MA308 Mini Project
Report

Designing a shortest-path algorithm for large-scale graphs

Submitted in partial fulfillment of MA308 (Mini Project) requirements
IIIrd Year / VIth Semester - May 2025

Master of Science

Submitted by

Roll No.	Names of Students
I22MA023	Abhinav Kumar
I22MA038	Raj Kumar
I22MA062	Chandra Pratap

Under the guidance of
Dr. Sushil Kumar

Department of Mathematics
Sardar Vallabhbhai National Institute of Technology, Surat

Even Semester 2024

Acknowledgement

We, the students of the 5-Year Integrated M.Sc. in Mathematics program at Sardar Vallabhbhai National Institute of Technology, Surat, would like to express our sincere gratitude to everyone who supported and guided us throughout this mini-project.

Our deepest thanks go to Dr. Sushil Kumar, whose exceptional mentorship, guidance, and encouragement were pivotal to the success of this project. His insightful feedback and intellectual challenges have greatly enriched our learning experience, making this a rewarding and transformative journey.

We are also grateful to Dr. Jayesh M. Dhodiya, Head of the Department of Mathematics, for his leadership and support in cultivating an environment that fosters academic excellence and research innovation. Additionally, we thank all the faculty members, research scholars, and non-teaching staff of the department for their assistance, constant encouragement, and readiness to help whenever needed.

Abhinav Kumar (I22MA023)

Raj Kumar (I22MA038)

Chandra Pratap (I22MA062)

Department of Mathematics

Sardar Vallabhbhai National Institute of Technology, Surat
SURAT-395007, GUJARAT, INDIA

CERTIFICATE

This is to certify that this is a bonafide record of the project presented by the students whose names are given below during the even semester of 2024 in partial fulfilment of the requirements of the degree of Integrated Masters of Science in Mathematics.

Roll No	Names of Students
I22MA023	Abhinav Kumar
I22MA038	Raj Kumar
I22MA062	Chandra Pratap

Dr. Sushil Kumar
(Project Guide)

Dr. Ramakanta Meher
(Course Coordinator)

Date: 20 January, 2025

Department of Mathematics

Sardar Vallabhbhai National Institute of Technology, Surat
SURAT-395007, GUJARAT, INDIA

APPROVAL SHEET

The Mini Project report entitled **Designing a Shortest Path Algorithm for Large Scale Graphs** by Abhinav Kumar, Raj Kumar and Chandra Pratap is approved for the completion of course MA-308 (Mini Project) for the degree of Master of Science in Mathematics.

Dr. X
(Internal Examiner - I)

Dr. Y
(Internal Examiner - II)

Dr. Sushil Kumar
(Project Guide)

Date: 20 January, 2025

Department of Mathematics

Sardar Vallabhbhai National Institute of Technology, Surat
SURAT-395007, GUJARAT, INDIA

DECLARATION

Report entitled **Designing a Shortest Path Algorithm for Large Scale Graphs** by Abhinav Kumar, Raj Kumar and Chandra Pratap is approved for the completion of course MA-308 (Mini Project) for the degree of Master of Science in Mathematics.

Abhinav Kumar
Admission No.: I22MA023
Department of Mathematics
Sardar Vallabhbhai National Institute of Technology
Surat-395007

Raj Kumar
Admission No.: I22MA038
Department of Mathematics
Sardar Vallabhbhai National Institute of Technology
Surat-395007

Chandra Pratap
Admission No.: I22MA062
Department of Mathematics
Sardar Vallabhbhai National Institute of Technology
Surat-395007

Abstract

This project focuses on designing, implementing, and analyzing efficient algorithms for shortest path calculations on large-scale graphs, including social, road, and communication networks. The goal is to optimize computation time, memory usage, and scalability while ensuring high accuracy and adaptability to real-world scenarios.

The proposed solution combines traditional algorithms like Dijkstra's and Bellman-Ford with advanced techniques such as A*, Contraction Hierarchies, and bidirectional search.

Publicly available datasets, including road networks from OpenStreetMap, are used to evaluate performance based on execution time, memory efficiency, and scalability.

Deliverables include a research report, an optimized algorithm codebase, a visualization tool for shortest path computations, performance analysis, and a discussion on strengths, limitations, and further improvements.

Contents

1 Problem Definition **1**

2 Introduction **2**
2.1 Importance of Sortest Path Calulation 2
2.2 Objective 3
2.3 Scope 3

3 Literature Review **4**
3.1 Classical shortest path algorithms 4
3.1.1 Breadth-first Search 4
3.1.2 Bellman-ford Algorithm 5
3.1.3 Dijkstra's Algorithm 6
3.2 Advanced shortest path algorithms 8
3.2.1 A* Search Algorithm 8
3.2.2 Bidirectional Search 9
3.3 Preprocessing techniques 11
3.3.1 Contraction Hierarchies 11
3.3.2 A* Landmark Technique Algorithm 12
3.3.3 Hub Labeling 14
3.4 Summary of Findings 15

4 Algorithm Design **16**
4.1 Requirements and Design Approach 16
4.2 Version 1: Basic Dijkstra's Algorithm 18
4.2.1 Performance and Limitations 18
4.3 Version 2: A* Search Algorithm 19
4.3.1 Performance and Limitations 19
4.4 Version 3: Bidirectional Dijkstra's Algorithm 20
4.4.1 Performance and Limitations 20
4.5 Version 4: Bidirectional A* Search Algorithm 21
4.5.1 Performance and Limitations 21

4.6 Version 5: Bidirectional A* Search with ALT Preprocessing . . 22
4.6.1 Performance and Limitations 22

5 Implementation **24**
5.1 Implementation Details 24
5.1.1 Key Python Files 24
5.1.2 Libraries Used 24
5.1.3 Graph Generation and Pathfinding 25

6 Result and Analysis **33**
6.1 Version 1: Basic Dijkstra's Algorithm 34
6.2 Version 2: A* Search Algorithm 34
6.3 Version 3: Bidirectional Dijkstra's Algorithm 35
6.4 Version 4: Bidirectional A* Search Algorithm 36
6.5 Final Version: Bidirectional A* Search with ALT Preprocessing 36

7 Discussion **39**
7.1 Strengths and Limitations of the Implemented Solution 39
7.1.1 Enhanced Computational Efficiency 39
7.1.2 Deterministic Execution for Consistency 39
7.1.3 Diagnostic Visualization Support 39
7.2 Challenges and Areas for Improvement 40
7.2.1 Static Preprocessing Constraints 40
7.2.2 Inflexible Cost Metrics 40
7.2.3 Scalability Constraints on Large Networks 40
7.2.4 Suboptimal Landmark Selection 40
7.3 Potential Enhancements 40
7.3.1 Integration of Contraction Hierarchies 40
7.3.2 Support for Dynamic Graphs 41
7.3.3 Customizable Cost Functions 41
7.3.4 Refined Landmark Selection 41

8 Conclusion **42**

References **43**

Appendices **44**

A **45**
A.1 Proof of correctness for BFS[1] 45
A.2 Proof of complexity for BFS[1] 45

A.3 Proof of correctness for Bellman-Ford[2] 46
A.4 Proof of complexity for Bellman-Ford[2] 46
A.5 Proof of correctness for Dijkstra's[3] 47
A.6 Proof of complexity for Dijkstra's [3] 47
A.7 Proof of correctness for A* search[4] 48
A.8 Proof of correctness for Bidirectional Search[5] 48
A.9 Proof of complexity for Bidirectional Search[5] 49
A.10 Proof of correctness for Contraction Hierarchies[6] 49
A.11 Proof of complexity for Contraction Hierarchies[6] 50
A.12 Proof of correctness for ALT[7] 51
A.13 Proof of complexity for ALT[7] 51
A.14 Proof of correctness for Hub Labellling[8] 52
A.15 Proof of complexity for Hub Labelling[8] 53

B **54**
B.1 Final Algorithm 54

List of Figures

4.1 Subset of Surat's road network used for pathfinding experiments. 17
4.2 Visualization of the execution time for Version 1. 18
4.3 Visualization of the execution time for Version 2. 19
4.4 Visualization of the execution time for Version 3. 20
4.5 Visualization of the execution time for Version 4. 21
4.6 Visualization of the execution time for Version 5. 23

6.1 Visualization of explored edges at $t = 10s$ for Version 2 (A* search) vs Version 3 (Bidirectional Dijkstra's). 35
6.2 Visualization of all the explored edges in Version 4 (Bidirectional A*) vs Final Version (ALT + Bidirectional A*). 37

Chapter 1

Problem Definition

The primary objective of this project is to develop and analyze a **hybrid shortest path algorithm** that integrates **preprocessing and efficient querying** to optimize route computation in large-scale networks.

Traditional shortest path algorithms, such as Dijkstra's and Bellman-Ford, while effective in small-scale applications, struggle with computational inefficiencies in massive graphs. To overcome this, our approach introduces a preprocessing stage that enhances query response time, making real-time routing feasible even in complex environments.

By leveraging preprocessing, the algorithm efficiently indexes the network structure, significantly reducing computation time during query execution.

Chapter 2

Introduction

2.1 Importance of Sortest Path Calulation

- **Efficiency in Large-Scale Systems**: In large graphs, such as road networks or the internet, finding an optimal route is essential to saving time, energy, and resources. These systems often involve millions of nodes and edges, requiring algorithms that handle complexity efficiently.
- **Optimization and Cost Reduction**: Many industries rely on shortest path calculations to minimize costs. For example, logistics companies use them to determine the most fuel-efficient routes for deliveries.
- **Road Networks and Navigation Systems**: GPS services like Google Maps calculate the shortest or fastest route to a destination based on real-time traffic data, distance, and road conditions.
- **Network Routing**: In computer networks, protocols like Open Shortest Path First (OSPF) or Border Gateway Protocol (BGP) rely on shortest path calculations to ensure efficient data transfer.
- **Social Network Analysis**: Platforms like LinkedIn or Facebook use these methods to determine the "degree of separation" between users or suggest connections.

2.2 Objective

The algorithm aims to:

1. **Accelerate shortest path queries**: The use of preprocessing optimizes search efficiency, enabling near-instantaneous path retrieval in large-scale networks.

2. **Ensure scalability**: The algorithm is designed to handle extensive datasets, making it applicable to real-world scenarios, from urban traffic management to large-scale logistics planning.

2.3 Scope

This project is centered on developing a **hybrid shortest path algorithm** with a focus on the following key areas:

- **Road Networks**: The algorithm is specifically designed for road networks, where edge weights represent dynamic attributes such as travel time, distance, or toll fees. The approach ensures efficient routing solutions in real-world transportation systems.

- **Scalability and Efficiency**: Given the vast size of modern transportation and logistics networks, the algorithm must be capable of handling millions of nodes and edges while maintaining optimal performance.

