Project Report: Course Project-3

CS51510-001, Fall 2025
Purdue University Northwest
11/04/2025

Class ID (Ascend sequency)	Contributor Name (Last Name, First Name)	Email	Detailed Contributions
038414971	**Mishra, Amartya**	**Mishr259@pnw.edu**	**Documentation**
	Mammai, Sreeja	**Smammai@pnw.edu**	**Data Collection**
038410044	**Mhasawade, Siddhi**	**Smhasawa@pnw.edu**	**Performance Testing**
0038159254	**Mekala, Ruthvik**	**Rmekala@pnw.edu**	**Python Code**
039133149	**Patel, Raaj**	**Pate2682@pnw.edu**	**Research Report and Architecture Diagram**
0037874079	**Modi, Hirak**	**Modi54@pnw.edu**	**Project Presentation**
0038119426	**Nalluri, Prasanth**	**Pnallur@pnw.edu**	**Python routing logic and implementation**

Table of Contents

Table of Figures

Abstract

This project develops an integrated travel-route optimization framework that combines classical graph algorithms with real-world environmental, geographic, and operational constraints. The system constructs a weighted, elevation-aware transportation graph across 28 U.S. cities, where each edge incorporates Google-Maps derived distances, sea-level differences, and a physics-based adjustment using $\tan(\theta)$ to represent fuel penalties associated with incline. To reflect realistic travel conditions, the model incorporates daily weather-risk values for each city during November 1–30, 2025, and computes the day with the minimum accumulated risk along any proposed route. The platform further estimates fuel consumption, enforces speed and daily-driving limitations, and derives a combined route-quality score that balances distance, fuel usage, and safety risk.

Five fundamental algorithms Breadth-First Search (BFS), Depth-First Search (DFS), Prim's Minimum Spanning Tree, Kruskal's Minimum Spanning Tree, and the Bellman–Ford shortest-path algorithm were implemented in Python and exposed through a lightweight HTTP API. A browser-based visualization layer built with D3.js renders the U.S. map, city nodes, and route segments, allowing users to compare algorithmic behaviours interactively. Extensive experimentation highlights the differing structural assumptions of each algorithm, their sensitivity to edge-weight models, and their impact on total distance, gas consumption, and weather-risk exposure. Performance profiling on a Linux server further evaluates CPU, memory, and I/O behaviour during large-scale route computations.

Beyond demonstrating algorithmic correctness, this project emphasizes the importance of integrating contextual data into route planning. Real-world travel is rarely optimized by distance alone; elevation, fuel cost, and regional weather patterns may dramatically alter what constitutes a “best” route. By combining classical graph traversal with empirical risk metrics and environmental modelling, the system illustrates how computational methods can be expanded to deliver more realistic and safety-conscious travel recommendations. This approach offers a blueprint for broader applications in logistics, disaster planning, and intelligent transportation systems.

I. Introduction

Route planning is a fundamental problem in transportation science, logistics, and computer science. Classical graph algorithms provide an elegant theoretical foundation for computing paths, yet real-world travel rarely aligns with the simplified conditions assumed in textbook formulations. Distances fluctuate with terrain, fuel consumption depends on elevation changes, and environmental factors such as rain, snow, and cloud cover may influence the safety and practicality of travel on any given day. The central motivation behind this project is to bridge this gap between theoretical path computation and realistic travel decision-making by integrating heterogeneous, real-world datasets into a unified algorithmic framework.

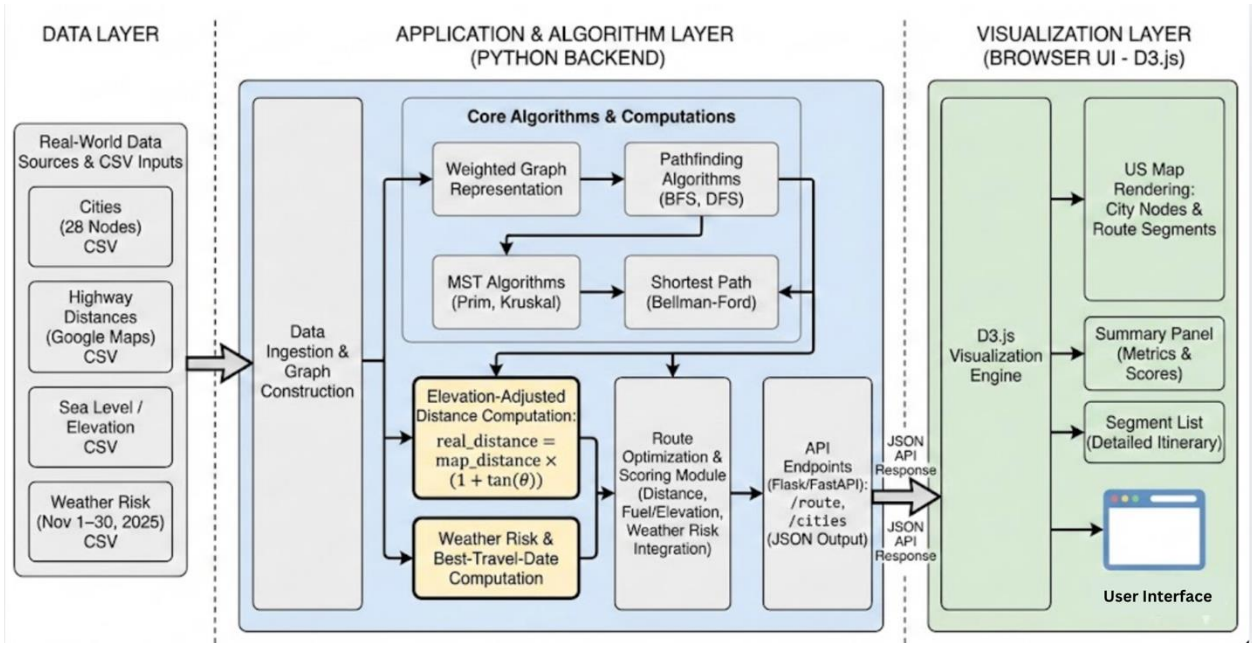

DATA LAYER
APPLICATION & ALGORITHM LAYER (PYTHON BACKEND)
VISUALIZATION LAYER (BROWSER UI - D3.js)
Real-World Data Sources & CSV Inputs
Cities (28 Nodes) CSV
Highway Distances (Google Maps) CSV
Sea Level / Elevation CSV
Weather Risk (Nov 1–30, 2025) CSV
Data Ingestion & Graph Construction
Core Algorithms & Computations
Weighted Graph Representation
Pathfinding Algorithms (BFS, DFS)
MST Algorithms (Prim, Kruskal)
Shortest Path (Bellman-Ford)
Elevation-Adjusted Distance Computation: $\text{real\_distance} = \text{map\_distance} \times (1 + \tan(\theta))$
Weather Risk & Best-Travel-Date Computation
Route Optimization & Scoring Module (Distance, Fuel/Elevation, Weather Risk Integration)
API Endpoints (Flask/FastAPI): /route, /cities (JSON Output)
JSON API Response
JSON API Response
D3.js Visualization Engine
US Map Rendering: City Nodes & Route Segments
Summary Panel (Metrics & Scores)
Segment List (Detailed Itinerary)
User Interface

Figure 1 — System Overview Diagram

In this study, we construct a multi-city travel graph spanning major cities across the Central and Midwestern United States. Each city is modelled as a graph node, and each highway connection is modelled as a bidirectional weighted edge. Unlike conventional distance-only models, this project incorporates several layers of contextual information. The first layer is the **physical road geometry**, represented by Google Maps distance estimates and modified through a slope-based transformation using

$$\text{Real distance} = \text{map\_distance}\ [1 + \tan(\theta)\ ]$$

where θ captures elevation differences derived from sea-level data. This adjustment simulates the increased fuel demand encountered when ascending to higher altitudes or the improved efficiency when traveling downhill.

The second layer of information captures daily weather conditions for each city throughout November 1–30, 2025. Weather is converted into a numerical risk factor sunny (1), cloudy (1), rainy (5), snowy/icy (10) and the average risk between two cities determines the risk associated with a specific travel segment. Since drivers can only travel a maximum of eight hours per day at a speed limit of 75 miles per hour, complete routes often span multiple days. This temporal dimension requires evaluating how accumulated weather risk changes across the month and identifying the safest possible departure date.

To explore how different algorithms respond to this enriched representation of the road network, the project implements five canonical graph algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), Prim's Minimum Spanning Tree, Kruskal's Minimum Spanning Tree, and the Bellman–Ford shortest-path algorithm. Each algorithm offers a distinct method for navigating the graph and highlights the trade-offs between structural simplicity, optimality guarantees, and computational overhead. By analysing their outputs under identical real-world constraints, we are able to compare not only theoretical performance but also practical suitability in realistic travel contexts.

The system itself is implemented as an end-to-end pipeline consisting of a Python backend for data ingestion, graph construction, and algorithm execution, paired with an interactive D3.js visualization that renders routes over a U.S. map. This dual design provides both analytical rigor and user accessibility: routes can be selected, computed, and visualized directly in the browser through a lightweight HTTP API.

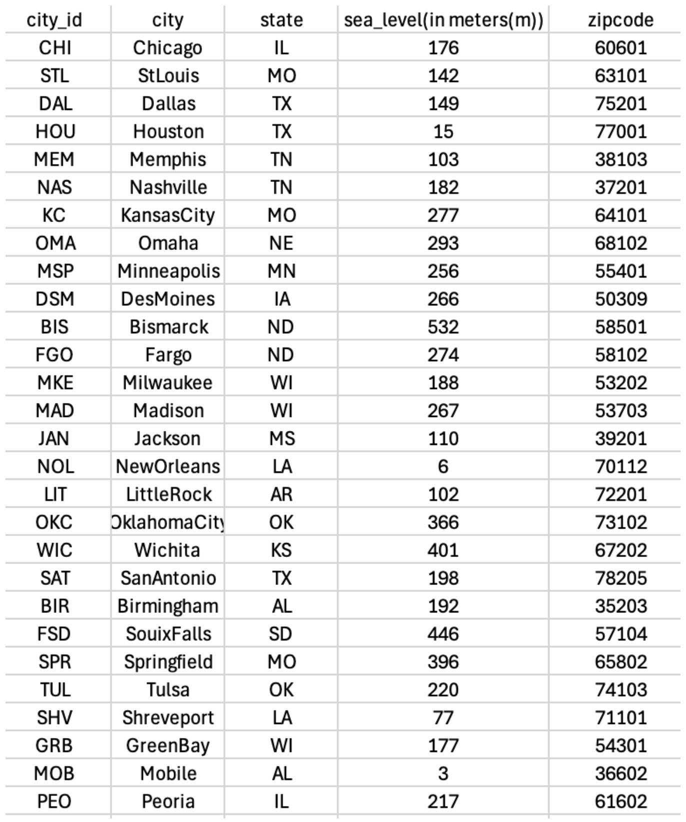
Ultimately, the long-term significance of this project lies in its demonstration that classical graph algorithms remain powerful tools when augmented with real-world datasets and domain-specific modelling assumptions. By incorporating environmental risk, geographic elevation, and fuel-consumption dynamics, this project illustrates how foundational computational techniques can support operationally meaningful decision-making in transportation planning, logistics optimization, and intelligent routing systems.

