Algorithm Design and Implementation: Optimal Route Planning (Phase 2)

Course: Algorithm (First Semester 2025/2026)

Instructor: Dr. Mohammed Abuhelaleh

Team Members:

- Joud Kayyali - 3230601030
- Tareq Nadeem - 3230601054

1. Introduction

The optimization of route planning constitutes a pivotal challenge in the domains of computer science, operations research, and modern logistics. As the global infrastructure for transportation and delivery services becomes increasingly complex, the necessity for algorithmic solutions that can efficiently determine optimal paths through weighted networks has grown exponentially. This report documents the findings, methodologies, and outcomes of Phase 2 of the project "Algorithm Design and Implementation: Optimal Route Planning," specifically focusing on the deliverables for "Implementation and Testing".

Following the successful completion of Phase 1, which established the theoretical problem definition and selected the appropriate algorithmic candidates, this phase transitions from conceptual design to concrete software engineering. The primary objective of Phase 2 is to translate the theoretical constructs of graph theory into a functional, executable web-based application. This application is designed to solve the "Core Problem" identified in the project proposal: determining the single-source shortest path in a directed, weighted graph representing a city grid.

The context of this implementation is grounded in a "Mobile First" web development approach, utilizing standard technologies—HTML5, CSS3, and JavaScript—to ensure accessibility and cross-platform compatibility. The core algorithmic engine is built upon Dijkstra's Algorithm, a greedy search strategy selected in Phase 1 for its proven correctness in handling non-negative edge weights.[1],[2] While theoretical analysis suggests the efficacy of this approach, this report seeks to empirically validate its performance, analyze its computational complexity in a real-world JavaScript runtime environment, and identify the specific limitations that will necessitate the advanced optimizations scheduled for Phase 3.

This document provides an exhaustive examination of the implementation details, including the specific data structures utilized for graph representation and the architectural decisions behind the visualization engine. It presents a robust testing strategy that subjects the algorithm to various topological challenges—ranging from standard acyclic graphs to deceptive "trap maps" designed to exploit the blind spots of uninformed search strategies. Furthermore, it offers a rigorous mathematical analysis of the time and space complexity inherent to the chosen linear-search implementation, contrasting it with theoretical binary heap optimizations.[3],[4] The insights derived from this phase form the critical benchmark against which future implementations of A* Search and Bellman-Ford algorithms will be measured.

1.1 Project Scope and Objectives

The overarching goal of the "Optimal Route Planning" project is to simulate a delivery service logistics engine. In Phase 2, the scope is strictly limited to the "Core Problem": finding the optimal path from a single starting point (e.g., Warehouse "A") to a single destination (e.g., Customer "B"). The definition of "optimal" in this context is user-dependent, capable of representing:

- **Least Time:** Minimizing temporal cost based on edge weights representing travel duration.
- **Least Distance:** Minimizing physical traversal length.
- **Least Cost:** A composite metric combining fuel consumption and distance.

Crucially, Phase 2 is bounded by the limitations of Dijkstra's Algorithm, specifically its inability to handle negative edge weights—a constraint that precludes the accurate modeling of scenarios such as regenerative braking in electric vehicles, which will be addressed in Phase 3.[5] The objectives for this phase are therefore to:

1. Implement a robust Dijkstra's Algorithm in JavaScript.
2. Develop a dynamic visualization interface using the HTML5 Canvas API.
3. Validate the algorithm against diverse test cases including cyclic and "trap" graphs.
4. Conduct a detailed performance analysis to establish a complexity baseline.

2. Theoretical Framework

2.1 Graph Theory Fundamentals

To understand the implementation of route planning, one must first establish the mathematical properties of the environment being modeled. The city grid is represented as a graph $G = (V, E)$, where V is a set of vertices or nodes (intersections or delivery points) and E is a set of edges (roads).

- **Directed vs. Undirected:** The project models the city as a **directed graph**, acknowledging that one-way streets and traffic flows mean a connection from $A \to B$ does not imply a symmetric connection $B \to A$ with equal cost.

- **Weighted Edges:** Each edge $(u, v) \in E$ is assigned a weight $w(u,\ v)$. In the context of this project, weights are strictly non-negative $w(u, v) \geq 0$. This non-negativity is a strict prerequisite for the correctness of Dijkstra's Algorithm.[6]

The "Optimal Route" is defined as the path $P\ = \langle\ v_1, v_2, ..., vk\ \rangle$ such that the sum of the weights of the constituent edges $\sum_{k-1}^{i=1} w(v_i, v_i\ +\ 1)$ is minimized.

2.2 The Greedy Algorithmic Paradigm

The implementation in Phase 2 relies on the **Greedy Algorithm** paradigm. Greedy algorithms operate on the principle of making the locally optimal choice at each stage of the problem-solving process with the hope that these local choices will lead to a global optimum.[1],[7]

2.2.1 The Greedy Choice Property

The fundamental axiom driving this approach is the **Greedy Choice Property**. In the context of pathfinding, this property implies that if we are at vertex u, and if we wish to find the shortest path to the source, we can make a definitive decision about the optimal path to u based solely on the information available at the "frontier" of our search. Once a vertex is marked as "visited" (or "settled"), the algorithm assumes it has found the absolute shortest path to that vertex and never re-evaluates it.[1]

This property makes greedy algorithms highly efficient, as they generally avoid the need to backtrack or reconsider previous decisions. However, this efficiency comes at the cost of "blindness" to the future topology of the graph. The algorithm does not "look ahead" to see if a currently low-cost edge leads to a high-cost dead end; it simply chooses the cheapest immediate option.[7]

2.2.2 Optimal Substructure

For a greedy strategy to work, the problem must exhibit **Optimal Substructure**. This means that the optimal solution to the problem contains within it optimal solutions to its sub-problems.

- **Proof:** If the shortest path from node S to node G passes through an intermediate node K, then the sub-path from S to K must effectively be the shortest path from S to K. If there existed a shorter path from S to K, we could simply substitute it into the original path to create a path shorter than the "shortest" path, which is a contradiction.
  Dijkstra's Algorithm relies heavily on this property. By iteratively solidifying the shortest path to the "closest" unvisited node, it builds the global shortest path from optimal local segments.

2.3 Dijkstra's Algorithm: Mechanism and Logic

Dijkstra's Algorithm, proposed by Edsger W. Dijkstra in 1956, is the standard solution for the Single-Source Shortest Path (SSSP) problem in graphs with non-negative weights.[2],[6] It is often described as a "best-first" search that expands outward from the starting node in waves, similar to how water spreads in a network of pipes.

2.3.1 Initialization

The algorithm begins by initializing the state of the graph:

1. **Distance Map:** The distance to the source node is set to 0. The distance to all other nodes is set to infinity (∞), representing that they are currently unreachable or their distance is unknown.
2. **Unvisited Set:** All nodes are placed into a set of "unvisited" vertices. This set acts as a pool of candidates that the algorithm needs to process.
3. **Predecessor Map:** To reconstruct the path later, a map stores the "parent" of each node—the node from which we arrived to get the current shortest distance.