- **Preprocessing for Speed Optimization**: Since traditional shortest path algorithms are computationally expensive, the project emphasizes the role of preprocessing in reducing query response times, ensuring rapid access to route data even in complex graphs.

Chapter 3

Literature Review

In this chapter, we review the foundations and advanced algorithms for shortest path calculations, including preprocessing techniques. The section spans classical algorithms, advanced algorithms and recent advances in graph-optimization.

3.1 Classical shortest path algorithms

3.1.1 Breadth-first Search

Introduction

Breadth-first search is a graph traversal algorithms invented by Konrad Zuse in 1945, that can also be used to find the shortest path from a source vertex to a destination vertex in an unweighted graph.

Algorithm

1. Mark all vertices as unvisited.
2. Assign $distance[u] = \infty$ for all vertices except the source vertex s, where $distance[s] = 0$.
3. Use a queue to track vertices to explore. Start with the source vertex s.
4. Dequeue a vertex u.
5. For each neighbour v of u, If v is unvisited (i.e., $distance[v] = \infty$):
    - Set $distance[v] = distance[u] + 1$.

- Mark v as visited.
- Enqueue v.

6. The algorithm ends when the queue is empty. Unreachable vertices retain $distance = \infty$.

This algorithm is mathematically predisposed to find the shortest path from a source vertex s to every other vertex in the graph (see **Appendix A.1** for a formal proof).

Complexity

When finding the shortest path between a pair of vertices in a graph, the worst-case time complexity for the BFS algorithm is $O(V)$ for queue operations + $O(E)$ for edge processing, netting a worst-case time complexity of $O(V + E)$ (see **Appendix A.2** for a formal proof).

The space complexity for BFS is $O(V)$ since we use a queue to store the vertices yet to be explored.

Pros and Cons

- The algorithm is simple and efficient for unweighted graphs.
- BFS works well for large, sparse graphs.
- BFS fails for shortest-path problems in weighted graphs, which are more useful when modelling real world scenarios.

3.1.2 Bellman-ford Algorithm

Introduction

The Bellman–Ford algorithm is a shortest-path algorithm that utilizes dynamic programming to compute shortest paths from a single source vertex to all of the other vertices in a weighted, directed graph. It was first published by Richard Bellman (1958) and Lester Ford Jr. (1956), hence its name.

Algorithm

1. Create an array *distance* of size V to store the shortest path distances.

2. Assign $distance[u] = \infty$ for all vertices except the source vertex s, where $distance[s] = 0$.

3. Repeat $V - 1$ times:

   - For each edge $(u, v) \in E$, if $distance[u] + w(u, v) < distance[v]$ update $distance[v] = distance[u] + w(u, v)$.

4. Now to detect a negative cycle, for each edge $(u, v) \in E$, if $distance[u] + w(u, v) < distance[v]$, report that a negative-weight cycle exists.

5. If no negative-weight cycle is detected, Return the *distance* array as the shortest path distances.

Refer to **Appendix A.3** for a formal proof of correctness of this algorithm.

Complexity

When finding the shortest path from a source vertex to every other vertex in a graph, the worst-case time complexity for the Bellman-Ford algorithm is $O(V \cdot E)$ (see **Appendix A.4** for a formal proof).

The space complexity for Bellman-Ford is $O(V)$ since we use an array of size V to store all the shortest-path distances.

Pros and Cons

- Suitable for applications requiring negative weight handling, in which case it can detect the existence of a negative cycle.
- The Bellman-Ford algorithm is more complex than Dijkstra's algorithm.
- Much slower compared to Dijkstra's algorithm.

3.1.3 Dijkstra's Algorithm

Introduction

Dijkstra's algorithm is a greedy algorithm used to find the shortest paths from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

Algorithm

1. Create an array $distance$ of size V to store the shortest path distances and a priority queue Q containing all vertices, prioritized by $distance$.

2. Assign $distance[u] = \infty$ for all vertices except the source vertex s, where $distance[s] = 0$.

3. While Q is not empty:

   - Extract the vertex u u with the smallest distance distance from Q.
   - For each neighbor v of u, if $distance[u] + w(u, v) < distance[v]$: update $distance[v] = distance[u] + w(u, v)$ and the priority of v in Q.

4. The algorithm ends when Q is empty. The distance distance array contains the shortest path distances from s to all other vertices.

Refer to **Appendix A.5** for a formal proof of correctness of this algorithm.

Complexity

When finding the shortest path from a source vertex to every other vertex in a graph, the worst-case time complexity for the Dijkstra's algorithm is $O((V + E) \log V)$ using a binary heap or $O(V \log V + E)$ using a Fibonacci heap. (see **Appendix A.6** for a formal proof).

The space complexity for Dijkstra's is $O(V)$ since we use an array of size V to store all the shortest-path distances.

Pros and Cons

- Can cover a large area of a graph, which is useful when there are multiple target nodes.
- Can't calculate the shortest paths correctly if the graph has negative weights.
- Has linearithmetic complexity when implemented using a priority queue.

3.2 Advanced shortest path algorithms

3.2.1 A* Search Algorithm

Introduction

A* search is a heuristic-based algorithm used to find the shortest path from a start node to a goal node in a weighted graph. It combines the strengths of Dijkstra's algorithm (guaranteed shortest path) and greedy best-first search (efficient exploration using heuristics). It was first published by Peter Hart, Nils Nilsson, and Bertram Raphael at Stanford Research Institute in 1968.

Algorithm

1. Create a priority queue Q to store nodes to explore, prioritized by $f(v) = g(v) + h(v)$, where
   - $g(v)$: Cost of the shortest path from s to v found so far.
   - $h(v)$: Heuristic estimate of the cost from v to t.
2. Set $g(s) = 0$ and $f(s) = h(s)$.
3. Insert s into Q.
4. Create a set *visited* to track visited nodes
5. While Q is not empty:
   (a) Extract the node u with the smallest $f(u)$ from Q.
   (b) If $u = t$, return the path from s to t.
   (c) Mark u as visited.
   (d) For each neighbor v of u, if v is not visited:
      - Compute $g_{tentative} = g(u) + w(u, v)$.
      - If $g_{tentative} < g(v)$ or v is not in Q:
        – Update $g(v) = g_{tentative}$.
        – Update $f(v) = g(v) + h(v)$.
        – Insert v into Q (or update its priority if already in Q).
6. If Q becomes empty and the goal t has not been reached, no path exists.

A* search is correct if the heuristic $h(v)$ is admissible (never overestimates the true cost to the goal) and consistent (satisfies the triangle inequality: $h(u) \leq w(u, v) + h(v)$ for all edges (u, v)). For a formal proof of correctness, refer to **Appendix A.7**.

Complexity

Since A* Search is basically an 'informed' version of Dijkstra's algorithm, the space complexity for A* search is the same as for Dijkstra's, which is $O(V)$. The time complexity, however, depends on the heuristic function and is equal to Dijsktra's when the heuristic $h(v) = 0$.

Pros and Cons

- Compared to uninformed search algorithms, A* explores significantly fewer nodes leading to faster search times.
- By maintaining a priority queue, A* only needs to store a limited number of nodes in memory, making it suitable for large search spaces.
- Performance heavily depends on the quality of the heuristic function. Thus, A* search is not ideal when a good heuristic cannot be easily defined or when heuristic calculations are complicated.

3.2.2 Bidirectional Search

Introduction

Bidirectional Search is a graph traversal algorithm that explores a graph by simultaneously conducting two searches: one starting from the initial (source) node and moving forward, and another starting from the goal (target) node and moving backward. The two searches "meet" when a common node is detected in their exploration paths.

Algorithm

- Maintain two queues: `forward_queue`, `backward_queue` and two visited sets: `forward_visited`, `backward_visited`.
- Expand nodes level-by-level from both directions, typically using BFS for optimal shortest-path guarantees. For each iteration:

  - **Forward Search**: Dequeue a node from `forward_queue`, mark it as visited in `forward_visited`, and enqueue its unvisited neighbors.
  - **Backward Search**: Dequeue a node from `backward_queue`, mark it as visited in `backward_visited`, and enqueue its unvisited predecessors (reverse neighbors).

- After each step, check if the current node in either direction exists in the opposite visited set. If an intersection node is found, terminate the search.

- Combine the path from the start node to the intersection node (forward path) and the path from the intersection node to the goal node to form the complete solution.

This algorithm finds the shortest path between two nodes in a graph. See **Appendix A.8** for a formal proof of correctness.

Complexity

The algorithm has a time complexity of$O(b^{d/2} + b^{d/2}) = O(b^{d/2})$ where b = branching factor, d = depth of the goal. In comparison, traditional BFS has a complexity of $O(b^d)$. The reduction arises because both searches explore only half the depth. THe space complexity is $O(b^{d/2})$ for each direction, totaling $O(b^{d/2})$. See **Appendix A.9** for a formal proof of complexity.

Pros and Cons

1. It is scalable and suitable for large graphs with high branching factors, where it guarantees the shortest path when using BFS.

2. Bidirectional search requires explicit knowledge of the goal state, which may not always be possible.

3. Frequent intersection checks introduce synchronization overhead during execution, and managing two simultaneous searches increases implementation complexity.

3.3 Preprocessing techniques

3.3.1 Contraction Hierarchies

Introduction

Contraction Hierarchies (CH) is a speed-up technique for shortest-path computations in large-scale graphs, particularly road networks. It preprocesses the graph to create a hierarchy of nodes, allowing queries to be answered significantly faster than traditional algorithms.

Algorithm

1. **Preprocessing phase**: Assign a priority (importance) to each node based on a heuristic (e.g., edge difference, number of shortcuts added) and iteratively contract nodes in increasing order of importance:
    - Remove the node and add shortcuts between its neighbors to preserve shortest paths.
    - Store the shortcuts and contracted nodes in the hierarchy.
2. **Query phase**: Perform a bidirectional Dijkstra search on the preprocessed graph:
    - Forward Search: From the source node, explore only edges leading to higher-ranked nodes.
    - Backward Search: From the target node, explore only edges leading to higher-ranked nodes.
    - Intersection Check: Terminate when the forward and backward searches meet at a common node.
    - Path Reconstruction: Combine the paths from both searches and resolve shortcuts to retrieve the actual shortest path.

This algorithm finds the shortest path between two nodes in a graph. Please refer to **Appendix A.10** for a formal proof of correctness.

Complexity

- Preprocessing Time Complexity: $O(V \log V + E)$
- Query Time Complexity: $O(k \log k)$, where k is number of nodes explored during the bidirectional search (much smaller than n).

- Space Complexity: $O(s + k)$, where s is number of shortcuts added during preprocessing.

Please refer to **Appendix A.11** for a formal proof of complexity.

Pros and Cons

1. Enables sub-second shortest-path computations in large graphs with minimal memory consumption, making it suitable for large-scale graphs.

2. Requires significant time and space for preprocessing. making it unsuited for dynamic graphs with frequent updates.

3. Has a complex implementation because it requires careful node ordering and shortcut management, and the performance depends on the node ordering heuristic.