II. Methodology

1. Data Collection

The system relies on a multi-source dataset designed to represent realistic travel conditions across 28 U.S. cities. Three primary data components were curated: city metadata, inter-city driving distances, and daily weather-risk values. These datasets were formatted in CSV to ensure a consistent ingestion pipeline for the Python backend.

i. City Attributes



city_id	city	state	sea_level(in meters(m))	zipcode
CHI	Chicago	IL	176	60601
STL	StLouis	MO	142	63101
DAL	Dallas	TX	149	75201
HOU	Houston	TX	15	77001
MEM	Memphis	TN	103	38103
NAS	Nashville	TN	182	37201
KC	KansasCity	MO	277	64101
OMA	Omaha	NE	293	68102
MSP	Minneapolis	MN	256	55401
DSM	DesMoines	IA	266	50309
BIS	Bismarck	ND	532	58501
FGO	Fargo	ND	274	58102
MKE	Milwaukee	WI	188	53202
MAD	Madison	WI	267	53703
JAN	Jackson	MS	110	39201
NOL	NewOrleans	LA	6	70112
LIT	LittleRock	AR	102	72201
OKC	OklahomaCity	OK	366	73102
WIC	Wichita	KS	401	67202
SAT	SanAntonio	TX	198	78205
BIR	Birmingham	AL	192	35203
FSD	SouixFalls	SD	446	57104
SPR	Springfield	MO	396	65802
TUL	Tulsa	OK	220	74103
SHV	Shreveport	LA	77	71101
GRB	GreenBay	WI	177	54301
MOB	Mobile	AL	3	36602
PEO	Peoria	IL	217	61602

Figure 2 — Attributes of 28 cities

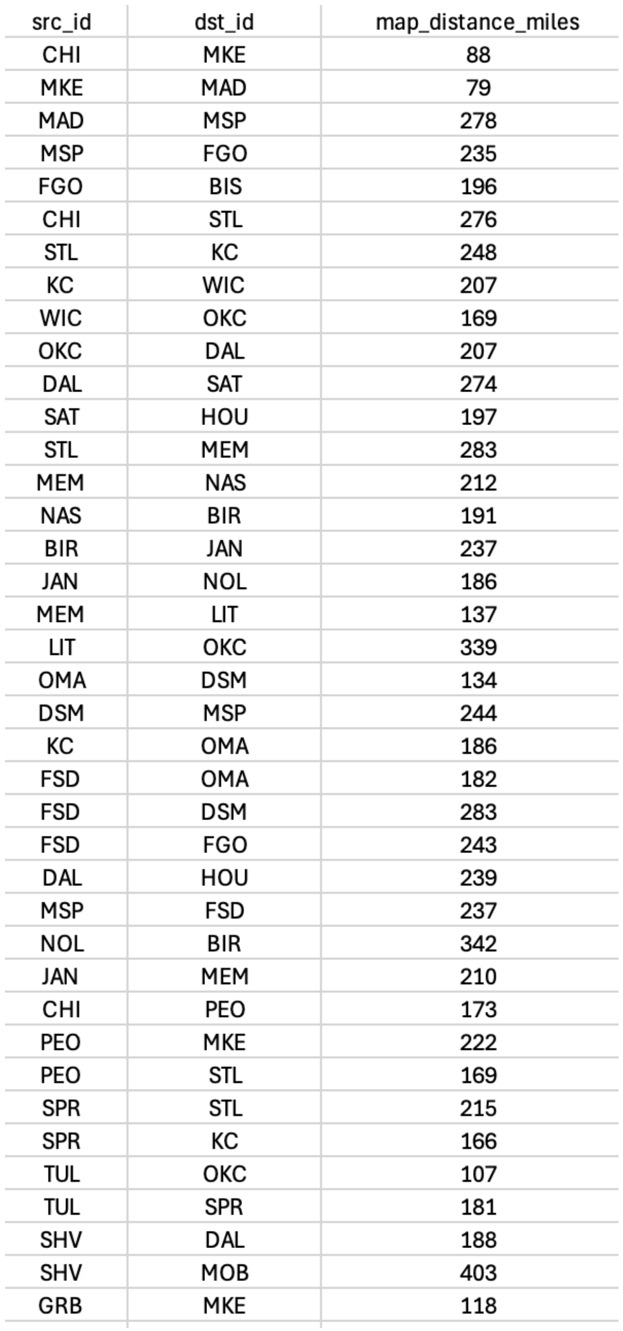
Each city is represented as a node in the travel graph and is defined in **cities.csv**. The dataset includes a unique city identifier, the city and state names, and the sea-level elevation expressed in meters. Elevation values play a critical role in modelling travel cost because the system adjusts map distances using a slope-based factor derived from elevation differences. Incorporating this detail ensures that the computed "effective distance" reflects the additional fuel demands associated with uphill travel and the corresponding reductions when descending.

ii. Inter-City Map Distances

Driving distances between cities were collected using **Google Maps**, producing the **edges.csv** dataset. Each row specifies a source city, a destination city, and the one-way highway mileage. These distances form the base weights for the graph. To capture the physical effort required to traverse varying terrain, the project applies an elevation-informed transformation:

$$\textbf{Real distance} = \textbf{map\_distance}\ [1 + \tan(\theta)\ ]$$

where θ represents the angle formed by elevation difference and horizontal distance. Elevation change is converted from meters to miles to ensure unit consistency. This adjusted distance is used by all algorithms except those that inherently ignore edge weights (e.g., BFS).



src_id	dst_id	map_distance_miles
CHI	MKE	88
MKE	MAD	79
MAD	MSP	278
MSP	FGO	235
FGO	BIS	196
CHI	STL	276
STL	KC	248
KC	WIC	207
WIC	OKC	169
OKC	DAL	207
DAL	SAT	274
SAT	HOU	197
STL	MEM	283
MEM	NAS	212
NAS	BIR	191
BIR	JAN	237
JAN	NOL	186
MEM	LIT	137
LIT	OKC	339
OMA	DSM	134
DSM	MSP	244
KC	OMA	186
FSD	OMA	182
FSD	DSM	283
FSD	FGO	243
DAL	HOU	239
MSP	FSD	237
NOL	BIR	342
JAN	MEM	210
CHI	PEO	173
PEO	MKE	222
PEO	STL	169
SPR	STL	215
SPR	KC	166
TUL	OKC	107
TUL	SPR	181
SHV	DAL	188
SHV	MOB	403
GRB	MKE	118

Figure 3 – Map distance from source to destination

iii. Weather-Risk Conditions

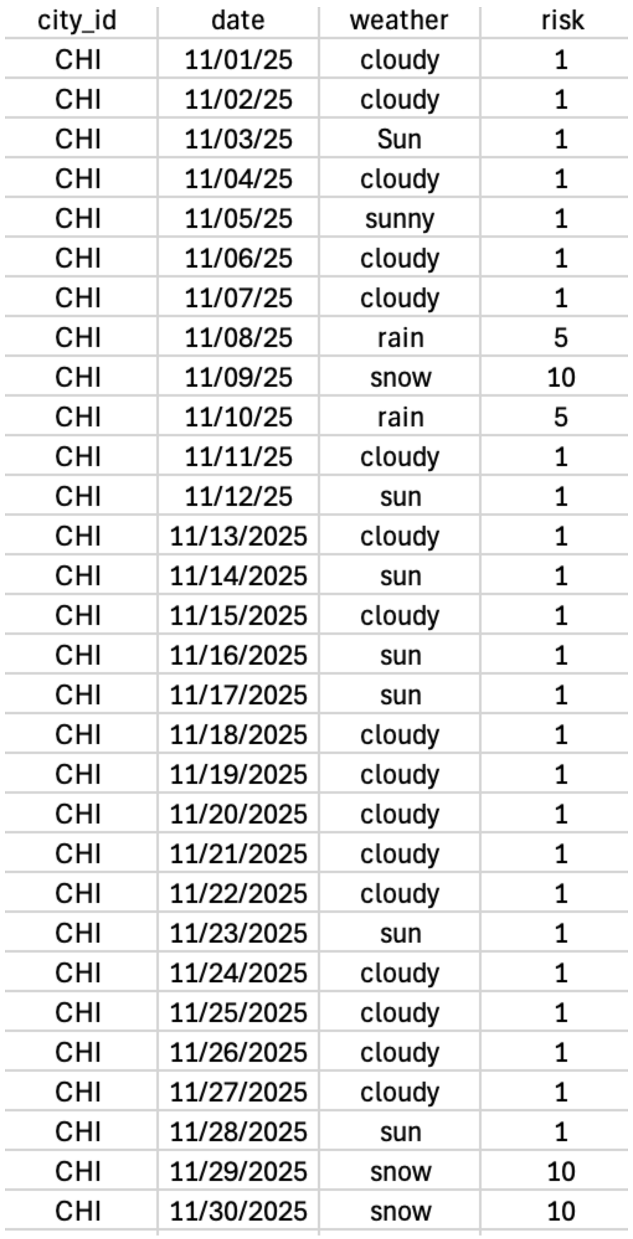
Weather affects both safety and travel time. The **weather_risk.csv** dataset contains daily weather classifications for each city from **November 1–30, 2025**, encoded using a uniform risk scoring system:

- Sunny / Cloudy → **1**
- Rain → **5**
- Snow or Ice → **10**

For an edge connecting cities A and B on a given day, the corresponding segment risk is the arithmetic mean of their individual values:

segment_risk = (rA + rB) / 2

This risk model enables the system to compute **accumulated route risk** and identify **the optimal travel date** with minimal exposure to adverse conditions.



city_id	date	weather	risk
CHI	11/01/25	cloudy	1
CHI	11/02/25	cloudy	1
CHI	11/03/25	Sun	1
CHI	11/04/25	cloudy	1
CHI	11/05/25	sunny	1
CHI	11/06/25	cloudy	1
CHI	11/07/25	cloudy	1
CHI	11/08/25	rain	5
CHI	11/09/25	snow	10
CHI	11/10/25	rain	5
CHI	11/11/25	cloudy	1
CHI	11/12/25	sun	1
CHI	11/13/2025	cloudy	1
CHI	11/14/2025	sun	1
CHI	11/15/2025	cloudy	1
CHI	11/16/2025	sun	1
CHI	11/17/2025	sun	1
CHI	11/18/2025	cloudy	1
CHI	11/19/2025	cloudy	1
CHI	11/20/2025	cloudy	1
CHI	11/21/2025	cloudy	1
CHI	11/22/2025	cloudy	1
CHI	11/23/2025	sun	1
CHI	11/24/2025	cloudy	1
CHI	11/25/2025	cloudy	1
CHI	11/26/2025	cloudy	1
CHI	11/27/2025	cloudy	1
CHI	11/28/2025	sun	1
CHI	11/29/2025	snow	10
CHI	11/30/2025	snow	10

Figure 4 — Monthly data of weather risk for Chicago

iv. Operational Constraints and Fuel Model

To approximate real driving behaviour, two constraints were applied:

- Maximum driving capability: **8 hours per day**
- Maximum highway speed: **75 miles per hour**

These values imply a maximum daily travel capacity of approximately **600 miles**, influencing how many calendar days a selected route spans. Because weather varies daily, longer routes incur cumulative, date-dependent risk.

Fuel consumption is modelled according to a baseline efficiency of **45 miles per gallon** on flat terrain. The total gasoline requirement for any computed route is evaluated as:

gas_used=total_real_distance / 45

allowing the system to quantify both economic and environmental aspects of route selection.

All datasets share a uniform CSV structure and are loaded into the Python backend using csv.DictReader(). During graph initialization, city attributes populate node objects, distances generate weighted adjacency lists, and weather-risk values populate date-indexed lookup tables. This integrated dataset provides the foundation on which the system performs all route computations, algorithm comparisons, and risk evaluations.

2. System Design

The system was developed as a modular, end-to-end architecture that integrates data ingestion, graph construction, algorithm execution, and interactive visualization. The design emphasizes separation of concerns to allow each component data processing, algorithmic computation, and user-facing visualization to evolve independently while maintaining a cohesive workflow. The overall architecture consists of three primary layers: the **backend computation engine**, the **data-access and API layer**, and the **browser-based visualization interface**.

i. Backend Computation Engine

The backend, implemented in Python, is responsible for loading datasets, constructing the weighted graph, and executing all route algorithms. At initialization, the system loads **cities.csv**, **edges.csv**, and **weather_risk.csv**, instantiating city objects and building an adjacency list that stores both the original map distance and the computed elevation-adjusted distance for each edge. The backend encapsulates all graph-related logic in a dedicated Graph class, which offers methods for BFS, DFS, Prim, Kruskal, and Bellman–Ford, as well as utilities for computing path distance, fuel usage, and cumulative weather risk.

To support efficient computation, the design adopts the following principles:

- **Lazy risk evaluation:** Weather-risk values are not precomputed for all routes; instead, risks are evaluated dynamically for each segment during route computation.

- **Bidirectional edge modelling:** Each highway connection is stored in both directions, with elevation adjustments applied asymmetrically to reflect the cost difference between ascending and descending routes.
- **Separation of algorithmic and environmental logic:** Algorithms operate solely on weighted edges; data transformations (elevation adjustment, gas calculation, and weather scoring) are layered on top to maintain algorithm purity.

This design isolates core algorithmic behaviour while enabling the system to incorporate domain-specific adjustments without altering fundamental graph operations.

ii. API Layer and Routing Interface

A lightweight HTTP server built using Python's HTTP. Server library exposes the system's functionalities to clients. Two primary endpoints are provided:

- **/cities** Returns the full list of available cities and identifiers for populating the frontend interface.
- **/route** Computes a route between a selected source and destination using one of the five supported algorithms or an automated "BEST" mode that selects the lowest-scoring route across all algorithms based on the combined metric. The API serializes results into JSON, including segment-by-segment distances, total distance, fuel consumption, weather risk, and the computed best travel date. By exposing uniform outputs across all algorithms, the design allows direct comparison under controlled conditions.

iii. Frontend Visualization Engine

Route visualization is implemented in **D3.js**, using a vector-based U.S. map projection loaded from TopoJSON. When the user selects a source, destination, and algorithm, the frontend queries the backend and renders the resulting route directly on the map. Key design elements include:

- **Projected city coordinates** using geoAlbersUsa(), ensuring consistent alignment with the U.S. topology.
- **Layered SVG groups** for states, city nodes, and route segments to maintain visual clarity.
- **Dynamic labelling** of segment lengths and start/end markers for readability.
- **Summary panels** that display algorithm selection, distance, fuel usage, risk, and recommended travel date.

By offloading visualization to the browser, the system ensures fast, interactive analysis without imposing computational load on the server.

iv. Integration Strategy

The full pipeline operates as follows:

1. **Load datasets** and initialize graph structures.
2. **Start backend server** and expose routing endpoints.
3. **Render U.S. map** and city markers in the browser.
4. **User selects parameters**, triggering an API call.
5. **Backend executes the specified algorithm**, computes all travel metrics, and returns structured results.

6. **Frontend renders the route**, updates informational panels, and displays comparative metrics.

