MA308 Mini Project
Report

Designing a shortest-path algorithm for large-scale graphs

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

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

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

List of Figures

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 Why Are Shortest Path Calculations Important?

- **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. **Enable customizable routing**: Users can define cost functions that prioritize specific factors such as travel time, distance, toll costs, or fuel consumption, allowing for personalized and adaptive route selection.

3. **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.

- **Customizable Routing**: The system will allow users to define personalized routing preferences based on multiple cost functions, making it adaptable for various use cases such as emergency response, shortest-distance travel, or eco-friendly routing.

- **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

Chapter 4

Algorithm Design

4.1 Preprocessing Phase

- Explain the choice of preprocessing techniques (e.g., Contraction Hierarchies, ALT).
- Describe how the graph is simplified or augmented with shortcuts/landmarks.
- Discuss trade-offs in preprocessing time and memory usage.

4.2 Query Execution

- Detail the hybrid algorithm combining preprocessing results with A* or bidirectional search.
- Include pseudocode for the query phase.

4.3 Dynamic Updates

- Describe methods for updating edge weights and recalculating shortest paths efficiently.
- Outline incremental update mechanisms.

4.4 Customization

- Define the composite cost function and how it integrates with the algorithm.

Chapter 5

Implementation

5.1 Tools and Technologies

- Programming language and libraries/frameworks used (e.g., Python, NetworkX, CUDA for parallelism).
- Hardware setup for experiments.

5.2 Code Structure

- Modular design: preprocessing, query execution, dynamic updates, and customization.

5.3 Dataset

- Description of datasets used (e.g., OpenStreetMap, SNAP datasets).
- Data preprocessing steps: parsing, cleaning, and formatting.

Chapter 6

Result and Analysis

6.1 Performance Metrics

- Preprocessing time and memory usage.
- Query execution time (average, worst-case).
- Accuracy of paths (for heuristic methods).
- Scalability with graph size and density.

6.2 Comparative Analysis

- Benchmark hybrid algorithm against standalone algorithms (e.g., plain Dijkstra's, A*).
- Use tables and plots to illustrate improvements.

6.3 Sensitivity Analysis

- Impact of graph properties (e.g., number of nodes, edge density).
- Effect of dynamic updates on performance.

Chapter 7

Discussion

7.1 Strengths

- Highlight significant improvements in speed, accuracy, or flexibility.
- Discuss adaptability to various real-world scenarios.

7.2 Limitations

- Discuss preprocessing overhead or constraints in memory usage.
- Identify scenarios where the hybrid approach might underperform.

7.3 Further Improvements

- Enhancing scalability for distributed systems.
- Incorporating machine learning to predict edge weights dynamically.
- Extending to multimodal routing or 3D graphs.

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] Cite all academic papers, libraries, datasets, and tools used. `<urlhere>`

Appendices

Appendix A

A.1 Proof of correctness for BFS

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

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

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

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

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

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

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

- 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

- 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

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

- 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

- 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

- 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

- 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

- 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.