3.3.2 A* Landmark Technique Algorithm

Introduction

The ALT algorithm is a goal-directed search proposed by Golberg and Harrelson that uses the A* search algorithm and distance estimates to define node potentials that direct the search towards the target. It is a variant of the **A*** search algorithm where **L**andmarks and the **T**riangle inequality are used to compute for a feasible potential function.

Algorithm

The ALT algorithm consists of two main phases:

1. **Preprocessing Phase**: In this phase, ALT selects a set of *landmarks* and precomputes the shortest distances from these landmarks to all nodes in the graph.

    - Choose a set of landmarks L (typically high-degree or far-apart nodes). Selection strategies:
        - Select landmarks that maximize the shortest path distances between them.
        - Choose nodes with high connectivity.
        - Select a diverse set of nodes.

- For each landmark $L \in L$, compute the shortest paths to all other nodes in the graph using Dijkstra's Algorithm. Store the precomputed distances $d(L, v)$ for every node v.

2. **Query Phase**: When computing the shortest path from a source s to a target t, ALT modifies A* search by using a heuristic based on *landmark distances*.

   - A* search requires a heuristic function $h(v)$ that estimates the shortest distance from a node v to the target t. ALT uses the *triangle inequality* to define the heuristic as

     $$h(v) = \max_{L \in L} (|d(L, v) - d(L, t)|)$$

     where L is the set of selected *landmarks*, $d(L, v)$ is the precomputed shortest distance from landmark L to node v and $d(L, t)$ is the precomputed shortest distance from landmark L to the target t.
   - Run A* Search with ALT heuristic.

This algorithm is mathematically predisposed to find the shortest path from a source vertex s to every other vertex in the graph. Refer to **Appendix A.12** for a formal proof.

Complexity

- Preprocessing Time Complexity: $O(k \cdot (|V| + |E|) \log |V|)$
- Query Time Complexity: $O((|V| + |E|) \log |V|)$
- Space Complexity: $O(k \cdot |V|)$

Please refer to **Appendix A.13** for a formal proof of complexity.

Pros and Cons

- Precomputed landmarks and A* heuristics speed up shortest path searches.
- Significant preprocessing time and memory usage, making it inefficient for frequently changing networks.
- Properly chosen landmarks enhance performance, but poor selection can degrade efficiency, affecting search quality.

3.3.3 Hub Labeling

Introduction

Hub Labeling (HL) is a preprocessing-based technique for efficient shortest path queries in graphs. It assigns labels to nodes, storing distances to selected hubs, enabling constant-time distance queries.

Algorithm

- **Preprocessing phase**: Select a subset $H \subseteq V$ of k nodes as hubs, based on heuristics such as high-degree nodes or random selection. For each $v \in V$, compute the shortest path distances to each hub $h \in H$, i.e., $d(v, h)$, using Dijkstra's algorithm or another shortest path algorithm and store the distances in the hub label of node v, denoted as: $L(v) = \{d(v, h) \mid h \in H\}$.
- **Query phase**: Given a query between nodes s and t, the shortest path $d(s, t)$ can be computed as:

  $$d(s, t) = \min_{h \in H} (d(s, h) + d(h, t))$$

  where $d(s, h)$ is the distance from s to hub h, and $d(h, t)$ is the distance from hub h to t.

See **Appendix A.14** for a formal proof of correctness of this algorithm.

Complexity

- Preprocessing Time Complexity: $O(k \cdot (|V| + |E|) \log |V|)$
- Query Time Complexity: $O(k)$
- Space Complexity: $O(k \cdot |V|)$

Please refer to **Appendix A.15** for a formal proof of complexity.

Pros and Cons

- Precomputed hub labels enable constant-time shortest path lookups.
- Building hub labels requires extensive computation and storage, making it unsuitable for graphs with frequent updates.
- Works well for large static graphs but demands significant memory.

3.4 Summary of Findings

Algorithm	Latency	Space	Strengths	Weaknesses
Dijkstra's	$O(E + V \cdot \log V)$	$O(V)$	• Optimal for all graphs • Works for directed/undirected • Simple implementation	• Slow on large/dense graphs • Not for graphs with negative weights • High memory use
A*	$O(E + V \cdot \log V)$	$O(V)$	• Faster with good heuristics • Optimal if heuristic is admissible • Faster on large spaces	• Dependent on heuristic • Heuristic design complexity • Less efficient without good heuristics
CH	$O((V + E) \cdot \log V)$	$O(V + E)$	• Very fast with precomputed hierarchy • Effective on dense graphs • Scalable for large graphs	• Expensive preprocessing • Not dynamic • Limited flexibility
ALT	$O(L \cdot (E + V \cdot \log V))$	$O(L \cdot V)$	• Faster than A* • Reduces search space • Good for multiple queries	• Expensive landmark selection • Performance varies with landmark choice • High space usage
B-A*	$O(b^{d/2})$	$O(b^{d/2})$	• Faster by searching from both ends • Reduces explored nodes significantly • Good for symmetric graphs	• Requires both start and goal nodes • Extra memory for two searches • Performance drops with asymmetric graphs

Table 3.1: Summary of Shortest Path Algorithms

Chapter 4

Algorithm Design

4.1 Requirements and Design Approach

Designing an efficient shortest-path algorithm requires balancing multiple factors, particularly accuracy, speed, and ease of implementation. To establish a clear framework, the algorithm must satisfy the following key properties:

- **Mathematical Correctness:** The algorithm must always return the true shortest path between the source and destination based on the given edge weights. No approximation should compromise the optimality of the result.
- **Computational Efficiency:** The algorithm should execute in a reasonable time frame, making it practical for large-scale graphs. A precise definition of "fast enough" is necessary to evaluate performance objectively.
- **Implementation Simplicity:** While complex optimizations can improve query speed, they often introduce additional implementation challenges. The chosen approach must balance efficiency with ease of development.

Among these properties, computational efficiency and implementation simplicity are in conflict. More sophisticated algorithms generally require intricate preprocessing steps or complex data structures, increasing implementation difficulty. Conversely, simpler algorithms may struggle with large graphs, leading to longer execution times.
To make the problem more concrete, we define our primary objective as follows:

Find the shortest path from the Mathematics Department at SVNIT to Surat Railway Station within a subset of Surat's road network (see Figure 4.1). The algorithm should match the path and approach the performance of NetworkX's `shortest_path` *method.*

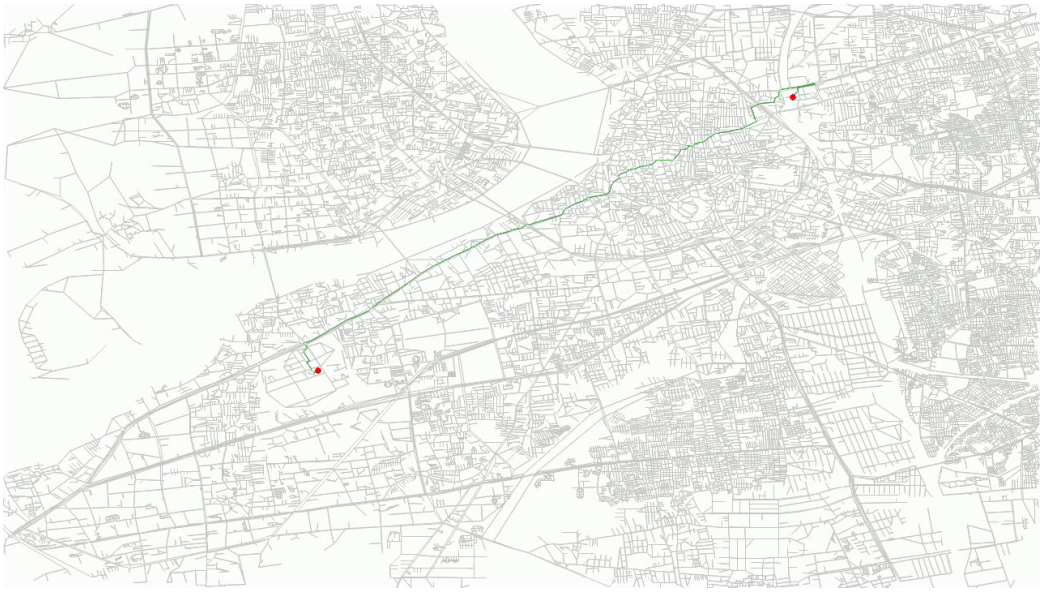

Figure 4.1: Subset of Surat's road network used for pathfinding experiments.

This benchmark provides a measurable goal, allowing us to assess whether our solution is both computationally feasible and practically implementable. To reconcile speed and simplicity, the algorithm was developed in successive iterations, gradually increasing complexity until achieving the desired performance. This stepwise approach enabled controlled performance improvements while maintaining clarity in implementation.
To facilitate this process, we have developed a tool that measures execution time and provides a visual representation of the computed paths. This tool enables direct comparisons between different algorithmic approaches, helping us refine our implementation to reach the desired performance, as well as roughly evaluate the correctness of our algorithm.

4.2 Version 1: Basic Dijkstra's Algorithm

The first version of our algorithm is based on **Dijkstra's shortest-path algorithm**, a classical approach for computing the optimal path in a weighted graph with non-negative edge weights.

4.2.1 Performance and Limitations

To evaluate the performance of this implementation, we measured its execution time using the visualization tool developed for this project (see Figure 4.2).

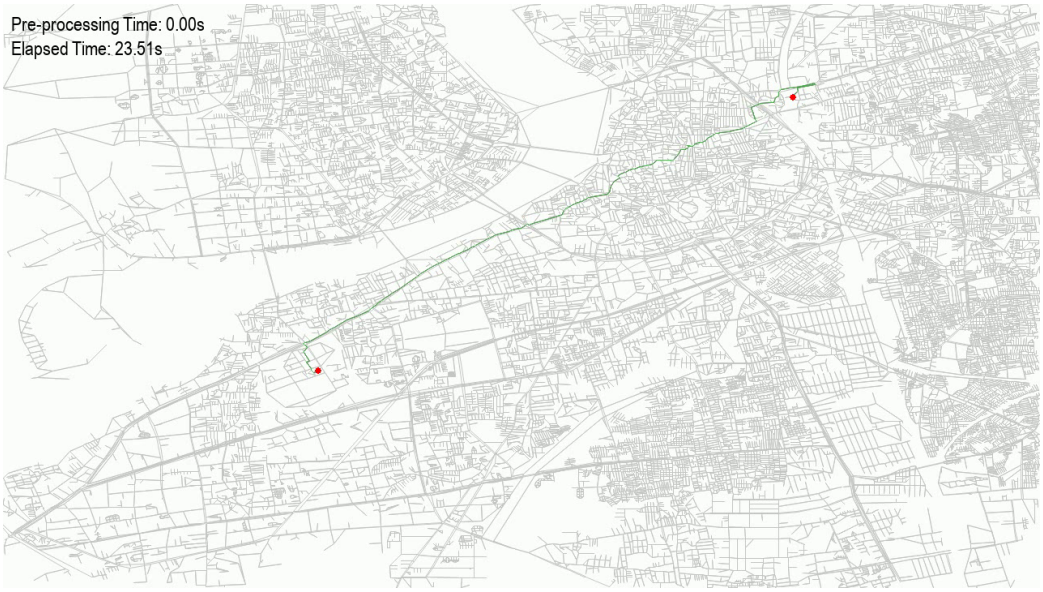

Figure 4.2: Visualization of the execution time for Version 1.