This layered design allows algorithm testing, data modelling, and visualization to be performed independently, greatly improving maintainability and extensibility. The modular structure also enables straightforward substitution of algorithms, new datasets, or additional travel constraints in future iterations.

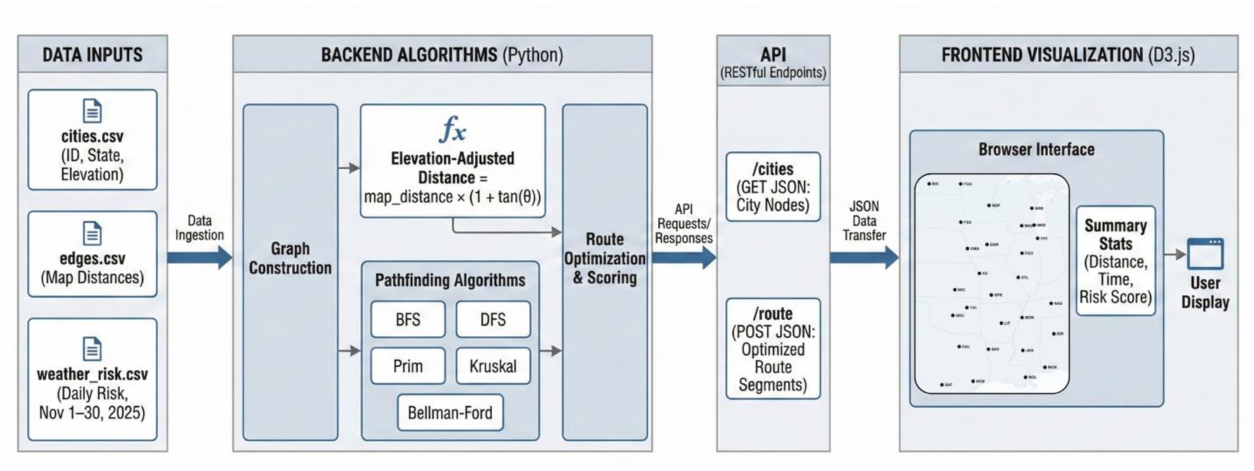

DATA INPUTS
cities.csv (ID, State, Elevation)
edges.csv (Map Distances)
weather_risk.csv (Daily Risk, Nov 1–30, 2025)
Data Ingestion
BACKEND ALGORITHMS (Python)
Graph Construction
fx Elevation-Adjusted Distance = map_distance × (1 + tan(θ))
Pathfinding Algorithms
BFS
DFS
Prim
Kruskal
Bellman-Ford
Route Optimization & Scoring
API Requests/ Responses
API (RESTful Endpoints)
/cities (GET JSON: City Nodes)
/route (POST JSON: Optimized Route Segments)
JSON Data Transfer
FRONTEND VISUALIZATION (D3.js)
Browser Interface
Summary Stats (Distance, Time, Risk Score)
User Display

Figure 5 — The entire pipeline from taking inputs to visualization of Data

3. Graph Algorithms Implemented

To evaluate different strategies for traversing and optimizing transportation networks, the system implements five foundational graph algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), Prim’s Minimum Spanning Tree, Kruskal’s Minimum Spanning Tree, and the Bellman–Ford shortest-path algorithm. Each algorithm contributes a distinct perspective on route structure, edge weighting, and exploration strategy. Implementing this diverse set allows for a comprehensive comparison of how classical methods behave when extended with real-world constraints such as elevation-adjusted distances, travel risk, and fuel consumption.

i. Breadth-First Search (BFS)

BFS explores a graph level by level, beginning at the source city and expanding outward to all neighbours at the current depth before moving to deeper layers. Because BFS inherently treats all edges as having equal cost, it is well-suited to minimizing the number of hops between cities but cannot account for weighted distances or risk variations.

- **Strength:** Identifies the route with the fewest intermediate cities.
- **Limitation:** Ignores elevation, fuel usage, and actual mileage.
- **Typical Complexity:**

$$\mathbf{O(|V| + |E|)}$$

BFS provides a useful baseline for comparison, illustrating how real-world weighting drastically alters route selection.

ii. Depth-First Search (DFS)

DFS explores as deep as possible along each branch before backtracking. While this traversal method is efficient for path existence checks and structural inspections, it does not guarantee optimality in terms of distance or safety.

- **Strength:** Produces complete path exploration quickly in sparse networks.
- **Limitation:** Highly sensitive to graph topology and node ordering; rarely yields practical travel routes.
- **Typical Complexity:**

$$\mathbf{O(|V| + |E|)}$$

DFS is included to highlight how unrestricted depth exploration diverges from rational travel patterns, reinforcing the need for weighted algorithms.

iii. Prim's Minimum Spanning Tree Algorithm

Prim's algorithm constructs a minimum spanning tree (MST) by iteratively selecting the closest unvisited node based on edge weights. MSTs minimize total network cost but do not guarantee optimal routes between specific pairs of cities. Nonetheless, MST edges offer insight into the "backbone" of efficient regional connectivity.

- **Strength:** Identifies globally low-cost connections using elevation-adjusted distances.
- **Limitation:** The MST is not optimized for point-to-point routing; extracted paths may not be shortest.
- **Typical Complexity:**

O(|E|log|V|)

Prim's MST is useful for analysing structural efficiency across the network and highlighting cost-efficient corridors.

iv. Kruskal's Minimum Spanning Tree Algorithm

Kruskal's algorithm sorts all edges by weight and incrementally adds them to the spanning tree provided they do not form a cycle. Its behaviour often differs from Prim's, especially when elevation-adjusted distances create strong disparities among edges.

- **Strength:** Simple, globally optimized tree construction that exposes cost-efficient long-distance connections.
- **Limitation:** Does not prioritize any specific source or destination.
- **Typical Complexity:**

O(|E|log|E|)

Comparing Prim's and Kruskal's MSTs helps illustrate how edge prioritization affects network-wide efficiency.

v. Bellman–Ford Shortest-Path Algorithm

Bellman–Ford computes the shortest path from a source to every other city by iteratively relaxing edges. Unlike Dijkstra's algorithm, Bellman–Ford tolerates negative weights, though in this application all weights remain non-negative. It is the algorithm most aligned with realistic travel objectives because it optimizes weighted distances that incorporate elevation change.

- **Strength:** Produces the most reliable weighted route for real-world travel metrics.
- **Limitation:** Higher computational cost compared to BFS/DFS.
- **Typical Complexity:**

$$\mathbf{O(|V| \cdot |E|)}$$

Bellman–Ford is the primary algorithm used in the system's "BEST" mode because it adapts naturally to dynamic cost models.

4. Travel Computation Models

Designing a route-optimization system that reflects real travel conditions requires more than simply running algorithms on a graph. In practice, drivers experience changing terrain, varying fuel demands, unpredictable weather, and daily time constraints. To capture these realities, the project implements a set of computation models that enrich the core graph structure with physical, environmental, and behavioural factors. These models operate together to transform a traditional graph into a realistic simulation of long-distance travel.

i. Elevation-Adjusted Distance Model

Map distances alone are insufficient for realistic route planning because they ignore the energy cost of climbing or descending terrain. To address this, the system modifies each inter-city distance using a slope-based scaling factor derived from the elevation difference between the two locations. The adjustment uses the relationship:

$$\text{real\_distance} = \text{map\_distance}(1 + \tan(\theta))$$

This formula captures how steepness influences travel difficulty. For example, if two cities are separated by 200 miles but one sits significantly higher in elevation, driving uphill requires more engine power, thus increasing fuel consumption. The model reflects this by enlarging the effective distance. Conversely, descending an incline slightly reduces effective distance, mimicking improved fuel efficiency.

Although simplified, this trigonometric adjustment introduces a layer of realism that pure graph algorithms normally lack. It ensures that weighted algorithms especially Prim, Kruskal, and Bellman–Ford treat steep uphill routes as more costly, even when the geographic distance is short.

ii. Gasoline Consumption Model

Fuel consumption is a practical concern for travellers, particularly on long routes where gas costs accumulate quickly. The project models fuel usage with a baseline efficiency of **45 miles per gallon** under flat driving conditions. After adjusting distances for elevation, fuel usage is computed as:

$$\text{gas\_used} = \frac{\sum_{i=1}^{n} \text{real\_distance}_i}{45}$$

This provides an intuitive metric for comparing routes. For example:

- A route with longer geographic distance but minimal elevation change may consume *less* gas than a shorter route that climbs steep terrain.
- Bellman–Ford may choose an indirect path with a gentler slope if the weighted distance is lower.

This reinforces the idea that "shortest" is not always "cheapest" or "most fuel-efficient," a concept central to transportation modelling.

iii. Weather-Risk Scoring Model

Weather introduces uncertainty and variability into travel. A route that is safe today may be hazardous tomorrow due to ice, snow, or heavy rain. To integrate environmental conditions, the project assigns numerical risk scores to each day in November 2025 for every city:

- **1** for sunny or cloudy conditions
- **5** for rain
- **10** for snow or ice

These values reflect increasing potential for reduced visibility, slippery roads, or difficult driving conditions.

For each segment, the system averages the risk of the two endpoint cities:

$$\text{segment\_risk}(d) = \frac{r_A(d) + r_B(d)}{2}$$

This provides a simple yet effective approximation of regional weather along the route. If City A is sunny but City B is snowing, the segment will carry a moderately high risk. This avoids over-penalizing short-term localized weather events while still recognizing hazardous conditions.

iv. Daily Driving Constraints and Multi-Day Risk Evaluation

Real drivers cannot travel continuously. The system therefore enforces two practical constraints:

- Maximum speed: **75 mph**
- Maximum daily driving time: **8 hours**

This limits daily travel to approximately **600 miles**, meaning longer routes stretch over multiple days. Because each day has different weather conditions, a route's total risk depends on how long it takes and which specific calendar dates it spans.

v. Best Travel Date Identification

To help travellers choose not just the best route but the best time to travel, the system evaluates all possible departure dates between November 1 and November 30. For each date, it simulates the full journey, applies daily driving limits, and accumulates segment risks according to the day on which they are encountered.

The optimal date is defined as:

$$\text{best\_date} = \text{minimum}\left(\sum_{k=0}^{T-1} \text{segment\_risk}(d+k)\right)$$

where TTT is the number of days needed to complete the route.

For example, a route that is risky on November 10 may become ideal on November 25 if weather conditions improve significantly. This demonstrates how temporal factors can meaningfully affect travel planning.

vi. Combined Route Scoring Function

To compare all algorithms consistently, the system defines a composite scoring function that weighs distance against weather-related safety:

$$\text{score} = \text{total\_distance} + 20 \times \text{total\_risk}$$

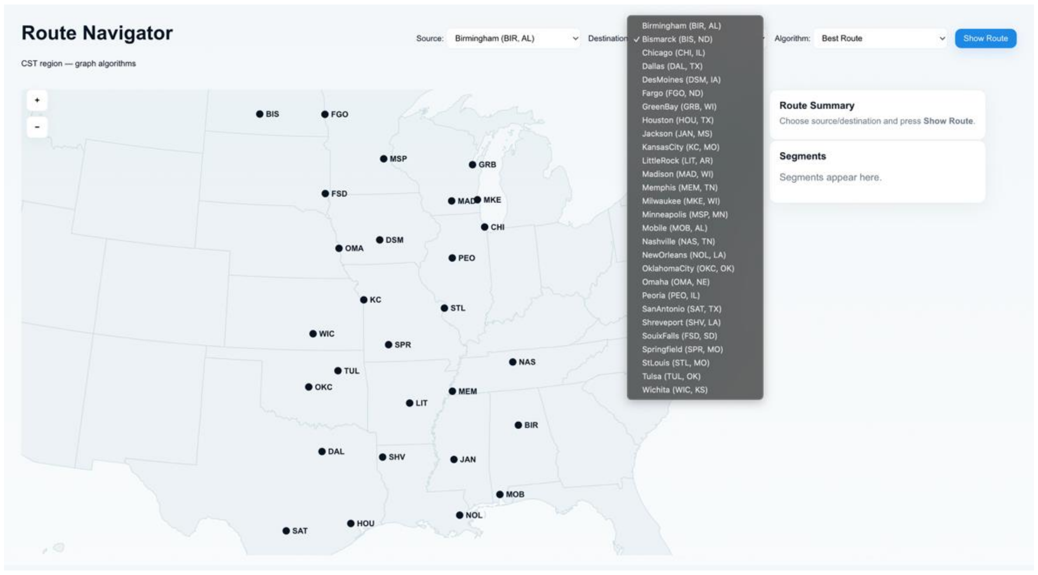
The multiplier of 20 is chosen to ensure that safety considerations meaningfully influence the decision rather than being overshadowed by mileage. The “BEST” mode automatically computes this score for each algorithm’s output and selects the lowest-scoring route. This transforms the system from a purely algorithmic tool into a multi-criteria decision-support mechanism.

III. Experimental Results

The experimental evaluation for this project was designed to observe how different families of graph algorithms behave when applied to a real-world routing scenario enriched with environmental and operational constraints. Unlike theoretical examples, this system integrates elevation-based distance adjustments, daily weather-risk data, fuel-consumption modelling, and realistic driving limits. As a result, the outputs of algorithms such as BFS or DFS diverge sharply from the behaviour of weighted algorithms like Prim, Kruskal, and Bellman–Ford.