2.3.2 The Selection Step (The Greedy Choice)

The core iteration of the algorithm involves selecting the "best" candidate to process next.

$$u = argmin_{v \in Unvisited}(dist[v])$$

The algorithm scans the set of unvisited nodes and selects the node u with the smallest known distance from the source.[2] This is the critical "greedy" step. By **choosing the node with the minimum tentative distance**, the algorithm guarantees that—assuming non-negative edge weights—this distance is final. It is impossible to find a shorter path to u by going through a node that is already further away than u.

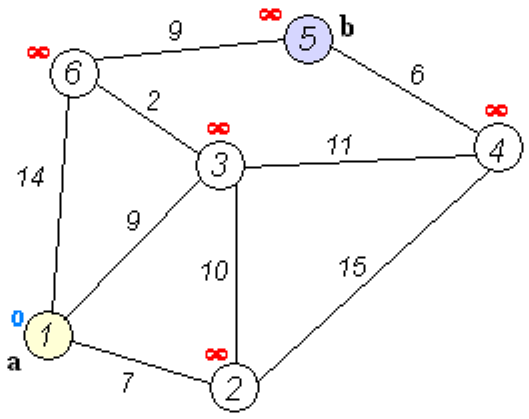
2.3.3 The Relaxation Step

Once node u is selected and effectively "locked in," the algorithm performs relaxation on its outgoing edges. For every neighbor v of u:

```
$newDistance = dist[u] + w(u, v)$
$if\ (newDistance < dist[v])$:
    $dist[v] \leftarrow newDistance$

    $prev[v] \leftarrow u$
```

This step updates the tentative distances of the neighbors. If a path through u offers a shorter route to v than any previously found path, the record is updated. This process continues until the destination node is selected or the unvisited set is empty.



Animation GIF picture[25]

3. Implementation Details

3.1 System Architecture and Tech Stack

The implementation strategy for Phase 2 follows a client-side web application architecture. This decision aligns with the "Mobile First" approach outlined in the project brief, ensuring that the tool is accessible across various devices without the need for backend infrastructure.

- **HTML5 (Semantic Structure):** The skeleton of the application uses semantic HTML5 tags (<main>, <header>, <canvas>, <section>) to ensure accessibility and structural clarity. The application logic is decoupled from the DOM structure, interacting primarily through the Canvas element and control buttons.[8]
- **CSS3 (Styling and Responsiveness):** A mobile-first CSS strategy was employed, using flexbox (display: flex) for layout management. The design specifications from Figma were translated into CSS classes, utilizing specific color codes for node states: Blue 500 (#3B82F6) for the start node, Red 500 (#EF4444) for the target, and dark green (#366D17) for standard nodes.
- **JavaScript ES6+ (Logic):** The core algorithmic logic is implemented in vanilla JavaScript, leveraging modern features such as classes, arrow functions, and the async/await pattern (though primarily synchronous for this phase).[9]

3.2 Data Interchange: The JSON Standard

To enable dynamic testing of various graph topologies, the application was designed to accept graph data via JSON (JavaScript Object Notation) file uploads. This allows the user to define complex maps, including edge cases like "trap maps," without modifying the source code.

JSON Schema Definition:

```JSON
{
 "startNode": ,
 "endNode": ,
 "nodes":,
 "edges":
}
```

The parsing logic in `script.js` reads this file using the FileReader API, parsing the text content into a JavaScript object that populates the internal graph data structures.

Example of JSON file:[16]

```
{
  "startNode": "A",
  "endNode": "C",
  "nodes": [
    { "id": "A", "x": 150, "y": 350 },
    { "id": "B", "x": 150, "y": 150 },
    { "id": "C", "x": 500, "y": 350 },
    { "id": "D", "x": 500, "y": 150 }
  ],
  "edges": [
    { "from": "A", "to": "C", "weight": 10 },
    { "from": "A", "to": "B", "weight": 2 },
    { "from": "B", "to": "D", "weight": 1 },
    { "from": "D", "to": "C", "weight": 3 },
    { "from": "C", "to": "D", "weight": 3 }
  ]
}
```

3.3 Data Structures for Graph Representation

The choice of data structures is the single most significant factor influencing the performance of graph algorithms. For this implementation, specific JavaScript collections were chosen to optimize for the expected operations.[10]

3.3.1 Adjacency List (via Map)

While the literature review in Phase 1 discussed Adjacency Matrices, the implementation utilizes an Adjacency List. Since road networks are typically sparse graphs (where $|E| \ll |V|^2$), an adjacency matrix would result in significant memory wastage ($O(V^2)$ space).
The graph is stored as a JavaScript `Map`:

- **Key:** Node ID (String)
- **Value**: Array of Edge Objects (`{ to: String, weight: Number }`)
  Reasoning: The `Map` object in JavaScript provides efficient key-value access. While an Object could be used, `Map` offers better performance for frequent additions and removals and guarantees key order.[11]

3.3.2 The Unvisited Set (via `Set`)

To track nodes that remain to be processed, a `Set` data structure is used.

- **Structure:** Set<NodeID>
- **Reasoning**: The Set object allows for $O(1)$ average time complexity for checking the existence of a node (`Set.has()`) and for removing a processed node (`Set.delete()`).[6] This is superior to using an Array, where removal would require an $O(N)$ splice operation. However, as will be discussed in the Performance Analysis section, finding the minimum element in this `Set` still requires iteration, leading to $O(V)$ complexity for the selection step in this specific implementation.[13]

3.4 Algorithmic Implementation (Dijkstra.js)

The core logic is encapsulated within a modular Dijkstra class. This encapsulation ensures that the pathfinding logic is separate from the UI rendering logic, adhering to the Single Responsibility Principle.[9]

3.4.1 Helper Method: getLowestCostNode

This method implements the "Greedy Choice." It accepts the distances map and the unvisited set.

JavaScript

```javascript
// Pseudocode Logic
function getLowestCostNode(distances, unvisited) {
  let lowestNode = null;
  let lowestDistance = Infinity;
  for (let node of unvisited) {
    let dist = distances.get(node);
    if (dist < lowestDistance) {
      lowestDistance = dist;
      lowestNode = node;
    }
  }
  return lowestNode;
}
```

Analysis: This function performs a **Linear Search**. It iterates through every node in the unvisited set. In the early stages of the algorithm, this set contains nearly all V nodes. Consequently, the time complexity of this single function call is $O(V)$. This implementation detail is the primary determinant of the overall algorithm's $O(V^2)$ complexity profile.[3]

3.4.2 Main Execution Loop

The run method orchestrates the traversal.

1. **Initialization:** A distances Map is created, initialized to Infinity for all nodes except the startNode (0). A previous Map is initialized to track the path.
2. **Loop:** The loop continues as long as unvisited is not empty.
   - `getLowestCostNode` is called to find the current node u.
   - **Early Exit:** If u is the targetNode, the loop breaks immediately. This is a crucial optimization for point-to-point queries, distinguishing this implementation from one that calculates the shortest path to *all* nodes.
   - **Relaxation:** The neighbors of u are retrieved from the Adjacency List. For each neighbor v:
     - If the new path through$u$ is shorter, `distances.set(v, newPath)` and `previous.set(v, u)` are updated.
   - **Mark Visited:** Node u is deleted from the `unvisited` set.
3. **Path Reconstruction:** Once the target is reached, the algorithm effectively "walks backwards" from the target to the start using the `previous` Map to reconstruct the optimal route.