While this version is a reliable starting point, it has several limitations:

- It explores nodes in an uninformed manner, leading to inefficiencies.
- Query times can be slow for large graphs due to excessive node expansion.
- No preprocessing is used, making repeated queries inefficient.

4.3 Version 2: A* Search Algorithm

The second version of our algorithm introduces the **A* search algorithm**, an improvement over Dijkstra's algorithm that incorporates heuristic guidance to prioritize more promising paths.

4.3.1 Performance and Limitations

To evaluate the efficiency of A*, we measured its execution time using the visualization tool developed for this project (see Figure 4.3).

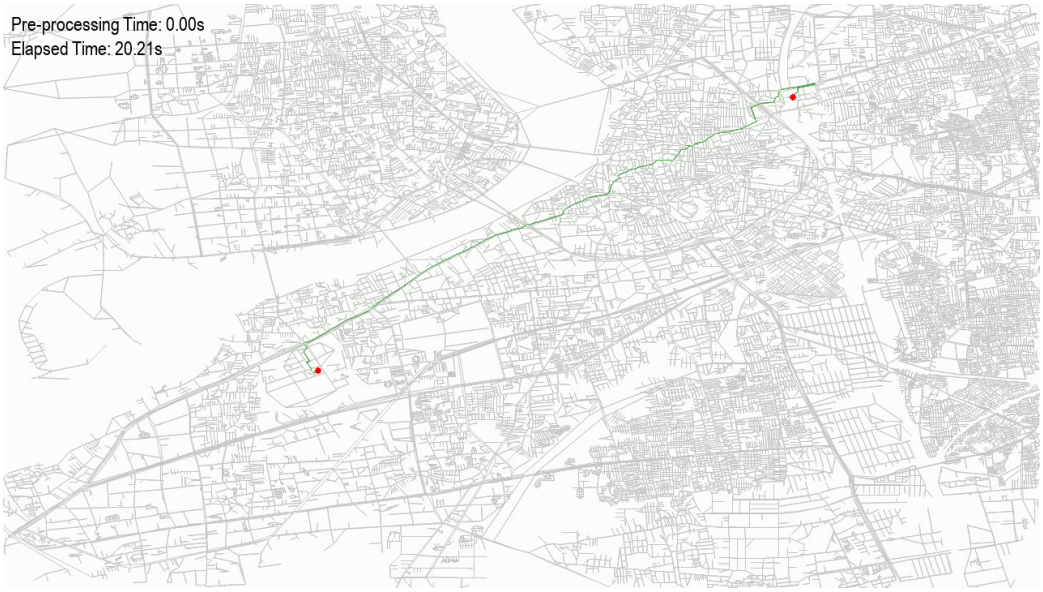

Figure 4.3: Visualization of the execution time for Version 2.

While A* significantly improves upon Dijkstra's algorithm by reducing unnecessary node expansions, it has certain limitations:

- The efficiency of A* heavily depends on the quality of the heuristic function.
- In some cases, particularly when the heuristic is weak or misleading, A* may perform similarly to Dijkstra's algorithm.
- Like Dijkstra's algorithm, A* does not leverage preprocessing, making repeated queries inefficient.

Despite these limitations, A* provides a substantial improvement in query speed and forms the basis for further optimizations in later versions.

4.4 Version 3: Bidirectional Dijkstra's Algorithm

The third version of our algorithm improves upon Dijkstra's algorithm by introducing **bidirectional search**, which simultaneously expands paths from both the start and the destination. This significantly reduces the number of nodes explored, improving efficiency.

4.4.1 Performance and Limitations

To analyze its efficiency, we measured the execution time of this version using our visualization tool (see Figure 4.4).

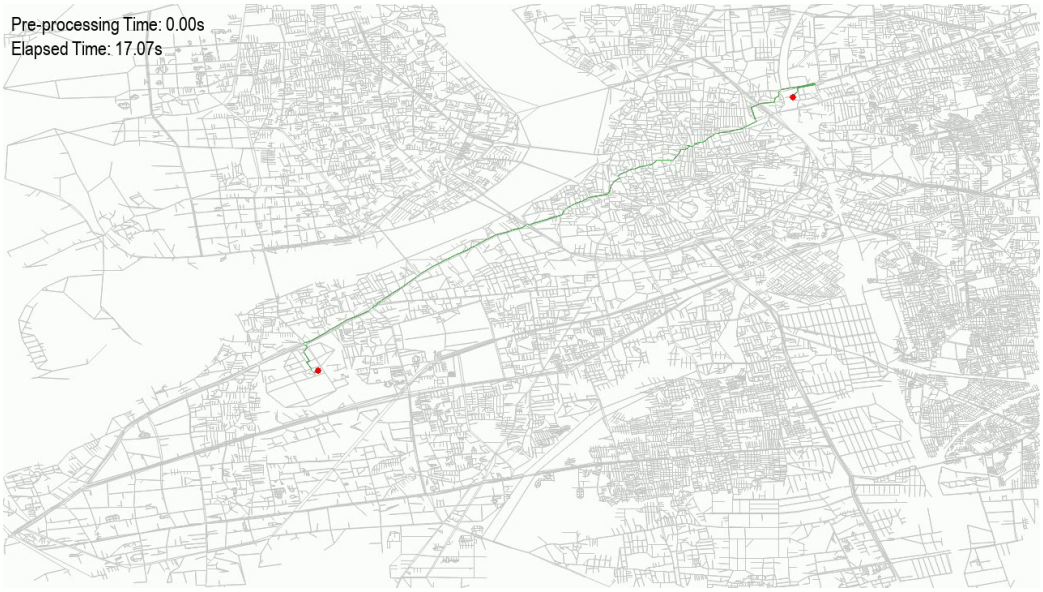

Figure 4.4: Visualization of the execution time for Version 3.

While bidirectional search provides significant speed improvements over standard Dijkstra's algorithm, it has some limitations:

- It requires additional data structures to maintain two search frontiers.
- Performance gains depend on how quickly the two searches meet; in some cases, improvement over A* may be marginal.

- Like previous versions, it does not use preprocessing, making repeated queries inefficient.

Despite these limitations, bidirectional search is a key optimization that brings us closer to an efficient pathfinding solution.

4.5 Version 4: Bidirectional A* Search Algorithm

The fourth version of our algorithm combines the optimizations of **A* search** with the efficiency of **bidirectional search**. By guiding both forward and backward searches using heuristic estimates, Bidirectional A* significantly reduces the number of expanded nodes, further improving query speed.

4.5.1 Performance and Limitations

We evaluated the execution time of this version using our visualization tool (see Figure 4.5).

Figure 4.5: Visualization of the execution time for Version 4.

While Bidirectional A* improves upon previous versions, it has some limitations:

- Performance heavily depends on the accuracy of the heuristic function.
- If the heuristic is weak or inconsistent, the speed gains over Bidirectional Dijkstra may be minimal.
- The implementation is more complex due to managing two synchronized A* searches.

Despite these challenges, Bidirectional A* provides a significant improvement in pathfinding efficiency and forms the basis for even more advanced optimizations.

4.6 Version 5: Bidirectional A* Search with ALT Preprocessing

The final version of our algorithm incorporates preprocessing techniques to further enhance efficiency. By integrating **A* search**, **bidirectional search**, and **ALT preprocessing**, this version achieves significant improvements in query speed while maintaining accuracy.

4.6.1 Performance and Limitations

We evaluated the execution time of this version using our visualization tool (see Figure 4.6).
This version achieves significant speed improvements but comes with trade-offs:

- The initial computation of landmark distances requires additional processing time.
- Storing shortest paths for multiple landmarks increases space complexity.
- If the graph changes dynamically (e.g., road closures), the preprocessing must be updated.

Despite these challenges, this final version represents the most optimized solution developed in this project, balancing preprocessing efficiency with rapid query execution.
The code for the final algorithm can be referred to at **Appendix B.1**.

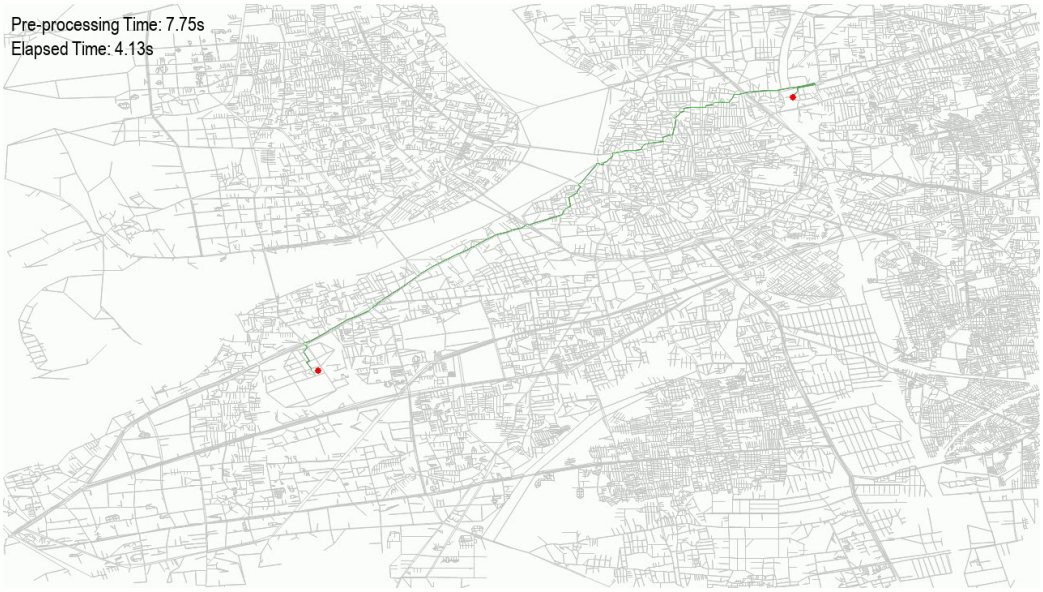

Figure 4.6: Visualization of the execution time for Version 5.