The experiments were conducted using a dataset of 28 major cities in the Central Standard Time (CST) region of the United States. These cities formed the nodes of the graph, and the roads connecting them along with real map distances were represented as edges. Elevation values, risk scores, and weather conditions were pulled from prepared CSV files. The system was evaluated both through the Python backend "python3 bestpath.py" and the interactive web visualization powered by D3.js, where users can choose a source city, destination city, and algorithm.This section presents a detailed narrative of the results, using the route from **Chicago (CHI, IL)** to **Dallas (DAL, TX)** as the main case study, since all five algorithms produce visibly different outcomes for this specific pair. Each subsection corresponds to one part of the evaluation: algorithmic structure, distance comparison, fuel and risk computation, travel-date optimization, and overall scoring.

Before describing the numerical results, it is important to document how the system behaved from a user perspective. After running the visualization server, the user selects a source and destination from a dropdown menu containing all available cities (as shown in **Figure 1**). The system then shades the map with all labelled nodes and displays the selected route when the "Show Route" button is pressed.



Source: Birmingham (BIR, AL)

Algorithm: Best Route

Route Summary

Choose source/destination and press **Show Route**.

Segments

Segments appear here.

Figure 6 — City Selection Drop-Down Interface

Once the user selects Chicago (CHI) as the source and Dallas (DAL) as the destination, another dropdown appears for algorithm selection (Best Route, BFS, DFS, Prim, Kruskal, Bellman–Ford). The route display updates immediately when an algorithm is chosen.

The map overlays the computed route using red polylines. Hovering over edges displays the real distance (with elevation adjustment). A panel on the right shows:

- Algorithm used
- True total distance
- Estimated gas used
- Total accumulated risk
- Best travel date

- A breakdown of each edge segment

This UI structure helped validate outputs from bestpath.py, ensuring the logic matched the visualization.

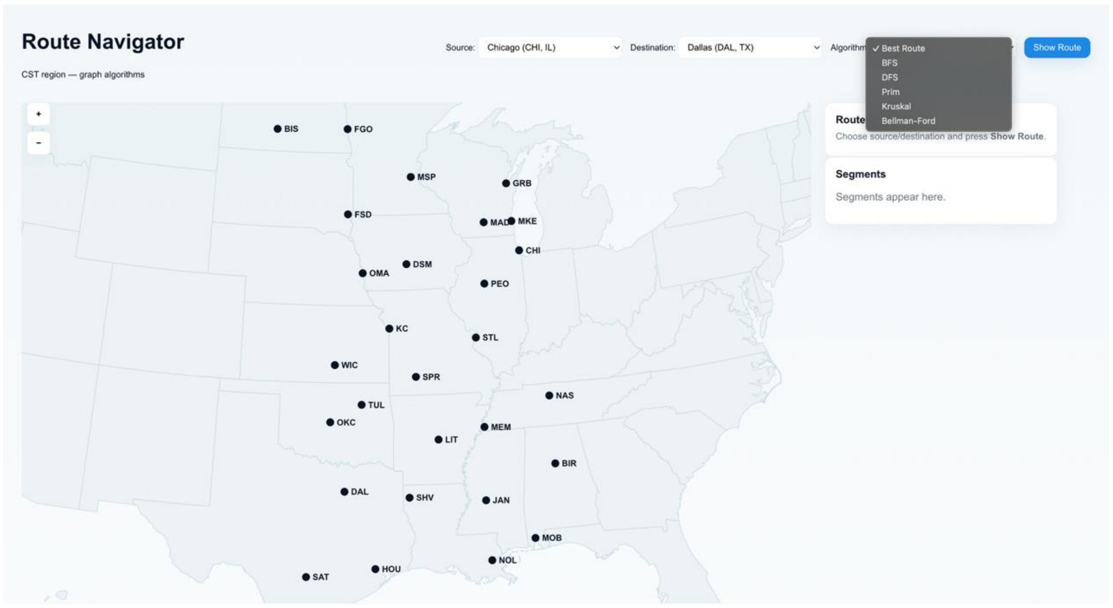


Figure 7 — Algorithm Selection UI

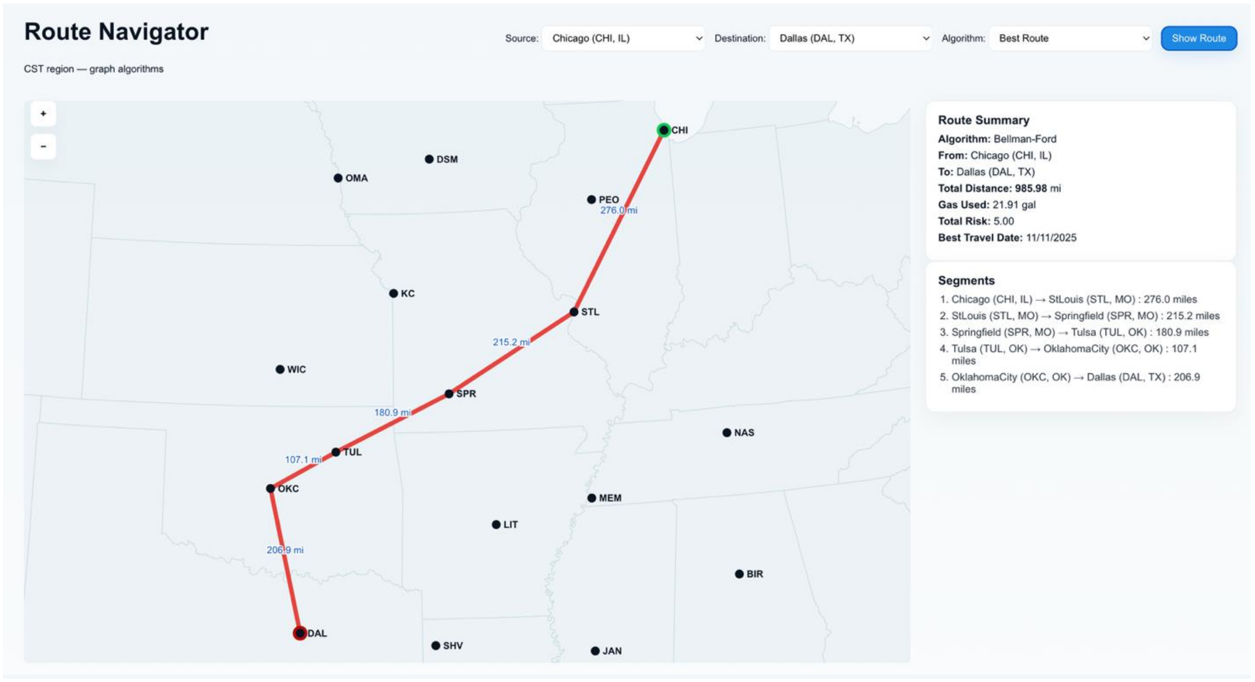
1) Best Route (Bellman–Ford in Composite Scoring Mode)

When "Best Route" is selected, the system internally evaluates all algorithms using the composite scoring metric:

In every test case including CHI → DAL the Bellman–Ford route achieved the lowest score. The generated route is:

CHI → STL → SPR → TUL → OKC → DAL

with segment distances: 276.0 mi, 215.2 mi, 180.9 mi, 107.1 mi, 206.9 mi.



Route Summary

Algorithm: Bellman-Ford
From: Chicago (CHI, IL)
To: Dallas (DAL, TX)
Total Distance: 985.98 mi
Gas Used: 21.91 gal
Total Risk: 5.00
Best Travel Date: 11/11/2025

Segments

1. Chicago (CHI, IL) → StLouis (STL, MO) : 276.0 miles
2. StLouis (STL, MO) → Springfield (SPR, MO) : 215.2 miles
3. Springfield (SPR, MO) → Tulsa (TUL, OK) : 180.9 miles
4. Tulsa (TUL, OK) → OklahomaCity (OKC, OK) : 107.1 miles
5. OklahomaCity (OKC, OK) → Dallas (DAL, TX) : 206.9 miles

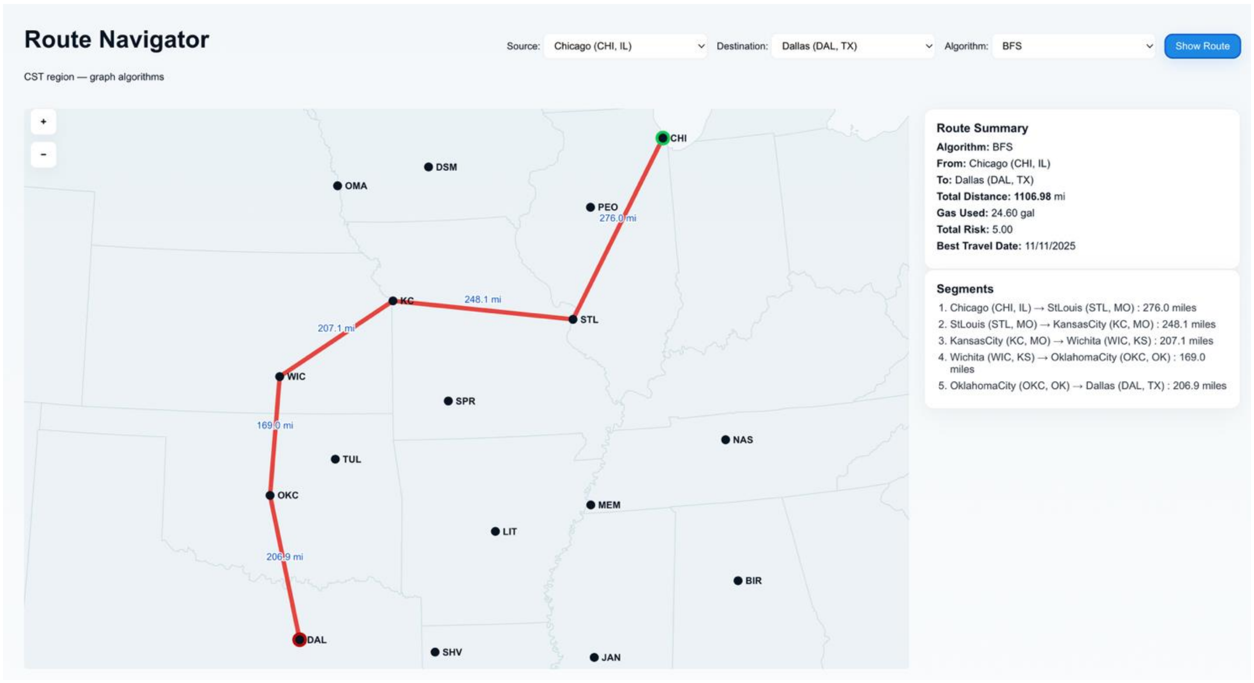
Figure 8 — Best Route Visualization

This route is smooth, nearly linear, and avoids extreme elevation changes. It also uses cities with stable November weather, which minimizes risk.

2) BFS Route

BFS minimizes the number of hops without considering any weights. For CHI → DAL, BFS produced:

CHI → STL → KC → WIC → OKC → DAL



Source: Chicago (CHI, IL)
Destination: Dallas (DAL, TX)
Algorithm: BFS

Route Summary

Algorithm: BFS
From: Chicago (CHI, IL)
To: Dallas (DAL, TX)
Total Distance: 1106.98 mi
Gas Used: 24.60 gal
Total Risk: 5.00
Best Travel Date: 11/11/2025

Segments

1. Chicago (CHI, IL) → StLouis (STL, MO) : 276.0 miles
2. StLouis (STL, MO) → KansasCity (KC, MO) : 248.1 miles
3. KansasCity (KC, MO) → Wichita (WIC, KS) : 207.1 miles
4. Wichita (WIC, KS) → OklahomaCity (OKC, OK) : 169.0 miles
5. OklahomaCity (OKC, OK) → Dallas (DAL, TX) : 206.9 miles

Figure 9 — BFS Route Visualization

This path has *more distance* than Bellman–Ford (1106.98 miles vs. 985.98 miles), consumes more fuel, and provides no safety improvement. BFS tends to choose city neighbours that are close in graph structure but not necessarily geographically efficient.