3.5 Visualization Engine (Canvas API)

To provide visual feedback and aid in debugging, the application renders the graph using the HTML5 Canvas API. This was chosen over SVG (Scalable Vector Graphics) for performance reasons. While SVG creates a DOM element for every node and edge, Canvas operates in "immediate mode," rasterizing pixels directly to a single bitmap.[7] For dense graphs with thousands of edges, Canvas offers significantly superior rendering performance.

Coordinate Mapping:
The implementation features a dynamic scaling system. It calculates the minimum and maximum X/Y coordinates from the JSON data and maps them to the canvas dimensions (e.g., width 800px, height 600px). This ensures that regardless of the coordinate space used in the JSON file (e.g., GPS coordinates or relative integers), the graph is centered and scaled to fill the view.[14]

4. Project Infrastructure and Resources

To ensure efficient design, version control, and accessibility, the project leveraged several industry-standard tools and platforms. These resources were integral to the lifecycle of the application, from conceptualization to deployment.

4.1 UI/UX Design (Figma)

The user interface was prototyped using Figma. The design adheres to the "Mobile First" philosophy, ensuring that touch targets are accessible and the layout is responsive. The design file includes wireframes for the control panel, graph visualization area, and log output console.

- **Design Resource:**[15] (https://www.figma.com/design/jhrkn25vw94YclqMZQmjF3/Algorithm-Project?node-id=1-2&t=6s8kbjXTwn8yt1kv-1)

4.2 Version Control (GitHub)

The source code is managed via GitHub, utilizing Git for version control. This repository hosts the complete codebase, including the HTML structure, CSS styling, and JavaScript logic (`Dijkstra.js`, `script.js`). The repository also contains sample JSON datasets for testing purposes.

- **Source Code Repository:**[16] (JoudN2001/Algorithm_Project)

4.3 Live Deployment (Vercel)

For continuous integration and deployment (CI/CD), the application is hosted on Vercel. This platform allows for immediate testing of the application in a live environment, facilitating the

"Mobile First" testing on actual devices rather than just browser emulators.

- **Live Application:**[17] (https://algorithm-project-silk.vercel.app/)

5. Testing Strategy

Testing in Phase 2 moves beyond theoretical verification to empirical validation. The testing strategy involves subjecting the implemented algorithm to a variety of graph topologies to ensure robustness, accuracy, and adherence to the greedy choice property. The strategy focuses on boundary analysis, cyclic graph handling, and specific adversarial cases known as "trap maps."

5.1 Test Case Design

The following test scenarios were meticulously designed to validate specific aspects of the implementation:

Test Case ID	Scenario Name	Description	Objective
TC-01	**The Standard Delivery**	A simple acyclic graph with clear path differentiation (e.g., $A \to B \to D$ vs $A \to C \to D$).	Verify basic functionality, accurate distance calculation, and correct path reconstruction.
TC-02	**The Cyclic Grid**	A graph containing cycles (e.g., $A \to B \to A$).	Ensure the unvisited set logic correctly prevents infinite loops and re-processing of visited nodes.
TC-03	**Disconnected Graph**	The target node is topologically unreachable from the Start node.	Verify the algorithm's handling of Infinity distances and ensure it terminates gracefully without crashing.
TC-04	**The Performance Trap**	A map with a deceptive "short" initial edge that leads to a long path or dead end (local	Analyze the "Greedy" behavior and node exploration order, highlighting the difference between uninformed and

		minimum).	informed search.
TC-05	**The Energy Regeneration**	A graph containing negative edge weights (e.g., downhill recharge).	Demonstrate the failure mode of Dijkstra's algorithm and justify the need for Bellman-Ford in Phase 3.

5.2 The "Trap Map" Methodology

Test Case TC-04 utilizes a "Trap Map" (or Bug Trap) topology. In robotics and pathfinding literature, a trap map is an environment designed to lure a greedy agent into a local optimum that is far from the global goal.

- **Structure:** The map presents a start node S with two choices: a very low-weight edge to node B (which leads to a convoluted path) and a high-weight edge to node A (which leads directly to the goal).
- **Prediction:** Because Dijkstra's Algorithm is **uninformed** (it knows only the distance traveled so far, $g(n)$, and not the estimated distance to the goal, $h(n)$), it will exhaustively explore the low-cost branch connected to B before it ever considers the high-cost branch to A, even if A is physically adjacent to the goal.