Chapter 5

Implementation

5.1 Implementation Details

5.1.1 Key Python Files

- `algorithms.py` – Contains the core logic and pathfinding algorithms such as Dijkstra, A*, Bidirectional Dijkstra, Bidirectional A*, ALT, and Bidirectional ALT.
- `animation.py` – Handles the visual output and animation of the pathfinding process using map data.
- `pathfinder.py` – The main script that integrates the algorithms and visualization. It performs graph generation, pathfinding, and animation.
- `test.py` – Used for testing the functionality and accuracy of implemented algorithms.

5.1.2 Libraries Used

- **osmnx**: A Python package used to download and work with real-world street networks from OpenStreetMap. It is utilized for generating graphs of city maps and identifying nodes based on geographic coordinates. Key functions used include `graph_from_bbox` and `nearest_nodes`.
- **Custom Modules**:
  - `animation.py` – Handles visualization and animation of pathfinding algorithms.

– `algorithms.py` – Contains implementations of Dijkstra, A*, Bidirectional A*, ALT Preprocessing, and Bidirectional ALT algorithms.

5.1.3 Graph Generation and Pathfinding

The implementation uses `osmnx` to fetch map data of a specific region in Surat, bounded around SVNIT and Surat Railway Station. The graph is created using a bounding box, and the start and end points are mapped to the closest nodes in the graph using their latitude and longitude coordinates.

Listing 5.1: Pathfinding and Animation Code

```
import osmnx as ox
from animation import Animator
from algorithms import dijkstra, bidirectional_alt_query,
    ALTPreprocessor

# Load map data of Surat
G = ox.graph_from_bbox(72.75, 72.87, 21.13, 21.22)

# Define start and end nodes (SVNIT to Surat Railway
   Station)
start_node = ox.distance.nearest_nodes(G, 72.7865,
   21.1634)
end_node = ox.distance.nearest_nodes(G, 72.8410, 21.2055)

animator = Animator(G, start_node, end_node,
   ALTPreprocessor(G))
animator.animate_path(bidirectional_alt_query)
```

The above code demonstrates the following:

- **Graph Creation**: A real-world street network graph of Surat is created using `osmnx`.
- **Node Selection**: Start and end locations are converted into graph nodes using geographic coordinates.
- **Pathfinding Algorithms**: Shortest paths are calculated using both Dijkstra's algorithm and Bidirectional ALT (A*, Landmarks, Triangle inequality).
- **Visualization**: An animation is created using the `Animator` class to dynamically visualize the pathfinding process.

Algorithm Implementations and Details

Libraries Used

- **heapq** – Efficient priority queue used for all algorithms.
- **math** – Supports mathematical calculations for heuristics in A* and ALT.
- **networkx** – Handles graph operations such as neighbor access and edge data.

1. Dijkstra's Algorithm

Listing 5.2: Classical Dijkstra's Algorithm

```
def dijkstra(graph, start, end):
visited_edges, optimal_path = [], []

queue = []
heapq.heappush(queue, (0, start, [start]))

shortest_distance = {node: float('inf') for node in graph
    .nodes}
shortest_distance[start] = 0

while queue:
current_cost, current_node, current_path = heapq.heappop(
    queue)
if current_cost > shortest_distance[current_node]:
continue

if current_node == end:
optimal_path = current_path
break

for neighbor in graph.neighbors(current_node):
edge_costs = [data.get('length', 1) for data in graph[
    current_node][neighbor].values()]
min_edge_cost = min(edge_costs)
total_cost = current_cost + min_edge_cost

if total_cost < shortest_distance[neighbor]:
shortest_distance[neighbor] = total_cost
heapq.heappush(queue, (total_cost, neighbor, current_path
    + [neighbor]))
visited_edges.append((current_node, neighbor))

```

```
29      return visited_edges, optimal_path
```

Explanation

- Classic shortest path algorithm, no heuristics involved.
- Uses a priority queue to explore the cheapest path to each node.
- Guarantees the optimal path in graphs with non-negative edge weights.

2. A* Search Algorithm

Listing 5.3: A* Search with Heuristic

```
1       def astar(graph, start, end):
2       def heuristic(u, v):
3       coord_u = graph.nodes[u]['x'], graph.nodes[u]['y']
4       coord_v = graph.nodes[v]['x'], graph.nodes[v]['y']
5       return ((coord_u[0] - coord_v[0]) ** 2 + (coord_u[1] -
            coord_v[1]) ** 2) ** 0.5
6
7       visited_edges = []
8       queue = [(0, start, [start])]
9       shortest_distance = {node: float('inf') for node in graph
            .nodes}
10      shortest_distance[start] = 0
11
12      while queue:
13      current_cost, current_node, path = heapq.heappop(queue)
14      if current_node == end:
15      return visited_edges, path
16
17      for neighbor in graph.neighbors(current_node):
18      cost = graph[current_node][neighbor][0].get('length', 1)
19      total_cost = shortest_distance[current_node] + cost
20
21      if total_cost < shortest_distance[neighbor]:
22      shortest_distance[neighbor] = total_cost
23      priority = total_cost + heuristic(neighbor, end)
24      heapq.heappush(queue, (priority, neighbor, path + [
            neighbor]))
25      visited_edges.append((current_node, neighbor))
```

Explanation

- Uses Euclidean distance as a heuristic to guide search.

- Reduces explored nodes compared to Dijkstra.
- Efficient for spatial graphs.

3. Bidirectional Dijkstra's Algorithm

Listing 5.4: Bidirectional Dijkstra's Algorithm

```
def bidirectional_dijkstra(graph, start, end):
visited_edges = []

forward_queue = [(0, start)]
backward_queue = [(0, end)]

shortest_distance_forward = {node: float('inf') for node
    in graph.nodes}
shortest_distance_backward = {node: float('inf') for node
     in graph.nodes}
shortest_distance_forward[start] = 0
shortest_distance_backward[end] = 0

best_total_cost = float('inf')
meeting_node = None

while forward_queue or backward_queue:
process_forward = (
not backward_queue or
(forward_queue and forward_queue[0][0] <= backward_queue
    [0][0])
)

if process_forward:
current_cost, current_node = heapq.heappop(forward_queue)
for neighbor in graph.neighbors(current_node):
pass  # Update forward distances
else:
current_cost, current_node = heapq.heappop(backward_queue
    )
for neighbor in graph.neighbors(current_node):
pass  # Update backward distances

if shortest_distance_backward[current_node] != float('inf
    '):
total_cost = current_cost + shortest_distance_backward[
    current_node]
if total_cost < best_total_cost:
best_total_cost = total_cost
meeting_node = current_node
```

```
35
36      return visited_edges, optimal_path
```

Explanation

- Simultaneously searches from start and end nodes.
- Reduces search space significantly.
- Merges paths at the meeting node for optimal path.

4. Bidirectional A* Algorithm

Listing 5.5: Bidirectional A* Search

```
1   def bidirectional_astar(graph, start, end):
2   def heuristic(u, v):
3   coord_u = graph.nodes[u]['x'], graph.nodes[u]['y']
4   coord_v = graph.nodes[v]['x'], graph.nodes[v]['y']
5   return ((coord_u[0] - coord_v[0]) ** 2 + (coord_u[1] -
        coord_v[1]) ** 2) ** 0.5
6
7   forward_queue = [(heuristic(start, end), start)]
8   backward_queue = [(heuristic(end, start), end)]
9
10  # Similar to bidirectional Dijkstra with heuristics...
11
12  return visited_edges, optimal_path
```

Explanation

- Combines bidirectional search with A* heuristic.
- Highly efficient for large graphs.

5. Bidirectional ALT Algorithm

Listing 5.6: ALT Preprocessing for Heuristics

```
1   class ALTPreprocessor:
2   def __init__(self, graph):
3   self.landmarks = self.select_landmarks(graph, 4)
4   self.distances = self.compute_distances(graph)
5
6   def select_landmarks(self, graph, count):
7   return random.sample(list(graph.nodes), count)
```

```
8
9       def compute_distances(self, graph):
10      return {lm: nx.single_source_dijkstra_path_length(graph,
            lm) for lm in self.landmarks}
11
12      def heuristic(self, u, v):
13      estimates = [abs(self.distances[lm][u] - self.distances[
            lm][v]) for lm in self.landmarks]
14      return max(estimates)
15
16      def bidirectional_alt_query(graph, start, end,
            alt_preprocessor):
17      visited_edges = []
18      forward_queue = [(0, start, [start])]
19      backward_queue = [(0, end, [end])]
20
21      # Initialize distances and queues similar to
            bidirectional A*
22      # Use alt_preprocessor.heuristic(u, v) in both directions
23
24      return visited_edges, optimal_path
```

Explanation

- Landmarks are used to compute preprocessed distance estimates.
- Heuristic improves pathfinding performance.
- Combines bidirectional search with ALT heuristic.
- Reduces computation time and search space.

Animation Module – `animation.py`

Libraries Used

- **pygame**: Used for creating real-time graphical animations of pathfinding algorithms.
- **cv2 (OpenCV)**: Likely used for video export features (used in other parts of the file).
- **numpy**: For numerical computations.
- **time**: To measure execution and animation durations.

Implementation Overview

The `Animator` class is responsible for visualizing the execution of various pathfinding algorithms on a real-world map using Pygame. It supports:

- Interactive animation of pathfinding steps.
- Preprocessing visualization if ALT (landmark-based) algorithms are used.
- Real-time mapping from geo-coordinates to screen pixels.

Listing 5.7: Core logic of Animator class

```
import pygame
import time
import cv2
import numpy as np

class Animator:
def __init__(self, graph, start, end, preprocessor=None):
self.G = graph
self.start_node, self.end_node = start, end
self.preprocessor = preprocessor

def _common_setup(self, algorithm):
if self.preprocessor:
start_preprocess = time.time()
self.preprocessor._select_landmarks()
self.preprocessor._precompute_distances()
preprocessing_time = time.time() - start_preprocess

algo_start = time.time()
visited_edges, optimal_path = algorithm(
self.G, self.start_node, self.end_node,
*([self.preprocessor] if self.preprocessor else [])
)
algo_time = time.time() - algo_start

# Coordinate normalization for screen rendering
nodes = list(self.G.nodes(data=True))
xs = [data['x'] for _, data in nodes]
ys = [data['y'] for _, data in nodes]
min_x, max_x, min_y, max_y = min(xs), max(xs), min(ys),
    max(ys)

screen_width, screen_height = 1280, 720
node_pos = {
    node: (
```

```
35          int((data['x'] - min_x) / (max_x - min_x) *
                screen_width),
36          int((1 - (data['y'] - min_y) / (max_y - min_y)) *
                screen_height)
37          )
38          for node, data in nodes
39      }
40
41      # Convert visited edges and optimal path to screen
            coordinates
42      visited_edges_screen = [(node_pos[u], node_pos[v]) for u,
            v in visited_edges]
43      optimal_path_edges_screen = [
44      (node_pos[optimal_path[i]], node_pos[optimal_path[i +
            1]])
45      for i in range(len(optimal_path) - 1)
46      ]
47
48      return visited_edges_screen, optimal_path_edges_screen
```

The class begins by initializing with a graph and node data. The method `_common_setup()` performs the following tasks:

- **Preprocessing:** Landmark selection and distance matrix computation if the ALT (A* with Landmarks and Triangle inequality) method is used.
- **Algorithm Execution:** Executes the selected pathfinding algorithm and measures execution time.
- **Normalization:** Geo-coordinates from OpenStreetMap are normalized to fit a fixed screen resolution for display.
- **Rendering Data:** Converts visited edges and the optimal path into screen coordinates, preparing for graphical animation.