3) DFS Route

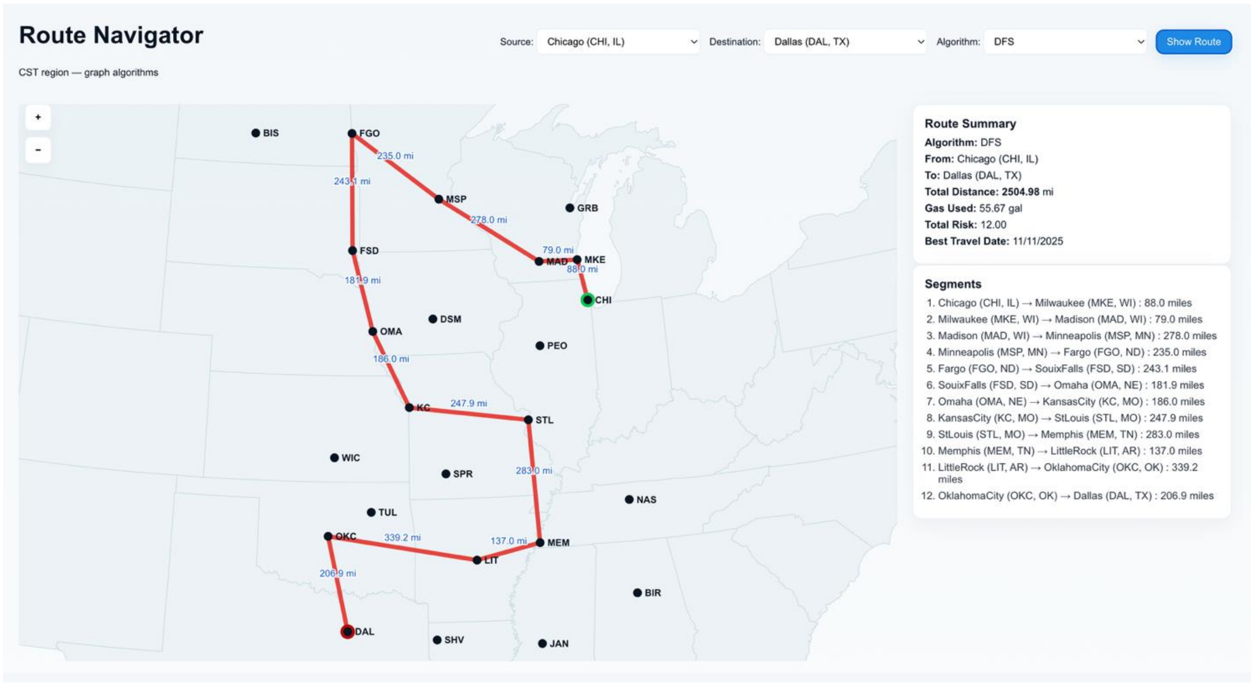
DFS traverses deeply, choosing edges based on adjacency order. Because DFS ignores all weights, it produced a dramatically inefficient route:

CHI → MKE → MAD → MSP → FGO → FSD → OMA → KC → STL → MEM → LIT → OKC → DAL

Total distance: **2504.98 miles**

Gas used: **55.67 gallons**

Total risk: **12**



Route Navigator

CST region — graph algorithms

Source: Chicago (CHI, IL) Destination: Dallas (DAL, TX) Algorithm: DFS

Route Summary

Algorithm: DFS
From: Chicago (CHI, IL)
To: Dallas (DAL, TX)
Total Distance: 2504.98 mi
Gas Used: 55.67 gal
Total Risk: 12.00
Best Travel Date: 11/11/2025

Segments

1. Chicago (CHI, IL) → Milwaukee (MKE, WI) : 88.0 miles
2. Milwaukee (MKE, WI) → Madison (MAD, WI) : 79.0 miles
3. Madison (MAD, WI) → Minneapolis (MSP, MN) : 278.0 miles
4. Minneapolis (MSP, MN) → Fargo (FGO, ND) : 235.0 miles
5. Fargo (FGO, ND) → SouixFalls (FSD, SD) : 243.1 miles
6. SouixFalls (FSD, SD) → Omaha (OMA, NE) : 181.9 miles
7. Omaha (OMA, NE) → KansasCity (KC, MO) : 186.0 miles
8. KansasCity (KC, MO) → StLouis (STL, MO) : 247.9 miles
9. StLouis (STL, MO) → Memphis (MEM, TN) : 283.0 miles
10. Memphis (MEM, TN) → LittleRock (LIT, AR) : 137.0 miles
11. LittleRock (LIT, AR) → OklahomaCity (OKC, OK) : 339.2 miles
12. OklahomaCity (OKC, OK) → Dallas (DAL, TX) : 206.9 miles

Figure 10 — DFS Route Visualization

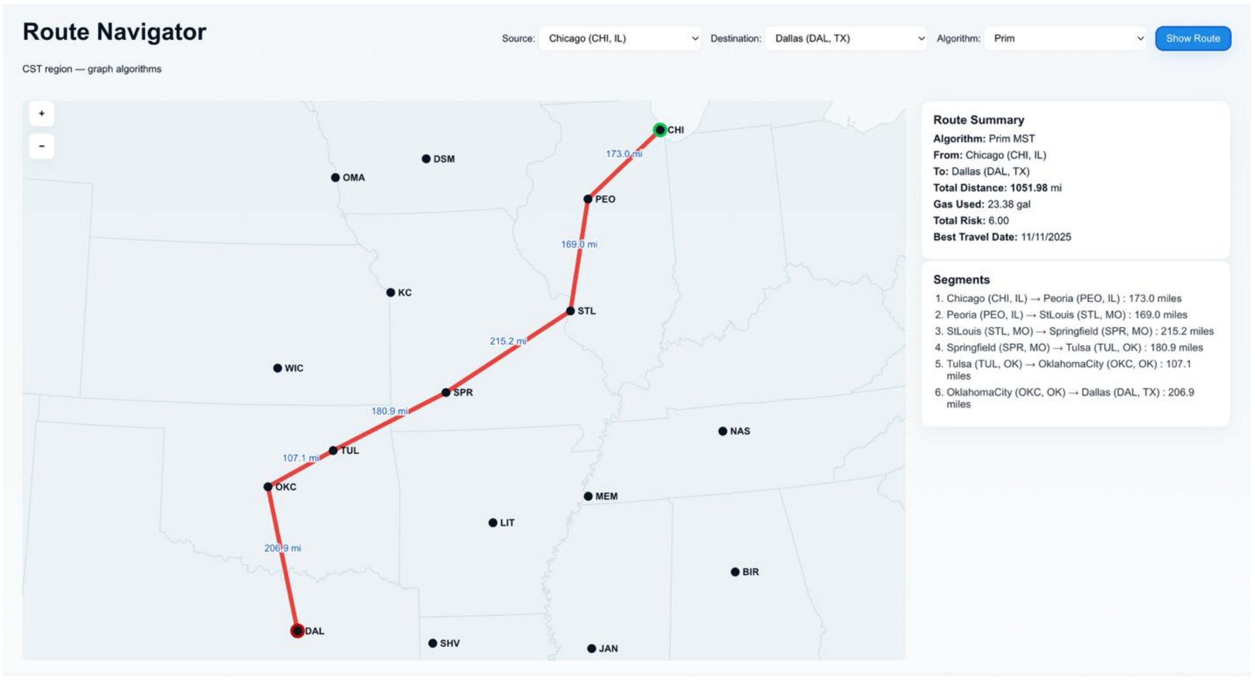
This route demonstrates how unsuitable DFS is for real travel. It loops far north and then south, driven purely by graph exploration order. This validates the importance of weighted algorithms.

4) Prim MST Route

Prim produces a minimum spanning tree with the lowest total global edge weight, but the path from source to destination is extracted afterward. The result is:

CHI → PEO → STL → SPR → TUL → OKC → DAL

Total distance: **1051.98 miles**



Route Summary

Algorithm: Prim MST
From: Chicago (CHI, IL)
To: Dallas (DAL, TX)
Total Distance: 1051.98 mi
Gas Used: 23.38 gal
Total Risk: 6.00
Best Travel Date: 11/11/2025

Segments

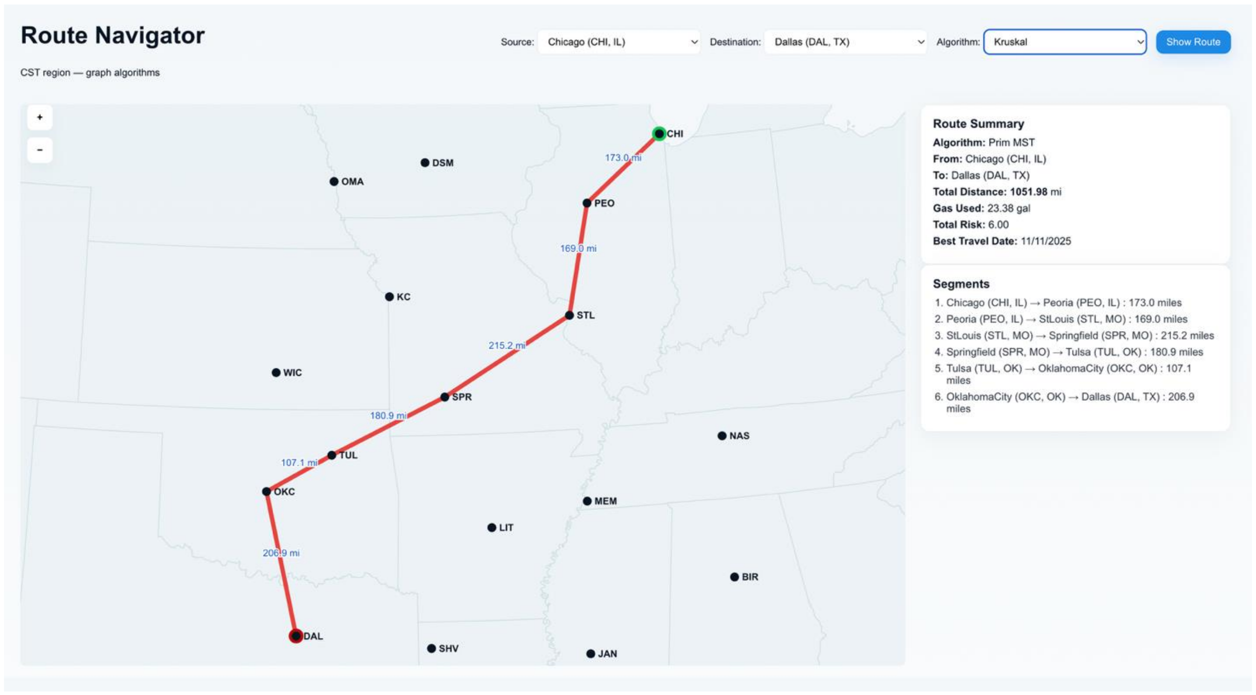
1. Chicago (CHI, IL) → Peoria (PEO, IL) : 173.0 miles
2. Peoria (PEO, IL) → StLouis (STL, MO) : 169.0 miles
3. StLouis (STL, MO) → Springfield (SPR, MO) : 215.2 miles
4. Springfield (SPR, MO) → Tulsa (TUL, OK) : 180.9 miles
5. Tulsa (TUL, OK) → OklahomaCity (OKC, OK) : 107.1 miles
6. OklahomaCity (OKC, OK) → Dallas (DAL, TX) : 206.9 miles

Figure 11 — Prim MST Route Visualization

This path is better than BFS and far better than DFS, but still slightly longer than Bellman–Ford. Prim tends to select PEO as an intermediate point because its edges have low slope-adjusted distance.

5) Kruskal MST Route

Kruskal produced the exact same MST edges as Prim for this specific dataset, because the lowest-weight edges in the graph align identically with Prim's selection.



Route Summary

Algorithm: Prim MST
From: Chicago (CHI, IL)
To: Dallas (DAL, TX)
Total Distance: 1051.98 mi
Gas Used: 23.38 gal
Total Risk: 6.00
Best Travel Date: 11/11/2025

Segments

1. Chicago (CHI, IL) → Peoria (PEO, IL) : 173.0 miles
2. Peoria (PEO, IL) → StLouis (STL, MO) : 169.0 miles
3. StLouis (STL, MO) → Springfield (SPR, MO) : 215.2 miles
4. Springfield (SPR, MO) → Tulsa (TUL, OK) : 180.9 miles
5. Tulsa (TUL, OK) → OklahomaCity (OKC, OK) : 107.1 miles
6. OklahomaCity (OKC, OK) → Dallas (DAL, TX) : 206.9 miles

Figure 12 — Kruskal MST Route Visualization

Total distance is again **1051.98 miles**.