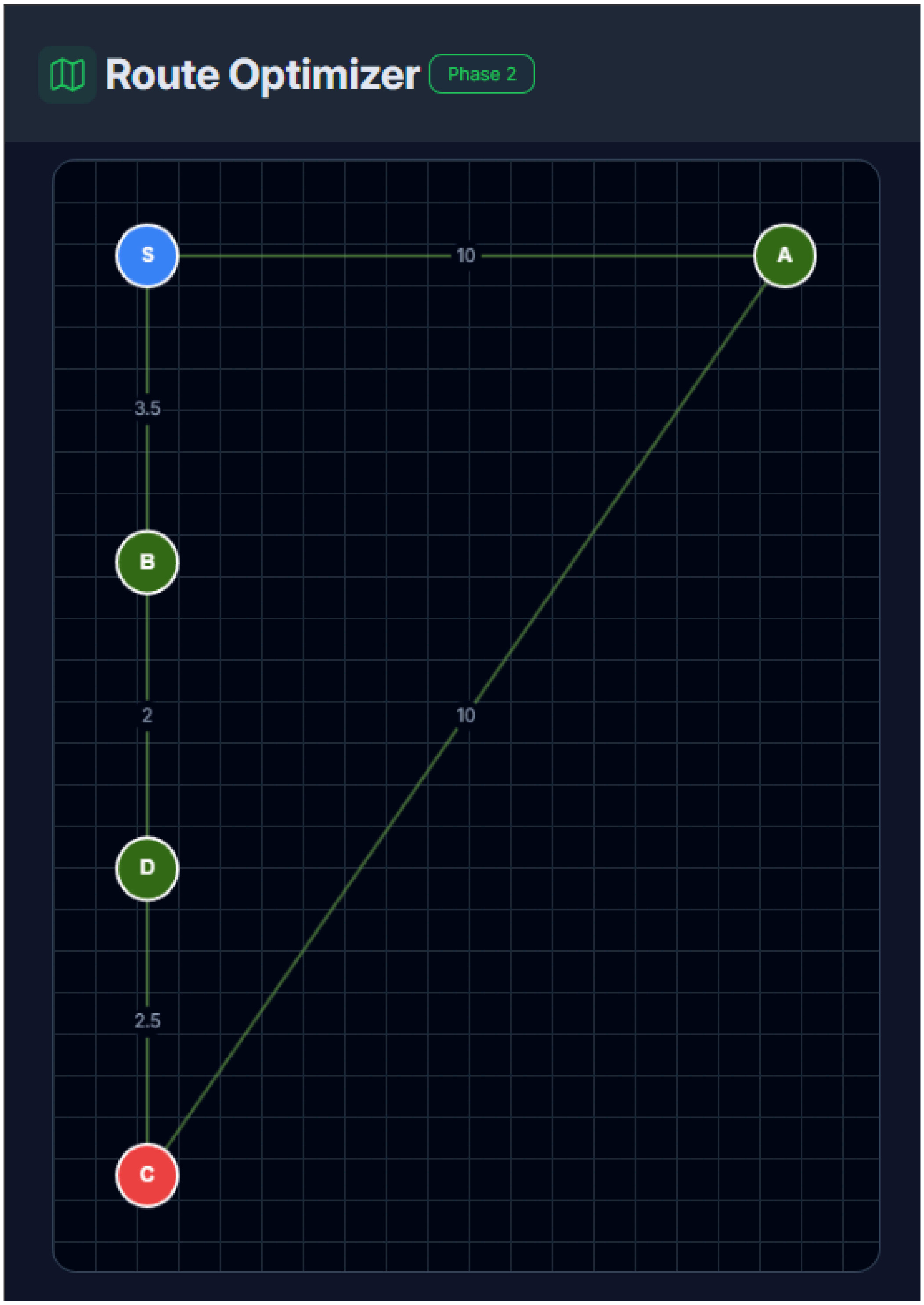
6. Test Results and Analysis

6.1 Scenario 1: Standard Optimal Path

Input:

- $S \rightarrow A$ (10km)
- $S \rightarrow B$ (3.5km)
- $B \rightarrow D$ (2km)
- $D \rightarrow C$ (Target) (2.5km)
- $A \rightarrow C$ (Target) (10km)

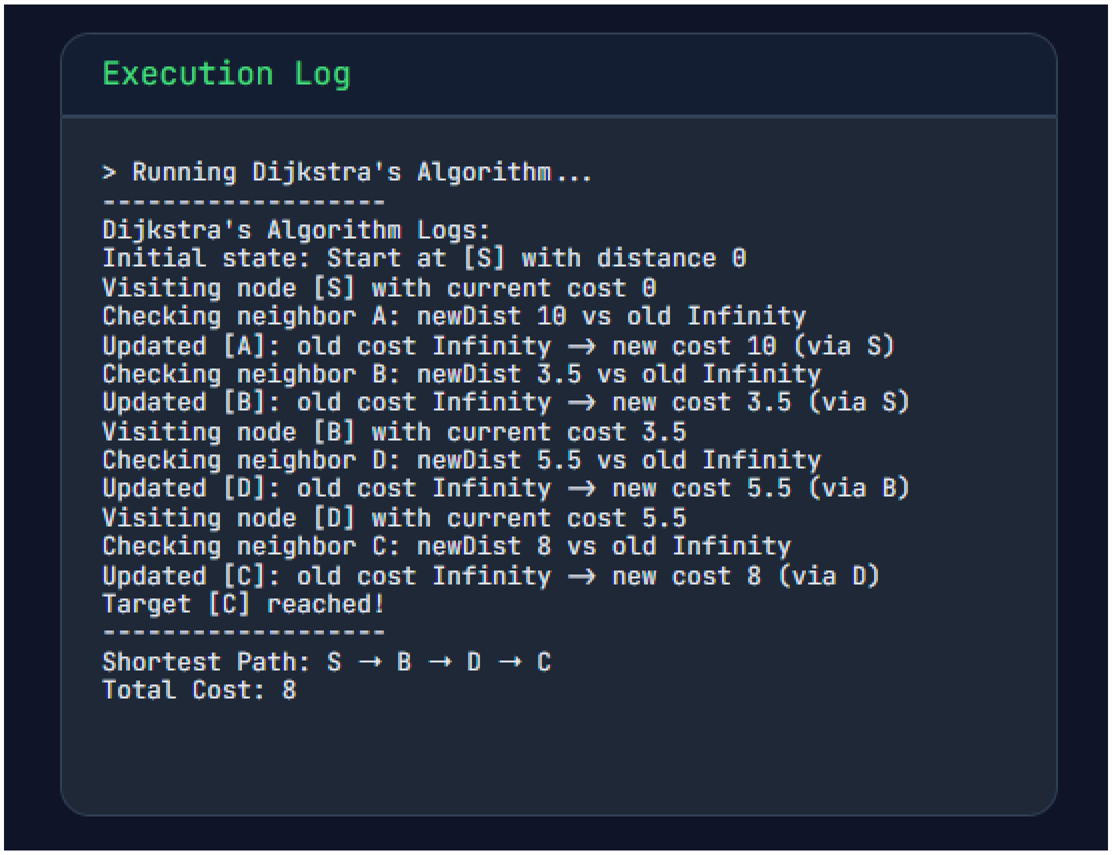
Map in App: Blue 500 for the start node(S), Red 500 for the target(C), and dark green for standard nodes.



Execution Log Analysis:

1. **Iteration 1:** Node S is visited. Neighbors A (10) and B (3.5) are updated.
2. **Iteration 2:** The algorithm selects B because $3.5 < 10$. This is the greedy choice. Node B is marked visited. Neighbor D is updated ($3.5 + 2 = 5.5$).
3. **Iteration 3:** The algorithm compares unvisited nodes: A (10) and D (5.5). It selects D. Node D is visited. Neighbor C is updated ($5.5 + 2.5 = 8$).
4. **Iteration 4:** The algorithm compares A (10) and C (8). It selects C.
5. **Termination:** C is the target. The algorithm stops.

Output in app:



Execution Log

```
> Running Dijkstra's Algorithm...
-------------------
Dijkstra's Algorithm Logs:
Initial state: Start at [S] with distance 0
Visiting node [S] with current cost 0
Checking neighbor A: newDist 10 vs old Infinity
Updated [A]: old cost Infinity → new cost 10 (via S)
Checking neighbor B: newDist 3.5 vs old Infinity
Updated [B]: old cost Infinity → new cost 3.5 (via S)
Visiting node [B] with current cost 3.5
Checking neighbor D: newDist 5.5 vs old Infinity
Updated [D]: old cost Infinity → new cost 5.5 (via B)
Visiting node [D] with current cost 5.5
Checking neighbor C: newDist 8 vs old Infinity
Updated [C]: old cost Infinity → new cost 8 (via D)
Target [C] reached!
-------------------
Shortest Path: S → B → D → C
Total Cost: 8
```