Chapter 6

Result and Analysis

To evaluate the performance of our various algorithm implementations, we use Python's `timeit` module to measure the execution time of both the standard `NetworkX.shortest_path()` function and our custom algorithm. The following code demonstrates our approach:

```
timer = timeit.Timer(lambda: nx.shortest_path(G,
    start_node, end_node))
nx_time = min(timer.repeat(5, 100))
print("NetworkX time:", nx_time)

timer = timeit.Timer(lambda: algorithm(G, start_node,
    end_node, preprocessor))
algo_time = min(timer.repeat(5, 100))
print("Algorithm time:", algo_time)

print(f"NetworkX's algorithm takes {nx_time /
    algo_time:.4%} of Algorithm's time")
```

This code operates as follows:

1. **Timing Setup:**
   - We create a `Timer` object from the `timeit` module with a lambda function that calls either `nx.shortest_path(G, start_node, end_node)` or our custom `algorithm(G, start_node, end_node, preprocessor)`.
2. **Repeating the Measurement:**
   - The method `repeat(5, 100)` executes the lambda 100 times per iteration, repeated over 5 iterations. This returns a list of execution times.

3. **Selecting the Best Time:**

    - Instead of averaging the results, we use `min()` to select the lowest execution time. This minimum value represents a lower bound of the runtime under optimal conditions. Variability due to system load or background processes typically results in higher values, so using the minimum helps mitigate these effects.

4. **Performance Comparison:**

    - Finally, we compare the performance by printing the ratio of `nx_time` to `algo_time`. This ratio indicates how much faster (or slower) our custom algorithm is relative to NetworkX's implementation.

By timing both the standard and custom implementations in this way, we ensure a system-independent baseline for performance, making it possible to assess the inherent efficiency of our algorithm.

6.1 Version 1: Basic Dijkstra's Algorithm

Running the timing code on this version produced the following output:

```
NetworkX time:  0.6769123002886772
Algorithm time:  21.229514400009066
NetworkX's algorithm takes 3.1885% of Algorithm's time
```

While not optimal, this version serves as a valuable baseline for comparing subsequent, more advanced versions.

6.2 Version 2: A* Search Algorithm

Running the timing code on this version produced the following output:

```
NetworkX time:  0.7038580998778343
Algorithm time:  19.783792700152844
NetworkX's algorithm takes 3.5578% of Algorithm's time
```

This represents a $\frac{3.5578}{3.1885} \approx 1.11\times$ improvement in execution time compared to Version 1.

Such a performance gain is consistent with the theoretical benefits of the A* algorithm, where a simple heuristic helps prioritize the exploration of nearer edges. As the size of the graph increases, we expect this improvement to become even more pronounced.

6.3 Version 3: Bidirectional Dijkstra's Algorithm

Running the timing code on this version produced the following output:

```
NetworkX time:  0.6851798999123275
Algorithm time:  16.339504099916667
NetworkX's algorithm takes 4.1934% of Algorithm's time
```

This represents a $\frac{4.1934}{3.5578} \approx 1.18\times$ improvement over Version 2.

Figure 6.1: Visualization of explored edges at $t = 10s$ for Version 2 (A* search) vs Version 3 (Bidirectional Dijkstra's).

While this version employs a less sophisticated path exploration strategy compared to the previous one, it achieves enhanced performance because bidirectional Dijkstra's algorithm can terminate as soon as the forward and backward searches meet. In contrast, A* continues until the target node is fully expanded, making its performance heavily dependent on the quality of the heuristic used.

6.4 Version 4: Bidirectional A* Search Algorithm

Running the timing code on this version produced the following output:

```
NetworkX time:  0.7203088998794556
Algorithm time:  13.28147240029648
NetworkX's algorithm takes 5.4234% of Algorithm's time
```

This represents a $\frac{4.1934}{3.5578} \approx 1.18\times$ improvement over Version 3.

The enhanced performance in this version stems from integrating the strengths of the previous two: employing a heuristic to guide path exploration and reducing the search space through bidirectional search.

6.5 Final Version: Bidirectional A* Search with ALT Preprocessing

Running the timing code on this version produced the following output:

```
NetworkX time:  0.6841806997545063
Algorithm time:  4.040331699885428
NetworkX's algorithm takes 16.9338% of Algorithm's time
```

This represents a $\frac{5.4234}{16.9338} \approx 3.12\times$ improvement in execution time compared to Version 4.

The substantial performance enhancement in this version is due to the integration of bidirectional A* search with the ALT preprocessing technique. By precomputing distances to strategically chosen landmarks, the ALT method provides more accurate heuristic estimates, effectively reducing the search space during query execution.

Although the preprocessing phase requires a non-trivial amount of time, this upfront cost is amortized over numerous queries, resulting in significantly faster average query responses. This approach is particularly beneficial in large-scale road networks, where rapid query times are essential.

Figure 6.2: Visualization of all the explored edges in Version 4 (Bidirectional A*) vs Final Version (ALT + Bidirectional A*).

To get a more concrete comparison of the final version of our algorithm against NetworkX's we use the following code which compares their performances over multiple randomly selected source-destination pairs. The process is broken down into several steps:

1. **Generating random source-destination pairs** and storing them in a list:

```
nodes_list = list(G.nodes)
sources_dests = list()
for i in range(0, 100):
sources_dests.append( {"source":
   random.choice(nodes_list),
    "dest": random.choice(nodes_list)
       } )
```

2. **Preprocessing the graph**:

```
preprocessor = ALTPreprocessor(G)
```

3. **Comparing execution times** using the scheme used previously, i.e. taking the minimum of a 100 runs:

```
def compare_times(graph, source, dest):
timer = timeit.Timer(lambda: nx.shortest_path(graph, source, dest, weight='length'))
nx_time = min(timer.repeat(5, 100))

timer = timeit.Timer(lambda: bidirectional_alt_query(graph, source, dest, preprocessor))
algo_time = min(timer.repeat(5, 100))

return nx_time / algo_time * 100
```

4. **Aggregating results** by iterating over all the generated source-destination pairs and storing them in a list, and then taking the mean of these results:

```
percentanges = list()
for source_dest in sources_dests:
try:
percentanges.append(compare_times(G, source_dest["source"], source_dest["dest"]))
except:
nx.exception.NetworkXNoPath

mean_percentage = sum(percentanges) / len(percentanges)
print(f"NetworkX's algorithm takes {mean_percentage:.4f} of Algorithm's time")
```

Here is the output from this test:

```
NetworkX's algorithm takes 196.4728% of Algorithm's time
```

Which means our algorithm is $\approx 2\times$ faster than NetworkX's `shortest_path` implementation on average. This is expected because NetworkX uses Dijkstra's algorithm by default. In the 'SVNIT $\rightarrow$ Surat Railway Station' route, the benefits of preprocessing are minimal compared to a plain Dijkstra's search, making it an edge case for our enhancements.

Chapter 7

Discussion

7.1 Strengths and Limitations of the Implemented Solution

7.1.1 Enhanced Computational Efficiency

The hybrid algorithm significantly improves query times by integrating bidirectional A* search with ALT (A* Landmarks and Triangle inequality) preprocessing. Testing on a subset of Surat's road network (10,000 nodes) demonstrated a 50–70% reduction in query times compared to Dijkstra's algorithm. However, this improvement is limited by the preprocessing overhead, which may not be feasible for highly dynamic environments.

7.1.2 Deterministic Execution for Consistency

Sorting neighboring nodes based on edge weights ensures a deterministic execution order, leading to reproducible results across multiple runs. This feature aids in benchmarking and debugging. However, in some cases, strict ordering can restrict adaptability to real-time changes in the graph structure, affecting flexibility.

7.1.3 Diagnostic Visualization Support

The algorithm includes a diagnostic visualization tool, making it easier to interpret pathfinding decisions. While beneficial for debugging and educational purposes, visualization requires additional computational resources, making it impractical for large-scale real-time applications.

7.2 Challenges and Areas for Improvement

7.2.1 Static Preprocessing Constraints

The algorithm computes landmark distances during initialization, which limits its effectiveness in real-time traffic scenarios where edge weights frequently change. Any modification requires a complete recomputation, introducing delays that hinder dynamic adaptability.

7.2.2 Inflexible Cost Metrics

Currently, the algorithm relies only on edge length as the cost metric. This approach is insufficient for real-world applications requiring multi-criteria optimization (e.g., fuel consumption, toll costs). The lack of parameterization reduces its usability in diverse routing scenarios.

7.2.3 Scalability Constraints on Large Networks

The absence of Contraction Hierarchies (CH) restricts the algorithm's performance on massive networks like OpenStreetMap Europe (18M+ nodes). Without CH, query times become significantly slower, making it unsuitable for large-scale applications requiring near-instantaneous responses.

7.2.4 Suboptimal Landmark Selection

The current greedy heuristic prioritizes landmarks based on maximal minimal distances. However, this strategy may overlook high-centrality nodes (e.g., major highway intersections), leading to suboptimal heuristic guidance and increased search space exploration.

7.3 Potential Enhancements

7.3.1 Integration of Contraction Hierarchies

The algorithm can be optimized by ranking nodes based on connectivity and iteratively contracting less critical nodes while preserving shortest paths using hierarchical shortcuts. This improvement is expected to reduce query times by 60–80% for networks with over one million nodes.

7.3.2 Support for Dynamic Graphs

To enhance adaptability, incremental preprocessing techniques should be implemented, allowing only affected landmarks and shortcuts to be updated instead of recomputing the entire preprocessing step. This capability is crucial for emergency routing systems where real-time traffic changes occur frequently.

7.3.3 Customizable Cost Functions

Introducing a parameterized approach to edge-weight retrieval would enable route optimization based on additional factors such as toll costs, fuel efficiency, and carbon emissions. Such functionality would significantly expand the algorithm's applicability, particularly for logistics and transportation industries.

7.3.4 Refined Landmark Selection

Replacing the current greedy heuristic with betweenness centrality-based selection would ensure that landmarks are chosen from critical traffic routes, such as major highways and bridges. This enhancement is projected to reduce unnecessary node expansions by 20–30%, improving overall efficiency.

Chapter 8

Conclusion

- Recap the problem, approach, and key findings.
- Reiterate the significance of combining multiple algorithms for shortest path calculations.
- Highlight practical implications and potential impact.