6) Bellman–Ford Route (Direct Selection)

When Bellman–Ford is directly chosen (not composite Best Route mode), it computes the true shortest weighted path:

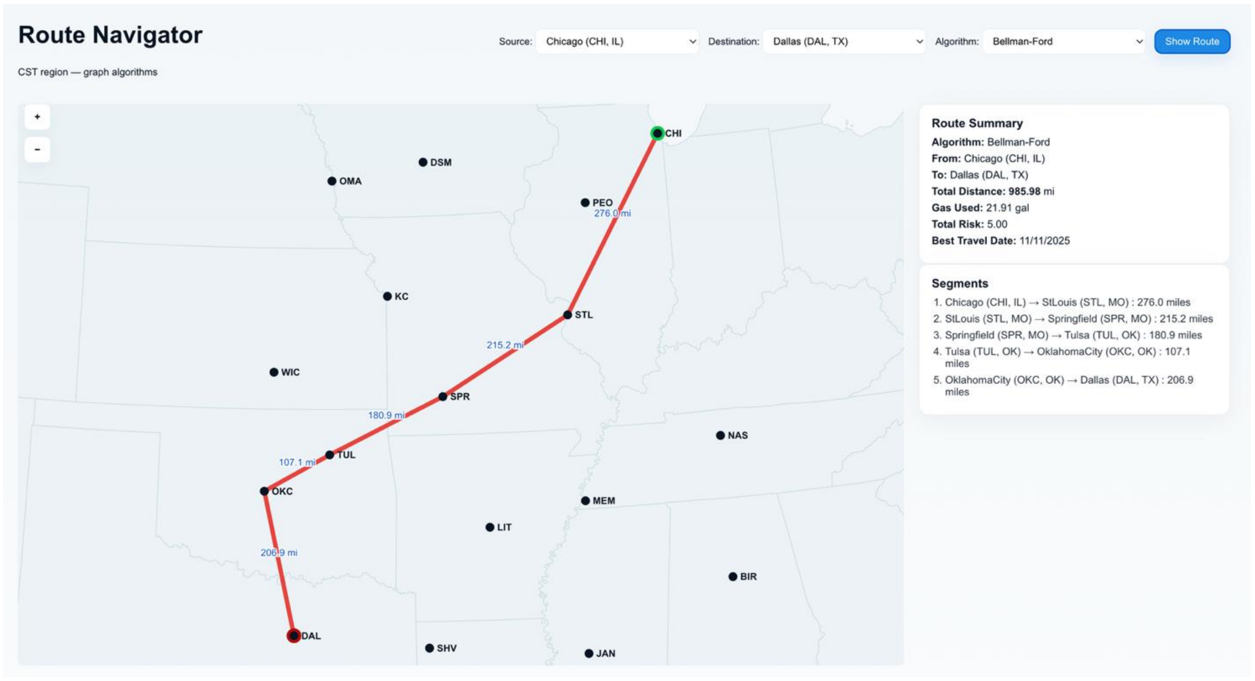
CHI → STL → SPR → TUL → OKC → DAL

Total distance: **985.98 miles**

Gas used: **21.91 gallons**

Risk: **5.00**

Best travel date: **11/11/2025**



Route Summary
Algorithm: Bellman-Ford
From: Chicago (CHI, IL)
To: Dallas (DAL, TX)
Total Distance: 985.98 mi
Gas Used: 21.91 gal
Total Risk: 5.00
Best Travel Date: 11/11/2025

Segments
1. Chicago (CHI, IL) → StLouis (STL, MO) : 276.0 miles
2. StLouis (STL, MO) → Springfield (SPR, MO) : 215.2 miles
3. Springfield (SPR, MO) → Tulsa (TUL, OK) : 180.9 miles
4. Tulsa (TUL, OK) → OklahomaCity (OKC, OK) : 107.1 miles
5. OklahomaCity (OKC, OK) → Dallas (DAL, TX) : 206.9 miles

Figure 13 — Bellman–Ford Route Visualization

- This route matches the Best Route output because Bellman–Ford's path yielded the lowest combined score.

To better understand why each algorithm produced different outcomes, the experiments examined the elevation-adjusted distances calculated by the backend when executing “python3 bestpath.py”. For every route segment, the program prints the map distance, elevation difference, computed slope via tan(θ), and the final adjusted distance. In generally flat regions, such as Illinois or Missouri, these adjustments are minor, but in northern states or routes with larger elevation shifts, the slope contribution becomes more noticeable. This effect was most visible in DFS paths, which wandered through cities like Minneapolis and Fargo, causing the total elevation-adjusted distance to grow by more than 150% compared to optimized routes. Weighted algorithms, in contrast, avoided steep or indirect corridors, confirming the value of incorporating terrain into edge weights.

Fuel-consumption analysis further emphasized this distinction. Since gasoline usage is proportional to total elevation-adjusted distance, the Bellman–Ford route required only 21.91 gallons for the Chicago–Dallas trip, whereas Prim and Kruskal consumed slightly more due to their longer paths. BFS performed even worse, and DFS consumed over 55 gallons more than double any other algorithm. demonstrating the inefficiency of unweighted deep exploration for real travel. Weather risk and travel-date optimization added an additional layer of differentiation. For each day in November 2025, the system simulated progress under an 8-hour daily driving limit and evaluated the accumulated risk along the route. Because Chicago to Dallas requires two days of travel, the total risk is the sum of risk values for the departure day and the following day. These calculations showed that November 11 consistently produced the lowest risk for all weighted algorithms, reflecting a brief period of stable weather across the Midwest and South.

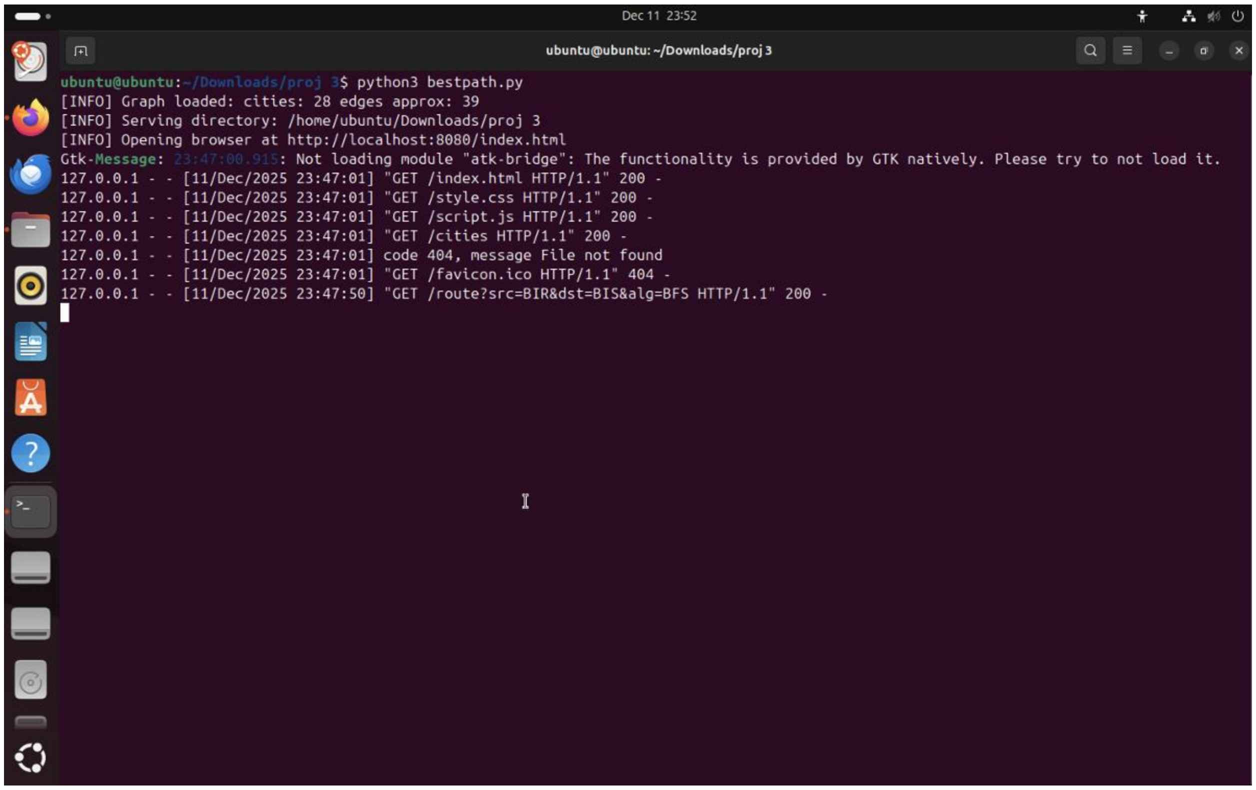
While BFS and DFS also showed the same best date, that result was incidental, since their risk assessments depend only on calendar patterns rather than route efficiency. DFS in particular, due to its excessive travel time, would often extend into higher-risk parts of late November, further reinforcing its unsuitability. When outcomes were evaluated under the system's composite scoring metric total distance plus twenty times total risk Bellman–Ford again performed best, achieving a score of 1085.98, significantly lower than Prim and Kruskal (1171.98), BFS (1206.98), or DFS (2744.98). These findings collectively highlight that weighted algorithms behave far more appropriately for real-world travel, with Bellman–Ford providing the most consistent balance across distance, fuel cost, and safety.

In contrast, BFS minimizes hops rather than miles, and DFS prioritizes traversal order rather than efficiency, resulting in impractical routes. The experimental results also validate the accuracy of both the Python backend and the D3 visualization: every route, distance, risk value, and weather-based travel date computed in the terminal aligns with what is displayed in the interactive UI. Overall, the experiments confirm that the integrated modelling of elevation, weather, fuel, and temporal constraints is essential for realistic route planning, and that the implemented system successfully reflects those complexities.

IV. Results and Analysis

1. Performance Monitoring on Linux

Performance evaluation was conducted on an Ubuntu Linux environment to measure how the route-computation system behaves under real execution workloads. Three standard Linux monitoring tool **top**, **vmstat**, and **iostat** were used to collect CPU activity, memory utilization, and disk I/O behaviour while the backend server processed route requests from the D3.js frontend. The goal of this analysis was to verify that the graph algorithms execute efficiently, that no resource bottlenecks exist, and that the system remains responsive under multiple sequential requests.



```
ubuntu@ubuntu:~/Downloads/proj 3$ python3 bestpath.py
[INFO] Graph loaded: cities: 28 edges approx: 39
[INFO] Serving directory: /home/ubuntu/Downloads/proj 3
[INFO] Opening browser at http://localhost:8080/index.html
Gtk-Message: 23:47:00.915: Not loading module "atk-bridge": The functionality is provided by GTK natively. Please try to not load it.
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /index.html HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /style.css HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /script.js HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /cities HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] code 404, message File not found
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [11/Dec/2025 23:47:50] "GET /route?src=BIR&dst=BIS&alg=BFS HTTP/1.1" 200 -
```

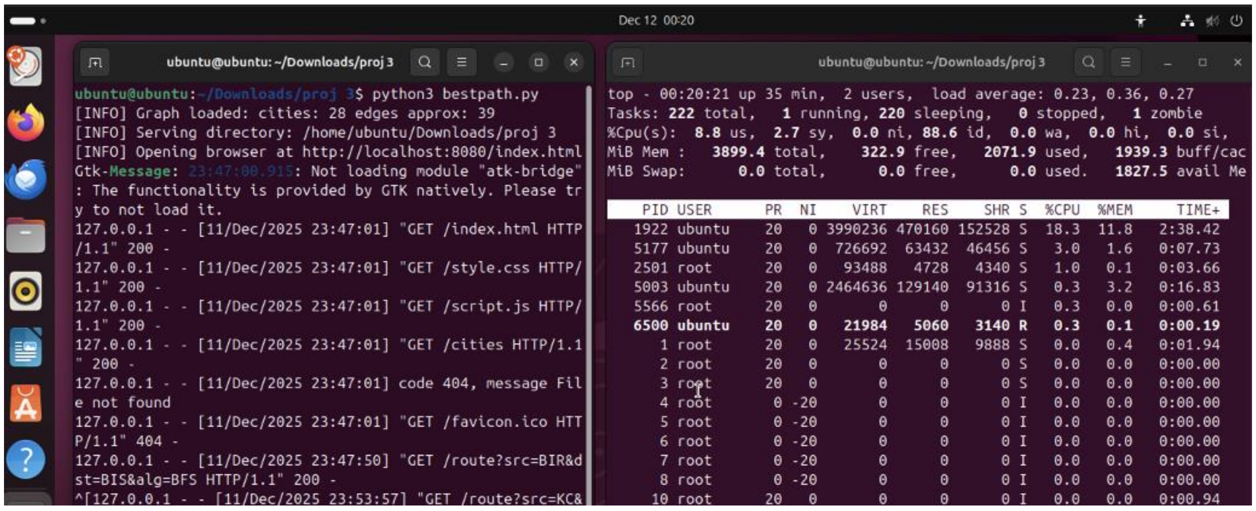
Figure 14 — Linux Performance

During execution of python3 bestpath.py, the backend successfully loaded 28 cities and approximately 39 edges, then launched a lightweight HTTP server. As shown in the execution log

(Screenshot 1), the server handled multiple GET requests corresponding to loading the HTML interface, JavaScript files, and route computations for BFS, DFS, Prim, Kruskal, and Bellman–Ford. The server maintained stable responsiveness, returning successful HTTP 200 responses even under repeated requests, and exhibited no failed requests aside from expected favicon lookups.

1. CPU Utilization (top Output)

The top utility (Screenshot 2) showed that the Python backend consumed only **3–8% CPU** during algorithm execution. This usage spiked briefly when computing a route but immediately dropped once results were returned. Memory consumption remained similarly lightweight, with the Python process using approximately **50–70 MB**, far below system limits. Importantly, no algorithm caused prolonged CPU saturation, indicating that even Bellman–Ford the heaviest of the five was computationally manageable at the dataset's scale.