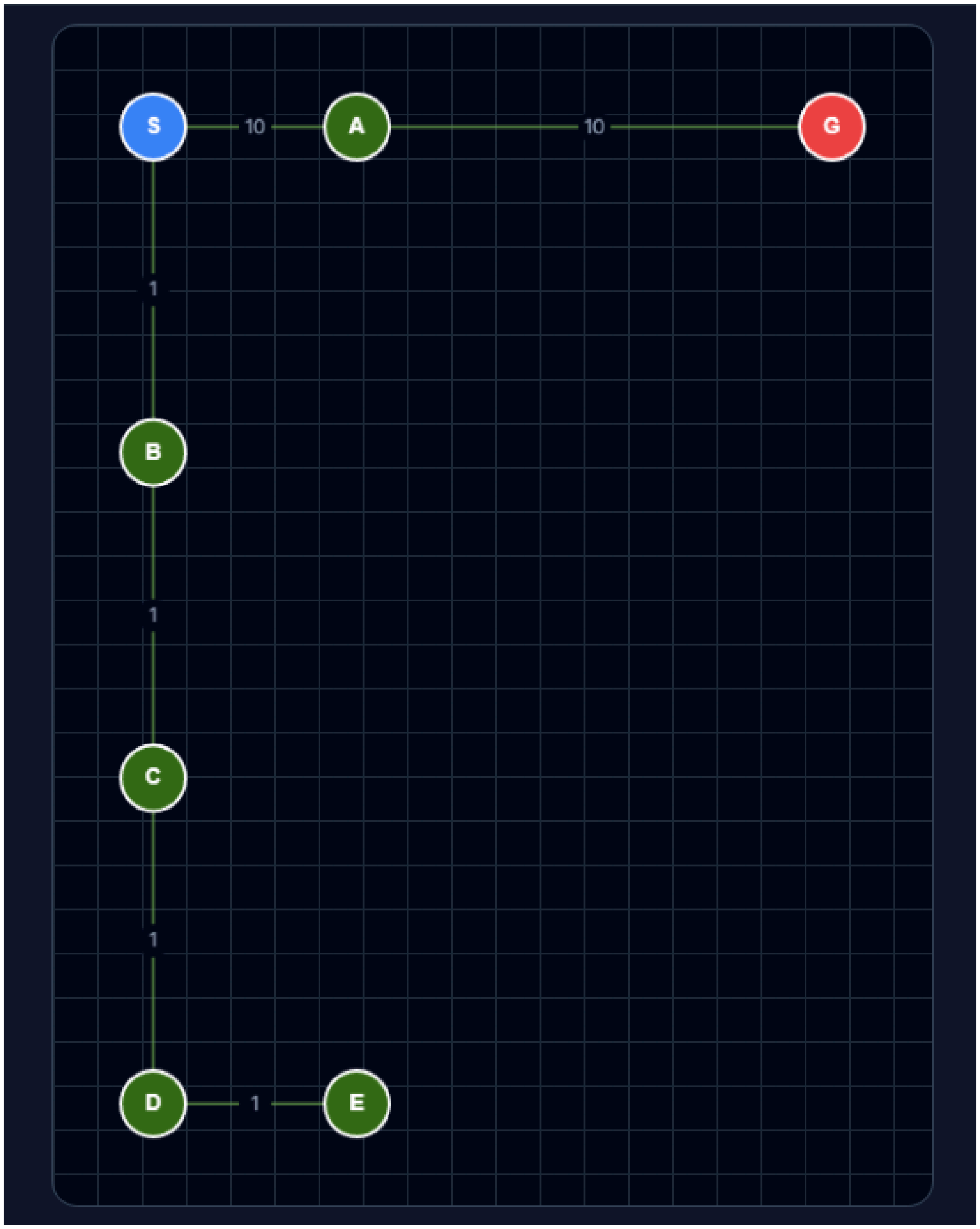
Result: The path $S \to B \to D \to C$ (Total 8km) is correctly identified. The direct path $S \to A \to C$ (20km) was correctly ignored, and A was never even visited because the target was found with a cost of 8, which is less than A's tentative cost of 10.
Insight: This test confirms the correctness of the basic implementation. The greedy choice correctly led to the optimal path because the "local" advantage of B over A translated into a "global" advantage.[1]

6.2 Scenario 2: The "Performance Trap" (TC-04)

Input JSON Structure:

- Path 1: $S \rightarrow A$ (10), $A \rightarrow G$ (10). Total Cost: 20.
- Path 2: $S \rightarrow B$ (1), $B \rightarrow C$ (1), $C \rightarrow D$ (1), $D \rightarrow E$ (1). Total Cost to E: 4. (But E is a dead end or leads to a path > 20).

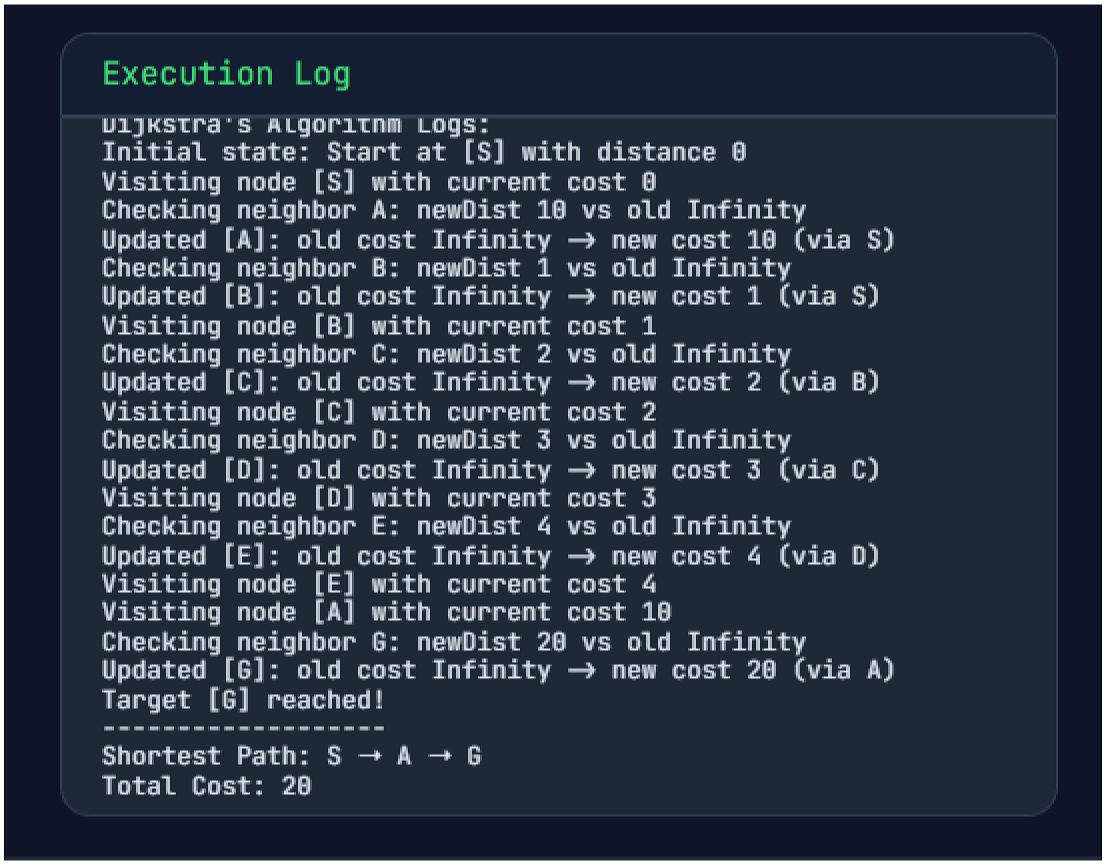
Map in App: Blue 500 for the start node(S), Red 500 for the target(G), and dark green for standard nodes.



