, location_dict, g.graph, graph_before_path, "Graph Before 
MST")
        visualize_graph(cities, location_dict, mst_result, graph_after_path, "Graph After 
MST")

        # Prepare response data
        mst_edges = [{"from": cities[u], "to": cities[v], "weight": w} for u, v, w in
mst_result]
        return jsonify({
            "mst_edges": mst_edges,
            "before_mst_image": "/" + graph_before_path,
            "after_mst_image": "/" + graph_after_path
        })
    except Exception as e:
        print(f"Error: {e}")
        return jsonify({"error": str(e)}), 500


if __name__ == "__main__":
    app.run(debug=True)
```

CHAPTER 6

REFERENCES

[1] Arogundade, O. T., & Akinwale, A. T. (n.d.). On the application of Prim's algorithm to transportation in rural Nigeria. University of Agriculture, Abeokuta.

[2] Ayegba, P. O., et al. (2016). Comparative analysis of Kruskal and Prim MST algorithms. IOSR Journal of Computer Engineering, 12(4), 41–45.

[3] Chen, J. (n.d.). Analysis of Prim, Kruskal, and Boruvka algorithms. Beijing Normal University - HKBU UIC.

[4] Dandure, F. M. (n.d.). Route planning using Kruskal's algorithm: Lobels Bulawayo.

[5] Effanga, E. O., & Edeke, U. E. (n.d.). Minimum spanning tree of road networks in Nigeria. University of Calabar.

[6] Kotoky, M. (2020). Study of Kruskal's algorithm in Nagaland's road network. IJERT, 13(12), 5386–5391.

[7] Melnikov, B. F., & Terentyeva, Y. Y. (n.d.). Communication networks: Kruskal's algorithm. Shenzhen MSU–BIT University, Moscow.

[8] Interactive Visualization Tools for Network Analysis (n.d.). Enhancing algorithmic efficiency in MST exploration.

[9] Ghosh, A., & Ghosh, P. (2020). Comparative study of Prim's and Kruskal's algorithms for optimal road network design. *International Journal of Advanced Research in Computer Science and Software Engineering*, 10(3), 67–72.

[10] Liang, Z. (n.d.). Application of MST in urban infrastructure planning. *Urban Planning Journal*.

[11] Singh, R., & Verma, S. (2019). Performance analysis of MST algorithms for large-scale road network optimization. *Engineering Computations*, 36(4), 890–912.

[12] Thomas, J., & Natarajan, M. (2021). Algorithmic approaches to transportation planning using Kruskal's algorithm. *IEEE Transactions on Intelligent Transportation Systems*, 22(7), 5203–5215.

[13] Network Analysis for Transportation Infrastructure (n.d.). MST visualization and implementation in complex road systems. *Algorithmic Insights*.