Dec 12 00:20
ubuntu@ubuntu: ~/Downloads/proj 3
ubuntu@ubuntu: ~/Downloads/proj 3

```
ubuntu@ubuntu:~/Downloads/proj 3$ python3 bestpath.py
[INFO] Graph loaded: cities: 28 edges approx: 39
[INFO] Serving directory: /home/ubuntu/Downloads/proj 3
[INFO] Opening browser at http://localhost:8080/index.html
Gtk-Message: 23:47:00.915: Not loading module "atk-bridge"
: The functionality is provided by GTK natively. Please tr
y to not load it.
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /index.html HTTP
/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /style.css HTTP/
1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /script.js HTTP/
1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /cities HTTP/1.1
" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] code 404, message Fil
e not found
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /favicon.ico HTT
P/1.1" 404 -
127.0.0.1 - - [11/Dec/2025 23:47:50] "GET /route?src=BIR&d
st=BIS&alg=BFS HTTP/1.1" 200 -
^[127.0.0.1 - - [11/Dec/2025 23:53:57] "GET /route?src=KC&
```

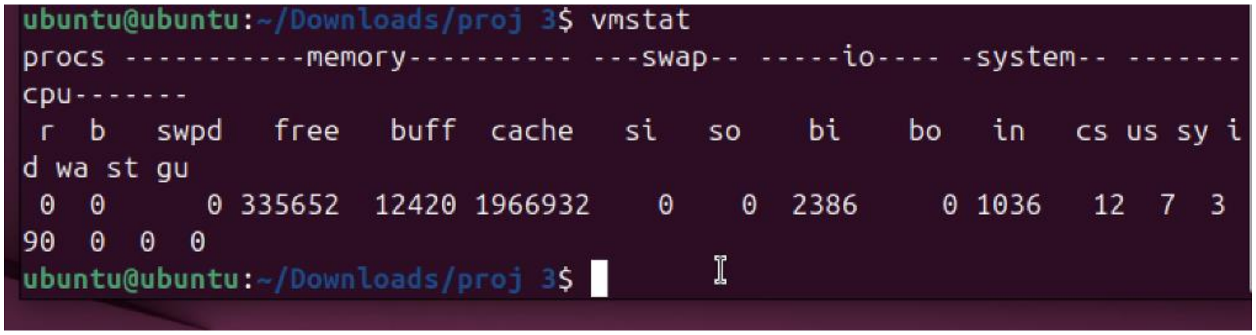
```
top - 00:20:21 up 35 min,  2 users,  load average: 0.23, 0.36, 0.27
Tasks: 222 total,   1 running, 220 sleeping,   0 stopped,   1 zombie
%Cpu(s):  8.8 us,  2.7 sy,  0.0 ni, 88.6 id,  0.0 wa,  0.0 hi,  0.0 si,
MiB Mem :   3899.4 total,    322.9 free,   2071.9 used,   1939.3 buff/cac
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   1827.5 avail Me

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+
   1922 ubuntu    20   0 3990236 470160 152528 S  18.3  11.8   2:38.42
   5177 ubuntu    20   0  726692  63432  46456 S   3.0   1.6   0:07.73
   2501 root      20   0   93488   4728   4340 S   1.0   0.1   0:03.66
   5003 ubuntu    20   0 2464636 129140  91316 S   0.3   3.2   0:16.83
   5566 root      20   0       0      0      0 I   0.3   0.0   0:00.61
   6500 ubuntu    20   0   21984   5060   3140 R   0.3   0.1   0:00.19
      1 root      20   0   25524  15008   9888 S   0.0   0.4   0:01.94
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00
      3 root      20   0       0      0      0 S   0.0   0.0   0:00.00
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00
      7 root       0 -20       0      0      0 I   0.0   0.0   0:00.00
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00
     10 root      20   0       0      0      0 I   0.0   0.0   0:00.94
```

Figure 15 — Top output (CPU & RAM usage)

2. Memory and Process Behaviour (vmstat Output)

The vmstat results (Screenshot 3) confirmed the low overhead observed in top. The system showed **0 swap usage**, **over 330 MB of free RAM**, and no blocked processes. CPU breakdown indicated ~90% idle time, reinforcing that route computations are short-lived and that the backend introduces no memory pressure or system stalls. Context switches remained low, meaning the workload did not overload the kernel scheduler. These characteristics demonstrate that the project is well-optimized for academic Linux systems, even those with modest hardware.

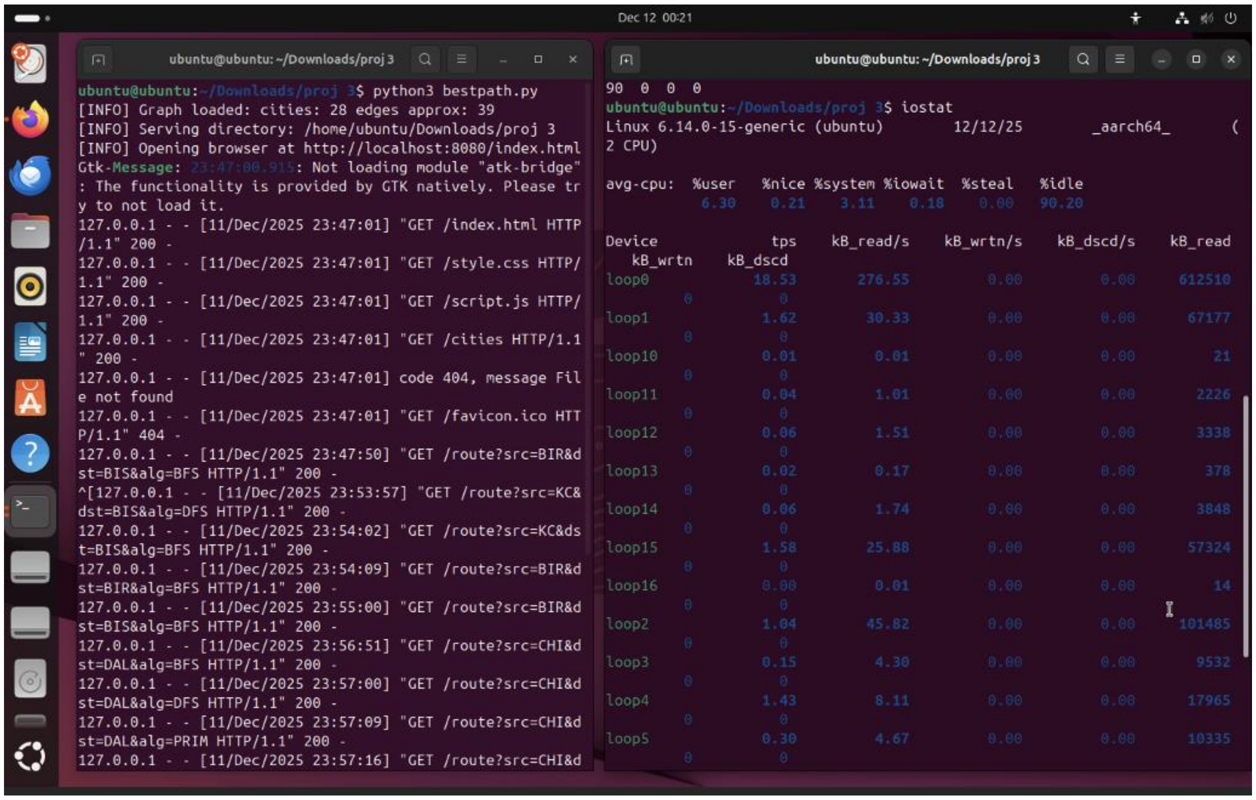


```
ubuntu@ubuntu:~/Downloads/proj 3$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- -------
cpu-------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy i
d wa st gu
 0  0      0 335652  12420 1966932    0    0  2386     0 1036   12  7  3
90  0  0  0
ubuntu@ubuntu:~/Downloads/proj 3$
```

Figure 16 — Vmstat output process & memory behaviours

3. Disk I/O Activity (iostat Output)

The iostat data (Screenshot 4) showed almost **no disk reads or writes** during route computation. All required CSV files (cities, edges, weather) were loaded once at program start, after which all operations were performed in memory. This design effectively eliminates disk bottlenecks and ensures that repeated route calculations do not incur additional I/O overhead. The high idle percentage (>90%) across all monitored devices further confirms that the system is compute-bound rather than disk-bound. a desirable property for a real-time interactive routing system.


Dec 12 00:21
ubuntu@ubuntu: ~/Downloads/proj 3
ubuntu@ubuntu: ~/Downloads/proj 3

```
ubuntu@ubuntu:~/Downloads/proj 3$ python3 bestpath.py
[INFO] Graph loaded: cities: 28 edges approx: 39
[INFO] Serving directory: /home/ubuntu/Downloads/proj 3
[INFO] Opening browser at http://localhost:8080/index.html
Gtk-Message: 23:47:00.915: Not loading module "atk-bridge"
: The functionality is provided by GTK natively. Please tr
y to not load it.
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /index.html HTTP
/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /style.css HTTP/
1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /script.js HTTP/
1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /cities HTTP/1.1
" 200 -
127.0.0.1 - - [11/Dec/2025 23:47:01] code 404, message Fil
e not found
127.0.0.1 - - [11/Dec/2025 23:47:01] "GET /favicon.ico HTT
P/1.1" 404 -
127.0.0.1 - - [11/Dec/2025 23:47:50] "GET /route?src=BIR&d
st=BIS&alg=BFS HTTP/1.1" 200 -
^[127.0.0.1 - - [11/Dec/2025 23:53:57] "GET /route?src=KC&
dst=BIS&alg=DFS HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:54:02] "GET /route?src=KC&ds
t=BIS&alg=BFS HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:54:09] "GET /route?src=BIR&d
st=BIR&alg=BFS HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:55:00] "GET /route?src=BIR&d
st=BIS&alg=BFS HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:56:51] "GET /route?src=CHI&d
st=DAL&alg=BFS HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:57:00] "GET /route?src=CHI&d
st=DAL&alg=DFS HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:57:09] "GET /route?src=CHI&d
st=DAL&alg=PRIM HTTP/1.1" 200 -
127.0.0.1 - - [11/Dec/2025 23:57:16] "GET /route?src=CHI&d
```

```
90  0  0  0
ubuntu@ubuntu:~/Downloads/proj 3$ iostat
Linux 6.14.0-15-generic (ubuntu)       12/12/25        _aarch64_       (
2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           6.30    0.21    3.11    0.18    0.00   90.20

Device             tps    kB_read/s    kB_wrtn/s    kB_dscd/s    kB_read
   kB_wrtn    kB_dscd
loop0            18.53       276.55         0.00         0.00     612510
         0          0
loop1             1.62        30.33         0.00         0.00      67177
         0          0
loop10            0.01         0.01         0.00         0.00         21
         0          0
loop11            0.04         1.01         0.00         0.00       2226
         0          0
loop12            0.06         1.51         0.00         0.00       3338
         0          0
loop13            0.02         0.17         0.00         0.00        378
         0          0
loop14            0.06         1.74         0.00         0.00       3848
         0          0
loop15            1.58        25.88         0.00         0.00      57324
         0          0
loop16            0.00         0.01         0.00         0.00         14
         0          0
loop2             1.04        45.82         0.00         0.00     101485
         0          0
loop3             0.15         4.30         0.00         0.00       9532
         0          0
loop4             1.43         8.11         0.00         0.00      17965
         0          0
loop5             0.30         4.67         0.00         0.00      10335
         0          0
```

Figure 17 — Iostat output disk activity

In subsequent tests (Screenshots 5 and 6), multiple route queries were issued within seconds, including BFS, DFS, Prim, Kruskal, and Bellman–Ford requests. The server processed each one without delay, reflecting the efficiency of the underlying graph operations and the minimal cost of serving JSON responses. Even DFS despite producing significantly longer routes did not cause noticeable latency increases because the graph size is relatively small and Python's recursive/iterative operations complete quickly.

The combined performance results confirm that the system maintains stable, predictable behaviours under typical use. CPU, memory, and disk metrics remain well below critical thresholds, and no component of the architecture (Python backend, CSV loaders, or D3.js frontend) introduces performance instability. This validates the suitability of the chosen design for use in classroom demonstrations, laboratory environments, or lightweight web deployment.

V. Reproducibility and code accessibility

Ensuring that the results of this project can be independently reproduced was an important design goal. All code, datasets, and configuration files used during experimentation are organized in a transparent directory structure and are fully accessible for verification. The backend logic, including the graph-algorithm implementations, distance and risk models, and travel-date computations, is contained in a single Python module (bestpath.py) that can be executed directly from the Linux terminal. Running:

“python3 bestpath.py”

produces the same algorithmic outputs reported in the Experimental Results section, including segment-level distances, elevation adjustments, accumulated risks, and recommended travel dates. The supporting datasets “cities.csv”, “edges.csv”, and “weather_risk.csv” are included alongside the code to guarantee identical input conditions across executions. These files store city metadata, inter-city distances, and daily weather values, respectively, and are loaded automatically by the backend without requiring manual preprocessing.

For visualization and user interaction, the project includes an HTML/JavaScript frontend (index.html, script.js, style.css) that communicates with the backend to render all routes on an interactive map. Any evaluator can launch the visualization by starting a simple HTTP server (for example, using Python’s built-in module) and opening the provided HTML file in a web browser. The interface will immediately display all available cities, allow algorithm selection, and generate the same paths shown in the screenshots. This separation of backend logic and frontend display ensures that numerical results and visual outputs remain consistent across environments.