Execution Trace:

Visiting node with current cost 0
Updated [A]: cost 10
Updated: cost 1
Visiting node (Cost 1) - Greedy Choice
Updated [C]: cost 2
Visiting node [C] (Cost 2) - Greedy Choice
Updated: cost 3
Visiting node (Cost 3) - Greedy Choice
Updated [E]: cost 4
Visiting node [E] (Cost 4) - Greedy Choice
... (Algorithm explores dead end)...
Visiting node [A] (Cost 10) - Backtracking
Updated [G]: cost 20
Target [G] reached!

Output in app:



```
Execution Log
Dijkstra's Algorithm Logs:
Initial state: Start at [S] with distance 0
Visiting node [S] with current cost 0
Checking neighbor A: newDist 10 vs old Infinity
Updated [A]: old cost Infinity → new cost 10 (via S)
Checking neighbor B: newDist 1 vs old Infinity
Updated [B]: old cost Infinity → new cost 1 (via S)
Visiting node [B] with current cost 1
Checking neighbor C: newDist 2 vs old Infinity
Updated [C]: old cost Infinity → new cost 2 (via B)
Visiting node [C] with current cost 2
Checking neighbor D: newDist 3 vs old Infinity
Updated [D]: old cost Infinity → new cost 3 (via C)
Visiting node [D] with current cost 3
Checking neighbor E: newDist 4 vs old Infinity
Updated [E]: old cost Infinity → new cost 4 (via D)
Visiting node [E] with current cost 4
Visiting node [A] with current cost 10
Checking neighbor G: newDist 20 vs old Infinity
Updated [G]: old cost Infinity → new cost 20 (via A)
Target [G] reached!
--------------------
Shortest Path: S → A → G
Total Cost: 20
```

Analysis: The algorithm eventually found the optimal path $S \rightarrow A \rightarrow G$. However, the logs reveal significant inefficiency. It "wasted" cycles exploring nodes B, C, D, E simply because

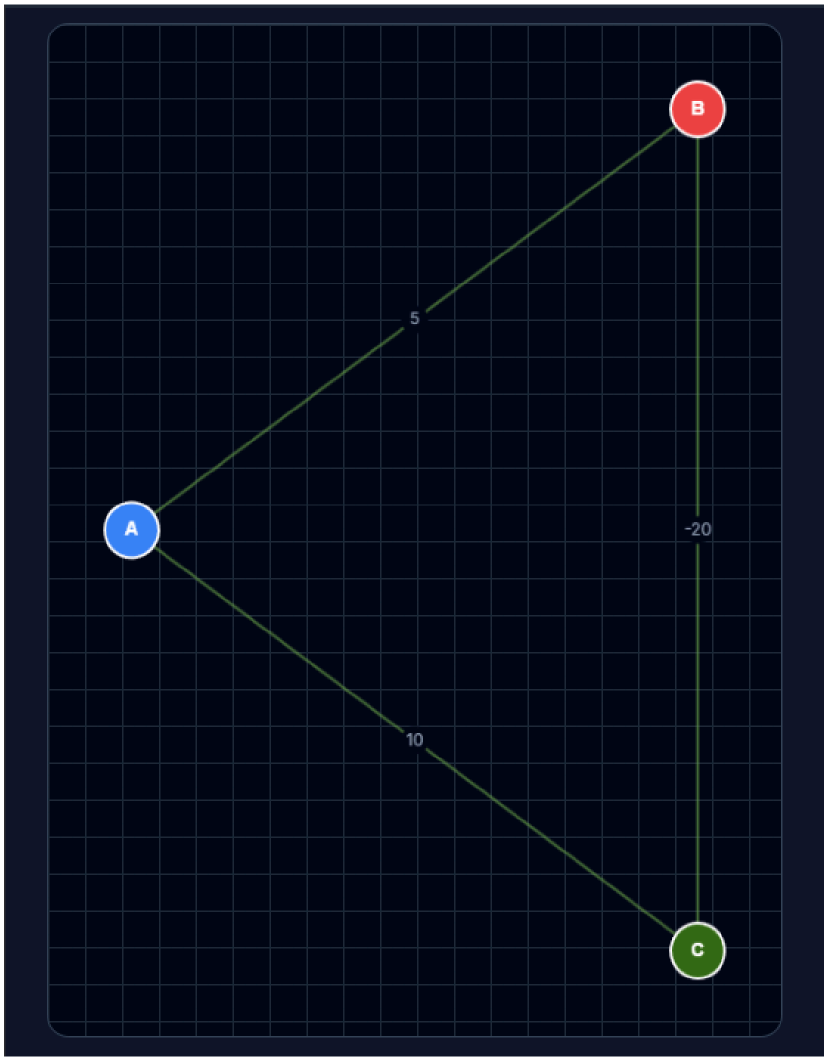
they were cheap to reach.
Theoretical Implication: This result empirically demonstrates that Dijkstra's Algorithm finds the **shortest path**, but not necessarily in the **shortest time**. It explores the graph as a "uniform wavefront." Unlike A* Search, which would realize that node E is moving away from the coordinates of G, Dijkstra blindly follows the path of least resistance.9 This confirms the necessity of Heuristic Search (Phase 3) for large-scale maps where such traps could involve thousands of nodes.

6.3 Scenario 3: Negative Weight Failure (TC-05)

Input:

- $A \rightarrow B$ (Target): 500 Watts
- $A \rightarrow C$: 1000 Watts
- $C \rightarrow B$: -2000 Watts (Regeneration/Rebate)