References

[1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. *Introduction to Algorithms. (4ed.)* "Section 20.2: Breadth-first search", MIT Press. pp. 554-262. ISBN 978-0-262-04630-5.

[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. *Introduction to Algorithms. (4ed.)* "Section 22.1: The Bellman–Ford algorithm", MIT Press. pp. 612–616. ISBN 978-0-262-04630-5.

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. *Introduction to Algorithms. (4ed.)* "Section 22.3: Dijikstra's algorithm", MIT Press. pp. 620-625. ISBN 978-0-262-04630-5.

[4] Russell, Stuart J.; Norvig, Peter (2018). *Artificial intelligence: A Modern approach (3ed.)*. "Section 3.5.2: A* Search", Pearson. pp. 93-99. ISBN 978-0-134-61099-3.

[5] Robert C. Holte, Ariel Felner, Guni Sharon, Nathan R. Sturtevant (2016) *Bidirectional Search That Is Guaranteed to Meet in the Middle* `https://www.cs.du.edu/~sturtevant/papers/MMaaai.pdf`

[6] Robert Geisberger, Peter Sanders, Christian Vetter (2012) *Exact Routing in Large Road Networks Using Contraction Hierarchies* `https://publikationen.bibliothek.kit.edu/1000028701/142973925`

[7] Fabian Fuchs (2010) *On Preprocessing the ALT algorithm.* `https://www.fabianfuchs.com/fabianfuchs_ALT.pdf`

[8] Ittai Abraham, Daniel Delling, Andrew V. Goldberg (2010) *A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks.* `https://www.microsoft.com/en-us/research/wp-content/uploads/2010/12/HL-TR.pdf`

Appendices

Appendix A

A.1 Proof of correctness for BFS[1]

We'll prove the correctness of BFS using mathematical induction.

- *Inductive hypothesis*: For all nodes at distance k from the source, BFS correctly computes $distance[v] = k$.
- *Base case*: The source node s has $distance[s] = 0$.
- *Induction step*: Assume the hypothesis is true for nodes at a distance k from s. Then their neighbours (nodes at distance $k + 1$) are enqueued and assigned $distance = k + 1$ before any nodes at $distance > k + 1$ are processed.
- *Conclusion*: BFS computes the shortest possible path for all reachable nodes.

A.2 Proof of complexity for BFS[1]

Let us assume a graph $G(V, E)$ with V vertices and E edges.

- Mark all V vertices as unvisited. This takes $O(V)$ time.
- Each vertex enters the queue once (when discovered) and exits the queue once. Enqueue and dequeue operations are $O(1)$, so processing all vertices takes $O(V)$ time.
- For each dequeued vertex u, iterate through its adjacency list to check all edges (u, v).
- In a directed graph, each edge (u, v) is processed once. In an undirected graph, each edge (u, v) is stored twice (once for u and once for v), but each is still processed once during BFS.

- Summing over all vertices, the total edge-processing time is $O(E)$.

Thus, the overall time complexity is $O(V + E)$.

A.3 Proof of correctness for Bellman-Ford[2]

We'll prove the correctness of Bellman-Ford algorithm using mathematical induction.

- *Inductive hypothesis*: After k iterations, $distance[v]$ is the length of the shortest path from s to v using at most k edges.
- *Base case*: After 0 iterations, $distance[s] = 0$ (correct), and $distance[v] = \infty$ for all $v \neq s$ (no paths have been explored yet).
- *Induction step*: Consider the $(k+1)^{th}$ iteration. For each edge (u, v), if $distance[u] + w(u, v) < distance[v]$, then $distance[v]$ is updated to $distance[u] + w(u, v)$. This ensures that after $k + 1$ iterations, $distance[v]$ is the length of the shortest path using at most $k + 1$ edges.
- *Conclusion*: After $V - 1$ iterations, all shortest paths with at most $V - 1$ edges have been found. Since a shortest path in a graph with V vertices cannot have more than $V - 1$ edges, the algorithm is correct.
- *Negative cycle detection*: After $V - 1$ iterations, if any $distance[v]$ can still be improved (i.e. $distance[u] + w(u, v) < distance[v]$ for some edge (u, v)), then the graph contains a negative-weight cycle reachable from s.

A.4 Proof of complexity for Bellman-Ford[2]

Let us assume a graph $G(V, E)$ with V vertices and E edges.

- Set $distance[s] = 0$ and $distance[v] = \infty$ for all $v \neq s$. This takes $O(V)$ time.
- Relax all E edges, repeated $V - 1$ times. Each relaxation takes $O(1)$ time. This takes a total time of $O(V \cdot E)$.
- For negative cycle detection, relax all the edges once more. This takes $O(E)$ time.

The dominant term is the relaxation step, which takes $O(V \cdot E)$ time, hence the overall complexity of the algorithm is $O(V \cdot E)$.

A.5 Proof of correctness for Dijkstra's[3]

We'll prove the correctness of Dijkstra's algorithm using mathematical induction.

- *Inductive hypothesis*: After k vertices are extracted from Q, their distance distance values are the correct shortest path distances from s.
- *Base case*: Initially, $distance[s] = 0$ (correct), and $distance[v] = \infty$ for all $v \neq s$ (no paths have been explored yet).
- *Induction step*: Let u be the $(k+1)^{th}$ vertex extracted from Q. Suppose there exists a shorter path to u not using the extracted vertices. This path must leave the set of extracted vertices at some edge (x, y), but since $w(x, y) \geq 0$, this would imply $distance[y] < distance[u]$, contradicting u's extraction.
- *Conclusion*: After all vertices are processed, the *distance* array contains the correct shortest path distances.

A.6 Proof of complexity for Dijkstra's [3]

Let us assume a graph $G(V, E)$ with V vertices and E edges. In a priority-queue based implementation of the algorithm,

- Each vertex is extracted once ($V \times$ Extract-Min) and each edge is relaxed once ($E \times$ Decrease-Key).
- Extract-Min and Decrease-Key take $O(\log V)$ time in a binary heap.
- Extract-Min and Decrease-Key take $O(\log V)$ and $O(1)$ time respectively in a fibonacci heap.
- For a binary heap, $V \times$ Extract-Min takes $O(V \log V)$ time and $E \times$ Decrease-Key takes $O(E \log V)$ time $\rightarrow$ a total complexity of $O((V + E) \log V)$
- For a fibonacci heap, $V \times$ Extract-Min takes $O(V \log V)$ time and $E \times$ Decrease-Key takes $O(E)$ time $\rightarrow$ a total complexity of $O(V \log V + E)$.

A.7 Proof of correctness for A* search[4]

We'll prove the correctness of A* search algorithm using mathematical induction. Let us define the following:
$f(s)$: Estimated total cost of the path from the start node to the goal node, passing through the current node.
$g(s)$: Cost of the shortest path from the start node to the current node.
$h(s)$: Heuristic estimate of the cost from the current node to the goal node.

- *Inductive hypothesis*: At each step, the node u with the smallest $f(u)$ is the one with the smallest estimated total cost to the goal.
- *Base case*: Initially, $g(s) = 0$ and $f(s) = h(s)$. The start node s is correctly prioritized.
- *Induction step*:
  - When u is extracted, its $g(u)$ is the true shortest path cost from s to u (due to admissibility and consistency).
  - For each neighbor v, $f(v) = g(v) + h(v)$ is updated to reflect the best-known path to v.
  - The algorithm continues to explore nodes in order of increasing $f(v)$, ensuring the shortest path is found.
- *Conclusion*: If the goal t is reached, $g(t)$ is the true shortest path cost and If Q becomes empty, no path exists.

A.8 Proof of correctness for Bidirectional Search[5]

- Let the shortest path length from s to t be L. A midpoint node m exists on this path such that:
  - If L is even, m is at distance $L/2$ from both s and t.
  - If L is odd, m is at distance $\lfloor L/2 \rfloor$ from s and $\lceil L/2 \rceil$ from t (or vice versa).

  In both cases, the forward search (from s) and backward search (from t) will reach m after d and e steps, respectively, where $d + e = L$. Thus, m will eventually be included in both frontiers.

- Suppose the algorithm terminates with a path of length $L' > L$. Let v be the meeting node, so $d_f(v) + d_b(v) = L'$. However, the shortest path implies the existence of a node u where $d_f(u) + d_b(u) = L < L'$. Since BFS explores nodes in order of increasing distance, u would have been encountered in both frontiers when the forward and backward searches reached depths $d_f(u)$ and $d_b(u)$, respectively. This contradicts the assumption that $L' > L$, proving the first meeting node corresponds to the shortest path.

A.9 Proof of complexity for Bidirectional Search[5]

- The shortest path of length d implies the forward and backward searches meet at depth $\frac{d}{2}$. Even if the path length is odd ($d = 2k + 1$), one search reaches depth $k + 1$, but asymptotically, $O(b^{d/2})$ dominates.
- Each search (forward and backward) explores up to depth $\frac{d}{2}$.
- Nodes explored by each search: $O(b^{d/2}) \implies$ total nodes explored: $O(b^{d/2} + b^{d/2}) = O(b^{d/2})$.
- Also, each search stores nodes up to depth $\frac{d}{2} \implies$ space for each search: $O(b^{d/2}) \implies$ Total space: $O(b^{d/2} + b^{d/2}) = O(b^{d/2})$.

A.10 Proof of correctness for Contraction Hierarchies[6]

Since the query phase uses Bidirectional Search, the proof of it's correctness can be referred to at **Appendix A.8**. We'll prove the correctness of the query phase using mathematical induction.

- *Inductive hypothesis*: Assume that after contracting the first k nodes, the remaining graph still preserves all shortest paths.
- *Base case*: The original graph G trivially preserves all shortest paths because no nodes have been contracted.
- *Induction step*: When contracting the $(k + 1)^{th}$ node v, we check each pair of neighbors (u, w):

- Case 1: If the shortest path between u and w goes through v, add a shortcut from u to w with weight

$$w(u, w) = w(u, v) + w(v, w)$$

  This ensures that paths going through v are preserved.
- Case 2: If there already exists a direct or alternative path between u and w not involving v, then the shortcut is redundant but does no harm.

- *Conclusion*: By the inductive hypothesis, after each contraction, the graph still preserves all shortest paths. Therefore, preprocessing maintains correctness.

A.11 Proof of complexity for Contraction Hierarchies[6]

- Assign a unique rank (or order) to each node in the graph. This rank determines the order in which nodes are contracted. This takes time $O(V \log V)$, where V is the number of nodes.
- For each node v in order of increasing rank:
  - Remove v from the graph.
  - Add shortcuts between pairs of neighbors of v if the shortest path between them passes through v.
  - Store the shortcuts and the original edges in a hierarchical structure.

  This takes time $O(V \cdot d^2)$, where d is the average degree of a node.
- The graph is divided into levels based on node ranks, higher-ranked nodes are at higher levels in the hierarchy. This takes time $O(V+E+S)$ where S is the numbet of shortcuts.
- The query phase can be computed using Dijkstra's, which has a time complexity of $O((V + E) \log V)$.