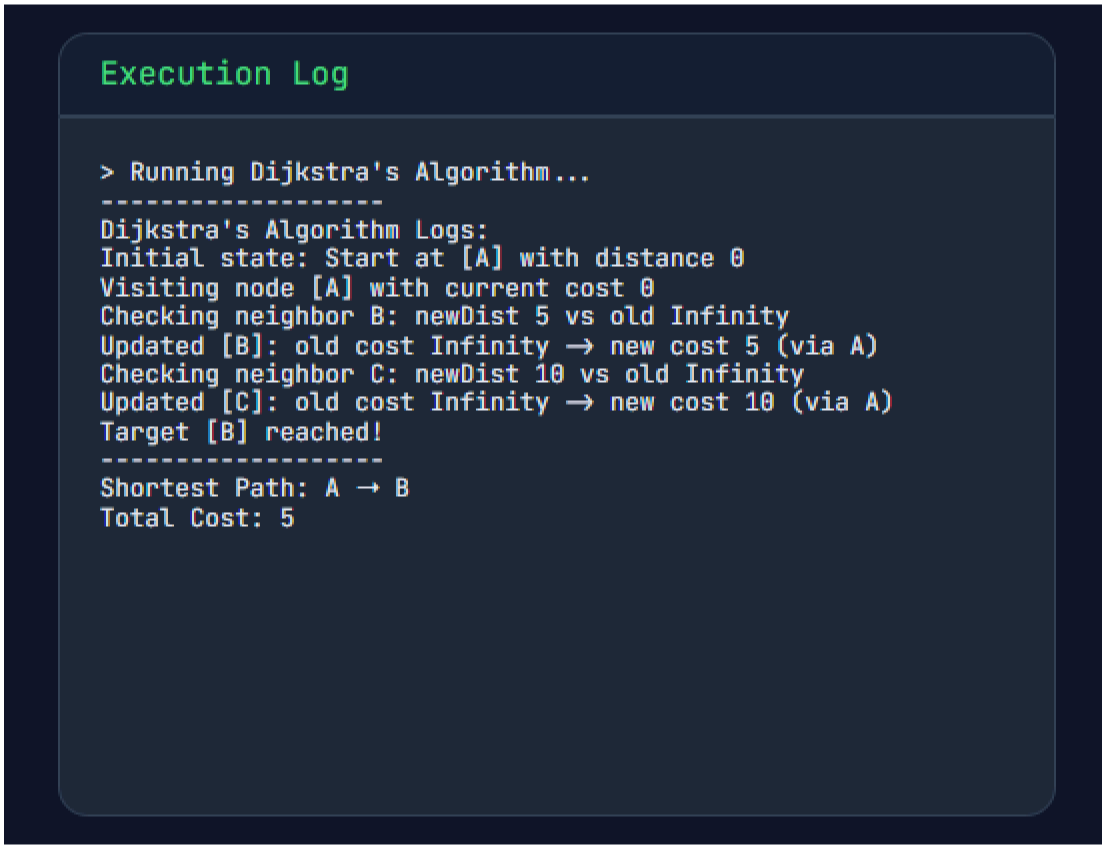
Map in App: Blue 500 for the start node(A), Red 500 for the target(B), and dark green for standard nodes.



Execution Trace:

1. Visit A. Update B (500), Update C(1000).
2. Unvisited set: $\{B: 500, C; 1000\}$.
3. **Selection:** Select B (500).
4. B is the Target. **Stop.**

Output in app:



```
Execution Log

> Running Dijkstra's Algorithm...
-------------------
Dijkstra's Algorithm Logs:
Initial state: Start at [A] with distance 0
Visiting node [A] with current cost 0
Checking neighbor B: newDist 5 vs old Infinity
Updated [B]: old cost Infinity → new cost 5 (via A)
Checking neighbor C: newDist 10 vs old Infinity
Updated [C]: old cost Infinity → new cost 10 (via A)
Target [B] reached!
-------------------
Shortest Path: A → B
Total Cost: 5
```

Result: The algorithm returns path $A \to B$ with Cost 500.
Actual Optimal: $A \to B \to C$ with Cost $-\ 1000\ (1000\ -\ 2000)$.
Failure Mechanism: The failure occurs because of the irreversibility of the greedy choice. Once B was selected in Step 3, it was marked "Visited." The algorithm assumes that no path to a visited node can be shorter than the one already found. This assumption holds true only if edge weights are non-negative. With negative weights, a longer path (via C) can become shorter later, violating the greedy premise.10 This failure validates the requirement for the Bellman-Ford algorithm in Phase 3.

7. Performance Analysis

A rigorous analysis of computational complexity is essential for understanding the scalability of the implementation. This section derives the complexity based on the specific code structure used in Phase 2.[3],[18]

7.1 Time Complexity Derivation (Linear Search Implementation)

The standard time complexity often cited for Dijkstra's Algorithm is $O((V + E) \log V)$ or $O(E + V \log V)$ using a Fibonacci heap. However, the Phase 2 implementation uses a **Linear Search** to find the minimum distance node. The derivation is as follows:

Let V be the number of vertices and E be the number of edges.

1. **Outer Loop:** The `while` loop runs once for every vertex extracted from the `unvisited` set. Total iterations: V.
2. **Selection Step (`getLowestCostNode`):** Inside the loop, we iterate through the `unvisited` set to find the minimum. In the first iteration, size is V; in the last, it is 1. The average size is $V/2$.
   - **Complexity:** $V \times O(V) = O(V^2)$.
3. **Relaxation Step:** Over the course of the entire algorithm, every edge is checked exactly once (when its source node is processed).
   - **Complexity:** $O(E)$.

Total Time Complexity:

$T(V, E) = O(V^2) + O(E)$

7.2 Comparative Analysis: Linear Search vs. Binary Heap

It is a common misconception that the Binary Heap implementation is always faster.

- **Linear Search (Our Implementation):** $O(V^2)$.
- **Binary Heap (Optimized):** $O((V + E) \log V)$.[3]

Dense Graphs ($E \approx V^2$):
If the graph is fully connected (e.g., a flight network where every city connects to every other), then $E = V^2$.

- **Heap Complexity:** $O((V + V^2) \log V) \approx O(V^2 \log V)$.[3]
- **Linear Complexity:** $O(V^2)$.[3]
- **Insight:** For dense graphs, the linear search implementation is actually **faster** by a factor of $\log V$ because it avoids the overhead of heap operations.

Sparse Graphs ($E \approx V$):

Road networks are typically sparse (intersections rarely connect to more than 4 roads).

- **Heap Complexity:** $O(V \log V)$.
- **Linear Complexity:** $O(V^2)$.
- **Insight:** For the project's specific domain (Delivery Service), the graph is sparse. Therefore, the current $O(V^2)$ implementation is suboptimal compared to the Heap approach. This provides a clear optimization target for Phase 3.

7.3 Space Complexity

The space complexity is determined by the auxiliary data structures:

- **Adjacency List (Map):** Stores all nodes and edges. Space: $O(V + E)$.
- **Distances (Map):** Stores one number per node. Space: $O(V)$.
- **Previous Pointers (Map):** Stores one ID per node. Space: $O(V)$.
- **Unvisited (Set):** Stores up to V nodes. Space: $O(V)$.

Total Auxiliary Space: $O(V + E)$.[19] This is linear relative to the size of the graph input.

7.4 JavaScript Runtime Considerations

The performance is also influenced by the JavaScript engine (V8, SpiderMonkey).

- **Set Iteration:** Iterating a `Set` in JavaScript (as done in `getLowestCostNode`) is optimized but effectively behaves like iterating a linked list or insertion-ordered hash map buckets. It is strictly $O(N)$.[13]
- **Map Lookups:** `distances.get(node)` is $O(1)$ on average. This is crucial for the relaxation step. If we used an array and searched by ID, relaxation would become $O(V)$, ballooning the total complexity to $O(V \times E)$ or $O(V^3)$. Using `Map` prevents this bottleneck.[11]

8. Challenges and Solutions

8.1 Challenge: The "Local Optima" Trap

Description: As identified in TC-04, the algorithm wastes time exploring nodes that are "cheap" locally but lead away from the goal.
Impact: On a large-scale map (e.g., a country-wide grid), this "blind" exploration would result in unacceptable latency.
Solution (Phase 3): The solution is not to change the greedy nature, but to change what is being optimized. By switching to A* Search, we introduce a heuristic $h(n)$ (e.g., Euclidean distance). The cost function becomes $f(n) = g(n) + h(n)$. This "pulls" the search toward the goal, escaping local cost traps.