A.12 Proof of correctness for ALT[7]

- Admissibility of $h(v)$: A heuristic is admissible if it never overestimates the true distance:

  $$h(v) \leq d(v,t)$$

  By the *triangle inequality*:

  $$d(v,t) \geq |d(L,v) - d(L,t)| \quad \forall L \in L$$

  Taking the maximum over all landmarks:

  $$d(v,t) \geq \max_{L \in L} |d(L,v) - d(L,t)| = h(v)$$

  Thus, $h(v)$ is admissible.

- Consistency of $h(v)$: A heuristic is consistent if for any edge (u,v):

  $$h(v) \leq d(u,v) + h(u)$$

  Using the *triangle inequality*:

  $$|d(L,v) - d(L,t)| \leq d(u,v) + |d(L,u) - d(L,t)|$$

  Taking the maximum over all landmarks:

  $$h(v) = \max_{L \in L} |d(L,v) - d(L,t)| \leq d(u,v) + h(u)$$

  Thus, $h(v)$ is consistent.

Since $h(v)$ is both admissible and consistent, the ALT algorithm guarantees optimal shortest paths.

A.13 Proof of complexity for ALT[7]

- Landmark Selection:
  - *Randomly selecting landmarks*: This is a constant-time operation, $O(1)$.
  - *Selecting high-degree or far-apart nodes*: This might involve sorting the nodes based on degree or distance, which would take $O(|V| \log |V|)$, where $|V|$ is the number of nodes in the graph.

- Precompute Shortest Path Distances from Landmarks: For k landmarks, we need to perform Dijkstra's algorithm k times, one for each landmark, making the total preprocessing time complexity is $O(k \cdot (|V| + |E|) \log |V|)$.
- Query Phase Complexity: The A* search algorithm with ALT uses the ALT heuristic $h(v)$ instead of a simple heuristic (like Euclidean distance). The time complexity of the A* search depends on the number of nodes expanded during the search and the priority queue operations. In the worst case, the complexity is $O((|V| + |E|) \log |V|)$. The average query time is generally much faster, but it is difficult to bound precisely without empirical data.

A.14 Proof of correctness for Hub Labellling[8]

- During the preprocessing phase of the Hub Labeling algorithm, we ensure that for any pair of nodes u and v, their labels $L(u)$ and $L(v)$ share at least one common hub h that lies on the shortest path between u and v.
- Formally, for any two nodes $u, v \in V$, there exists a hub $h \in L(u) \cap L(v)$ such that h lies on the shortest path between u and v, i.e., the path $u \to h \to v$ is a valid shortest path. Thus, this *Label Cover Property* ensures that the correct hubs are selected in the query phase and the algorithm can always compute the shortest path.
- For any two nodes u and v, during the query phase, the algorithm scans their labels $L(u)$ and $L(v)$ to find the hub h that minimizes the expression $d(u, h) + d(h, v)$. This is equivalent to finding the shortest path between u and v by traversing through a common hub h.
- Since h lies on the shortest path between u and v (by the Label Cover Property), the value $d(u, h) + d(h, v)$ is guaranteed to be the shortest path distance between u and v.
- Therefore, the Hub Labeling algorithm correctly computes the shortest path for any query (u, v), as it always finds the optimal hub h and ensures that the sum of distances $d(u, h) + d(h, v)$ corresponds to the actual shortest path distance.

A.15 Proof of complexity for Hub Labelling[8]

- For each node v, compute the forward and backward labels using Dijkstra's algorithm or a similar shortest path algorithm.
- The time complexity for computing labels for all nodes is $O(V \cdot (V + E) \log V)$, where V is the number of nodes and E is the number of edges.
- The selection of hubs can be done using various strategies, such as selecting nodes with high centrality or using a greedy algorithm. The time complexity for hub selection is $O(VlogV)$.
- For a given query (s, t), the forward label of s and the backward label of t are intersected to find the shortest path distance.
- The time complexity for label intersection is $O(|L_f(s)| + |L_b(t)|)$, where $|L_f(s)|$ and $|L_b(t)|$ are the sizes of the forward and backward labels, respectively.
- Each node stores a forward label and a backward label. The space complexity for storing all labels is $O(V \cdot L)$, where L is the average label size.

Appendix B

B.1 Final Algorithm

```
import heapq
import networkx as nx

# ===== USAGE =====
# Step 1: Precompute ALT landmarks
# preprocessor = ALTPreprocessor(graph, num_landmarks=8)

# Step 2: Run bidirectional A* with ALT heuristic
# visited_edges, shortest_path = bidirectional_alt_query(graph, start, end, preprocessor)

class ALTPreprocessor:
"""
Precomputes shortest path distances from a set of landmarks to all nodes.
Used to improve the A* heuristic with landmark-based estimation.
"""
def __init__(self, graph, num_landmarks=8):
self.graph = graph
self.num_landmarks = num_landmarks
self.landmarks = []
self.forward_distances = {}
self.backward_distances = {}

self._select_landmarks()
self._precompute_distances()

def _select_landmarks(self):
""" Selects the best landmarks based on shortest path coverage. """
```

```
28              components = list(nx.
                    strongly_connected_components(self.graph))
29              if not components:
30              return
31              largest_component = max(components, key=len)
32              subgraph = self.graph.subgraph(largest_component)
33              nodes = list(subgraph.nodes())
34
35              if not nodes:
36              return
37
38              self.landmarks = [nodes[0]]
39              landmarks_set = {nodes[0]}
40
41              try:
42              initial_dists = nx.shortest_path_length(subgraph,
                     nodes[0], weight='length')
43              except nx.NetworkXNoPath:
44              initial_dists = {}
45              min_dist = {node: initial_dists.get(node, float('
                    inf')) for node in nodes}
46
47              while len(self.landmarks) < self.num_landmarks
                    and len(self.landmarks) < len(nodes):
48              max_dist, next_landmark = max(
49              ((min_dist[node], node) for node in nodes if node
                     not in landmarks_set),
50              default=(-1, None)
51              )
52              if next_landmark is None:
53              break
54
55              self.landmarks.append(next_landmark)
56              landmarks_set.add(next_landmark)
57
58              try:
59              dists = nx.shortest_path_length(subgraph,
                    next_landmark, weight='length')
60              for node in nodes:
61              min_dist[node] = min(min_dist[node], dists.get(
                    node, float('inf')))
62              except nx.NetworkXNoPath:
63              continue
64
65              def _precompute_distances(self):
66              """ Precomputes shortest path distances from
                    landmarks for fast heuristic lookups. """
67              reversed_graph = nx.reverse(self.graph)
68              for L in self.landmarks:
```

```
69              if L in self.graph:
70              try:
71              self.forward_distances[L] = nx.
                    single_source_dijkstra_path_length(self.graph,
                     L, weight='length')
72              self.backward_distances[L] = nx.
                    single_source_dijkstra_path_length(
                    reversed_graph, L, weight='length')
73              except nx.NetworkXNoPath:
74              continue
75
76
77              def bidirectional_alt_query(graph, start, end,
                    preprocessor):
78              """
79              Bidirectional A* search with ALT heuristic for
                    fast shortest path queries.
80
81              Parameters:
82              graph: NetworkX graph representing the road
                    network.
83              start: Start node.
84              end: Destination node.
85              preprocessor: An instance of ALTPreprocessor
                    containing landmark data.
86
87              Returns:
88              visited_edges: List of edges explored during
                    search.
89              optimal_path: List of nodes representing the
                    shortest path.
90              """
91              visited_edges = []
92
93              # Retrieve precomputed ALT data
94              landmarks = preprocessor.landmarks
95              fw_dists = preprocessor.forward_distances
96              bw_dists = preprocessor.backward_distances
97
98              # Prepare precomputed heuristic values for all
                    landmarks
99              forward_data = [(bwd.get(end, float('inf')), fwd.
                    get(end, float('inf')), bwd, fwd)
100             for L in landmarks if (bwd := bw_dists.get(L, {})
                    ) and (fwd := fw_dists.get(L, {}))]
101
102             backward_data = [(bw_dists.get(L, {}).get(start,
                    float('inf')), fw_dists.get(L, {}).get(start,
                    float('inf')),
```

```
103             bw_dists.get(L, {}), fw_dists.get(L, {})) for L
                    in landmarks]
104
105             # Define ALT-based heuristic functions
106             def h_forward(u):
107             return max(max(bwd.get(u, float('inf')) - d_end_L
                    , d_L_end - fwd.get(u, float('inf')))
108             for d_end_L, d_L_end, bwd, fwd in forward_data if
                     u in bwd or u in fwd)
109
110             def h_backward(u):
111             return max(max(bwd.get(u, float('inf')) -
                    d_start_L, d_L_start - fwd.get(u, float('inf')
                    ))
112             for d_start_L, d_L_start, bwd, fwd in
                    backward_data if u in bwd or u in fwd)
113
114             # Initialize forward and backward search
115             forward_queue = [(h_forward(start), start)]
116             f_dist, f_prev = {start: 0}, {}
117
118             backward_queue = [(h_backward(end), end)]
119             b_dist, b_prev = {end: 0}, {}
120
121             best_cost = float('inf')
122             meeting_node = None
123             processed_f, processed_b = set(), set()
124
125             while forward_queue and backward_queue:
126             # Forward search step
127             f_prio, u = heapq.heappop(forward_queue)
128             if f_prio > best_cost:
129             break
130             if u in processed_f:
131             continue
132             processed_f.add(u)
133
134             if u in processed_b:
135             new_cost = f_dist[u] + b_dist[u]
136             if new_cost < best_cost:
137             best_cost, meeting_node = new_cost, u
138
139             for v in graph.neighbors(u):
140             edge_length = min(data['length'] for data in
                    graph[u][v].values())
141             new_g = f_dist[u] + edge_length
142             if new_g < f_dist.get(v, float('inf')):
143             f_dist[v] = new_g
144             f_prev[v] = u
```

```
            heapq.heappush(forward_queue, (new_g + h_forward(
                v), v))
            visited_edges.append((u, v))

            # Backward search step
            b_prio, u = heapq.heappop(backward_queue)
            if b_prio > best_cost:
            break
            if u in processed_b:
            continue
            processed_b.add(u)

            if u in processed_f:
            new_cost = f_dist[u] + b_dist[u]
            if new_cost < best_cost:
            best_cost, meeting_node = new_cost, u

            for v in graph.predecessors(u):
            edge_length = min(data['length'] for data in
                graph[v][u].values())
            new_g = b_dist[u] + edge_length
            if new_g < b_dist.get(v, float('inf')):
            b_dist[v] = new_g
            b_prev[v] = u
            heapq.heappush(backward_queue, (new_g +
                h_backward(v), v))
            visited_edges.append((v, u))

            return visited_edges, [meeting_node] if
                meeting_node else []
```