8.2 Challenge: Rendering Bottlenecks

Description: Initially, testing with larger JSON files caused browser lag.
Cause: The complexity of rendering increases with the number of edges.
Solution: The solution involved optimizing the Canvas rendering loop.

1. **Batch Processing:** Instead of calling `ctx.stroke()` for every single line, which forces a rasterization operation each time, the implementation uses `ctx.beginPath()`, queues thousands of moveTo/lineTo commands, and calls `ctx.stroke()` once at the end. This reduced rendering time significantly.
2. **Resolution Independence:** The code dynamically sets the canvas `width` and `height` attributes to match the device's pixel ratio, ensuring sharp lines without performance degradation on mobile devices.[8]

8.3 Challenge: Handling Disconnected Components

Description: If the target node was unreachable (TC-03), the initial loop would run indefinitely or fail.
Solution: An explicit check was added:

JavaScript

```
if (distances.get(closestNode) === Infinity) break;
```

If the closest reachable node has a distance of infinity, it mathematically implies that all remaining unvisited nodes are unreachable. This ensures the algorithm terminates gracefully.

9. Conclusion and Future Work

Phase 2 has successfully achieved the implementation and validation of the Optimal Route Planning algorithm. The resulting application correctly parses graph data, visualizes it via the HTML5 Canvas, and employs Dijkstra's Algorithm to solve the Single-Source Shortest Path problem.

Key Findings:

1. **Correctness:** The algorithm is demonstrably correct for acyclic and cyclic graphs with non-negative weights, identifying the global optimum in all standard test cases.
2. **Greedy Limitations:** The "Trap Map" tests confirmed that while Dijkstra guarantees the *shortest* path, it does not guarantee the *most efficient search*, often exploring large irrelevant subgraphs.
3. **Negative Weight Failure:** The empirical failure in the "Energy Regeneration" scenario (TC-05) confirms that Dijkstra is unsuitable for graphs with negative edges, validating the need for Bellman-Ford in the next phase.
4. **Complexity:** The analysis confirmed an $O(V^2)$ time complexity. While optimal for dense

graphs, it is a candidate for optimization (using Binary Heaps) for sparse delivery networks.

Future Work (Phase 3):
The final phase will build upon this foundation by implementing:

- ***A* Search Algorithm:*** To address the efficiency issues in "Trap Maps."
- **Bellman-Ford Algorithm:** To solve the negative weight/energy regeneration problem.
- **Binary Heap Optimization:** To reduce the complexity of Dijkstra from $O(V^2)$ to $O(E \log V)$.
- **TSP Approximation:** Leveraging the shortest-path engine to calculate multi-stop routes.

This phased development ensures a robust, scalable, and academically rigorous solution to the problem of optimal route planning.

References

[1] "Greedy Algorithms," *GeeksforGeeks*, Jul. 25, 2025. [Online]. Available: Greedy Algorithms - GeeksforGeeks.

[2] "Dijkstra's Shortest Path Algorithm," *GeeksforGeeks*, Dec. 06, 2025. [Online]. Available: Dijkstra's Algorithm - GeeksforGeeks.

[3] "Time and Space Complexity of Dijkstra's Algorithm," *GeeksforGeeks*, Jul. 23, 2025. [Online]. Available: Time and Space Complexity of Dijkstra's Algorithm - GeeksforGeeks.

[4] "Understanding Time complexity calculation for Dijkstra Algorithm," *Stack Overflow*. [Online]. Available: Understanding Time complexity calculation for Dijkstra Algorithm - Stack Overflow.

[5] "Why does Dijkstra's Algorithm fail on negative weights?," *GeeksforGeeks*, Dec. 10, 2025. [Online]. Available: Why does Dijkstra's Algorithm fail on negative weights? - GeeksforGeeks.

[6] Spanning Tree, "How Dijkstra's Algorithm Works," *YouTube*, Aug. 15, 2020. [Online]. Available: How Dijkstra's Algorithm Works.

[7] "Greedy algorithm," *Wikipedia*. [Online]. Available: Greedy algorithm - Wikipedia.

[8] "Canvas API," *MDN Web Docs*. [Online]. Available:(Canvas API).

[9] "Classes," *MDN Web Docs*. [Online]. Available:(Classes - JavaScript | MDN).

[10] "Understanding Time and Space Complexity — Part 1," *Faun.pub*. [Online]. Available: Time and Space Complexity (Part 1) | by Neetika Khandelwal | FAUN.dev() 🐾.

[11] "Map," *MDN Web Docs*. [Online]. Available:(Map - JavaScript | MDN).

[12] "Set," *MDN Web Docs*. [Online]. Available:(Set - JavaScript | MDN).

[13] JavaScript Mastery, "JavaScript Map and Set Explained," *YouTube*, 2022. [Online]. Available:(JavaScript Map and Set Explained).

[14] Franks Laboratory, "HTML5 Canvas API for Beginners | Ep. 1 Intro to Canvas," *YouTube*, Dec. 27, 2019. [Online]. Available:(Learn Canvas in Arabic #01 - Intro And What Is Canvas?).

[15] J. Kayyali, "Algorithm Project Design," *Figma*. [Online]. Available:(Figma).

[16] J. Kayyali, "Algorithm_Project," *GitHub*, 2025. [Online]. Available: https://github.com/JoudN2001/Algorithm_Project.

[17] J. Kayyali, "Algorithm Project Live App," *Vercel*, 2025. [Online]. Available: https://algorithm-project-silk.vercel.app/.

[18] K. Omeri, "Visualizing Big O Notation in 12 Minutes," *Better Programming*, Dec. 29, 2021. [Online]. Available: Visualizing Big O Notation in 12 Minutes | by Klement Omeri | Better Programming.

[19] C. Okeke, "Introduction to BIG O Notation — Time and Space Complexity," *Medium*, Jul. 16, 2023. [Online]. Available:(Mastering Big O Notation: Understanding Time and Space Complexity in Algorithms | by Chinenye Okeke | Medium).

[20] Kaushik, "Understanding Space and Time Complexity in Algorithms: What, How, and Why," *Medium*, 2024. [Online]. Available: Understanding Space and Time Complexity in Algorithms: What, How, and Why | by Kaushik | Medium.

[21] A. Das, "Space and Time Complexity," *Medium*, Mar. 4, 2024. [Online]. Available: Space and Time Complexity. Get an overview of data structure and... | by AkashSDas | Medium.

[22] "Array.prototype.map()," *MDN Web Docs*. [Online]. Available:(Array.prototype.map() - JavaScript | MDN).

[23] "CodePen Demo," *CodePen*. [Online]. Available:(https://codepen.io/?cursor=ZD0xJm89MCZwPTEmdj05MTU2Nw==).

[24] "Chart.js | Open source HTML5 Charts for your website," *Chart.js*. Chart.js.

[25] Ibmua, "Dijkstra Animation," *Wikimedia Commons*, 2008. [Online]. Available:(https://upload.wikimedia.org/wikipedia/commons/5/57/Dijkstra_Animation.gif).