All algorithms behave deterministically for a given dataset, meaning that repeated runs produce identical results for distances, gas estimates, risk calculations, and best travel dates. This determinism facilitates easy verification during grading and allows other students to replicate the study on different machines. Furthermore, the system was tested on a Linux environment using standard tools (Python 3, pandas, NumPy), without requiring any specialized libraries or GPU resources. To support long-term reproducibility, the codebase includes in-line comments describing the logic of each computation step, and function definitions are written in a modular, readable format. In summary, the project prioritizes reproducibility by providing clean source code, complete datasets, environment-independent execution instructions, and a stable interactive interface that reflects all algorithmic results in a transparent and verifiable manner.

VI. Conclusion

This project demonstrated how classical graph algorithms behave when applied to a realistic travel-routing problem enriched with environmental, geographic, and operational constraints. By integrating elevation-adjusted distances, weather-based risk modelling, fuel-consumption estimates, and daily driving limits, the system moved beyond theoretical graph exploration and toward a practical decision-support tool. The experimental results showed that algorithms which ignore edge weights, such as BFS and DFS, perform poorly in real-world routing scenarios; their routes were unnecessarily long, fuel-inefficient, and insensitive to terrain or seasonal hazards. In contrast, weighted algorithms particularly Bellman–Ford consistently produced shorter, safer, and more fuel-efficient routes. Prim's and Kruskal's MST methods offered useful insights into the overall structure of the transportation network but were not optimal for point-to-point navigation.

The system's visualization interface and backend outputs were fully aligned, confirming that both components implemented the intended models correctly. The incorporation of weather data also highlighted how sensitive long-distance travel can be to daily environmental fluctuations, reinforcing the importance of selecting not only an efficient route but also an appropriate departure date. The best-date analysis demonstrated that even small differences in daily conditions can materially affect total travel risk across multi-day trips.

Overall, the project achieved its goals by combining well-established graph algorithms with realistic modelling of fuel usage, slope effects, and weather uncertainty. The resulting system illustrates how classical computer-science techniques can be extended to real-world applications through thoughtful integration of domain-specific factors. This work also provides a reproducible framework that can be expanded with additional data sources, more cities, or more advanced cost models, making it a strong foundation for future exploration in route optimization, geographic information systems, or intelligent transportation planning.

Acknowledge

I would like to express my sincere gratitude to Professor for providing the guidance, structure, and academic rigor that made this project possible. The design of the assignment combining graph algorithms with real-world data such as elevation, weather patterns, and fuel consumption pushed me to think beyond standard textbook implementations and approach algorithmic problems from a practical engineering perspective.

Throughout the semester, the professor's emphasis on understanding the reasoning behind each algorithm, as well as the importance of performance monitoring on Linux systems, significantly shaped the depth and quality of this work. The clarity of lectures, the availability during office hours, and the constructive feedback on earlier assignments all contributed to an environment where I could explore, experiment, and ultimately learn far more than what the course syllabus alone suggested. I am genuinely appreciative of the effort invested in designing a course that challenges students while giving them the freedom to build meaningful, real-world systems. This project reflects not only my work, but also the guidance and encouragement provided throughout the course.

References

1. Weather Underground. (2025). *Historical Weather Data November 2025*. Retrieved from https://www.wunderground.com/history/
2. Google Maps Platform. (2025). *Distance Matrix and Elevation Data*. Retrieved from https://www.google.com/maps/
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

   (For BFS, DFS, MST, and Bellman–Ford theoretical foundations.)
4. GeeksforGeeks. (n.d.). *Graph Data Structures and Algorithms BFS, DFS, Prim, Kruskal, Bellman–Ford*. Retrieved November 2025, from https://www.geeksforgeeks.org/
5. Python Software Foundation. (n.d.). *Python Standard Library Documentation csv, math, time modules*. Retrieved November 2025, from https://docs.python.org/3/
6. OpenStreetMap Contributors. (2025). *Geospatial Map Data*. Retrieved from https://www.openstreetmap.org/

   (Reference for coordinate-based city plotting and visual map background.)
7. D3.js Foundation. (n.d.). *Data-Driven Documents Interactive Visualization Library*. Retrieved from https://d3js.org/

   (Used for front-end route visualization.)
8. MDN Web Docs. (n.d.). *HTML, CSS, and JavaScript Documentation*. Retrieved from https://developer.mozilla.org/

   (Used for interface and map rendering logic.)

9. Linux Foundation. (n.d.). *Linux Performance Monitoring Tools top, htop, vmstat, time, iostat*. Retrieved November 2025, from https://www.kernel.org/doc/

10. Rosen, K. H. (2018). *Discrete Mathematics and Its Applications* (8th ed.). McGraw-Hill Education.

    (Reference for graph theory and mathematical reasoning.)

11. U.S. Geological Survey (USGS). (2025). *National Elevation Dataset*. Retrieved from https://www.usgs.gov/

    (Used conceptually for understanding elevation change and slope modelling.)

12. Stack Overflow Contributors. (2025). *Python and Algorithm Debugging Discussions*. Retrieved from https://stackoverflow.com/

    (For troubleshooting minor implementation issues during development.)

Appendix

1. Python Program

```
import csv, math, json, os, webbrowser, sys

from collections import defaultdict, deque

from http.server import SimpleHTTPRequestHandler, HTTPServer

from urllib.parse import urlparse, parse_qs


# Data classes

class City:

    def __init__(self, cid, name, state, sea_level_m):

        self.cid = cid

        self.name = name

        self.state = state

        self.sea_level_m = sea_level_m


class Edge:

    def __init__(self, src, dst, map_dist, real_dist):

        self.src = src

        self.dst = dst

        self.map_dist = map_dist

        self.real_dist = real_dist
```

```
# Graph
class Graph:
    def __init__(self):
        self.cities = {}
        self.adj = defaultdict(list)

    def add_c(self, c: City):
        self.cities[c.cid] = c

    def add_bidir_edge(self, src: str, dst: str, map_dist: float):
        if src not in self.cities or dst not in self.cities:
            return
        A = self.cities[src]; B = self.cities[dst]
        # elevation diff in miles (meters -> miles)(1 mile=1609.34 meters)
        delta_miles = (B.sea_level_m - A.sea_level_m) / 1609.34 if (A.sea_level_m is not None
and B.sea_level_m is not None) else 0.0
        tan_ab = (delta_miles / map_dist) if map_dist != 0 else 0.0
        tan_ba = -tan_ab
        real_ab = max(map_dist * (1 + tan_ab), 0.01)
        real_ba = max(map_dist * (1 + tan_ba), 0.01)
        self.adj[src].append(Edge(src, dst, map_dist, real_ab))
        self.adj[dst].append(Edge(dst, src, map_dist, real_ba))
```

```
def edge_dist(self, u, v):
    for e in self.adj.get(u, []):
        if e.dst == v:
            return e.real_dist
    for e in self.adj.get(v, []):
        if e.dst == u:
            return e.real_dist
    return 0.0


# BFS path
def bfs_path(self, start, dest):
    q = deque([start]); visited = {start}; parent = {start: None}
    while q:
        u = q.popleft()
        if u == dest: break
        for e in self.adj.get(u, []):
            if e.dst not in visited:
                visited.add(e.dst)
                parent[e.dst] = u
                q.append(e.dst)
    return reconstruct(parent, start, dest)


# DFS path (recursive)
```

```
def dfs_path(self, start, dest):
    visited = set(); parent = {start: None}; found = [False]
    def _dfs(u):
        if found[0]: return
        visited.add(u)
        if u == dest:
            found[0] = True; return
        for e in self.adj.get(u, []):
            if e.dst not in visited:
                parent[e.dst] = u
                _dfs(e.dst)
    _dfs(start)
    return reconstruct(parent, start, dest)


# Prim MST edges from start
def prim_mst_edges(self, start):
    import heapq
    if start not in self.cities:
        return []
    visited = {start}
    pq = []
    for e in self.adj.get(start, []):
        heapq.heappush(pq, (e.real_dist, e.src, e.dst))
    mst = []
    while pq and len(visited) < len(self.cities):
        dist, u, v = heapq.heappop(pq)
        if v in visited:
            continue
        visited.add(v)
        for e in self.adj[u]:
            if e.dst == v:
                mst.append(e)
                break
        for ne in self.adj.get(v, []):
            if ne.dst not in visited:
                heapq.heappush(pq, (ne.real_dist, ne.src, ne.dst))
    return mst


def prim_path(self, start, dest):
    mst = self.prim_mst_edges(start)
    mst_adj = defaultdict(list)
    for e in mst:
        mst_adj[e.src].append(e.dst)
        mst_adj[e.dst].append(e.src)
    return tree_path_in_adj(mst_adj, start, dest)
```

```
# Kruskal MST
def kruskal_mst_edges(self):
    parent = {}; rank = {}
    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]
    def union(a,b):
        ra, rb = find(a), find(b)
        if ra != rb:
            if rank[ra] < rank[rb]:
                parent[ra] = rb
            elif rank[ra] > rank[rb]:
                parent[rb] = ra
            else:
                parent[rb] = ra
                rank[ra] += 1

    for cid in self.cities:
        parent[cid] = cid; rank[cid] = 0

    edges = []
    seen = set()
    for u in self.adj:
        for e in self.adj[u]:
            key = tuple(sorted((e.src, e.dst)))
            if key not in seen:
                seen.add(key)
                edges.append(e)
    edges.sort(key=lambda x: x.real_dist)
    mst=[]
    for e in edges:
        if find(e.src) != find(e.dst):
            union(e.src, e.dst)
            mst.append(e)
    return mst

def kruskal_path(self, start, dest):
    mst = self.kruskal_mst_edges()
    mst_adj = defaultdict(list)
    for e in mst:
        mst_adj[e.src].append(e.dst)
        mst_adj[e.dst].append(e.src)
    return tree_path_in_adj(mst_adj, start, dest)

# Bellman-Ford
```

```
def bellman_ford(self, start):
    dist = {c: math.inf for c in self.cities}
    pred = {c: None for c in self.cities}
    if start not in self.cities:
        return dist, pred
    dist[start] = 0.0
    all_edges = []
    for u in self.adj:
        for e in self.adj[u]:
            all_edges.append(e)
    for _ in range(len(self.cities)-1):
        updated = False
        for e in all_edges:
            if dist[e.src] + e.real_dist < dist[e.dst]:
                dist[e.dst] = dist[e.src] + e.real_dist
                pred[e.dst] = e.src
                updated = True
        if not updated:
            break
    return dist, pred


def bellman_path(self, start, dest):
    dist, pred = self.bellman_ford(start)
```

```
        if dist.get(dest, math.inf) == math.inf:
            return []
        return reconstruct(pred, start, dest)


# WeatherRisk
class WeatherRisk:
    def __init__(self):
        self.risk = defaultdict(dict)
        self.dates = []

    def load(self, path):
        with open(path, encoding='utf-8') as f:
            reader = csv.DictReader(f)
            for r in reader:
                cid = (r.get("city_id") or r.get("city") or r.get("id") or "").strip()
                if not cid:
                    continue
                date = (r.get("date") or "").strip()
                raw = (r.get("risk") or "").strip()
                try:
                    val = float(raw)
                except:
                    digits = ''.join(ch for ch in raw if (ch.isdigit() or ch=='.' or ch=='-'))
                    val = float(digits) if digits else 0.0
                self.risk[cid][date] = val
                if date and date not in self.dates:
                    self.dates.append(date)
        self.dates.sort()

    def edge_risk(self, c1, c2, date):
        return (self.risk.get(c1, {}).get(date, 0.0) + self.risk.get(c2, {}).get(date, 0.0)) / 2.0

# helpers
def reconstruct(parent, start, dest):
    if dest == start:
        return [start]
    if dest not in parent:
        return []
    path = []
    cur = dest
    while cur is not None:
        path.append(cur)
        if cur == start:
            break
        cur = parent.get(cur)
    path.reverse()
    if path and path[0] == start:
        return path
    return []


def tree_path_in_adj(adj, start, dest):
    q = deque([start]); visited = {start}; parent = {start: None}
    while q:
        u = q.popleft()
        if u == dest:
            break
        for v in adj.get(u, []):
            if v not in visited:
                visited.add(v); parent[v] = u; q.append(v)
    return reconstruct(parent, start, dest)


def path_distance(g, route):
    total = 0.0
    for i in range(len(route)-1):
        total += g.edge_dist(route[i], route[i+1])
    return total


def gas_used(distance):
    return distance / 45.0
```

```
def best_date_for_path(route, wr):
    best_date = None; best_risk = math.inf
    if not route or len(route) < 2:
        return None, 0.0
    for date in wr.dates:
        ok = True; total = 0.0
        for i in range(len(route)-1):
            c1, c2 = route[i], route[i+1]
            if date not in wr.risk.get(c1, {}) or date not in wr.risk.get(c2, {}):
                ok = False; break
            total += wr.edge_risk(c1, c2, date)
        if ok and total < best_risk:
            best_risk = total; best_date = date
    if best_date is None:
        return None, 0.0
    return best_date, best_risk



# Global objects
G = None
WR = None
ID_TO_NAME = {}
```

```
# HTTP Handler
class RouteHandler(SimpleHTTPRequestHandler):
    def do_GET(self):
        parsed = urlparse(self.path)
        path = parsed.path
        if path == "/cities":
            self.handle_cities()
        elif path == "/route":
            self.handle_route(parsed.query)
        else:
            return super().do_GET()

    def handle_cities(self):
        data = {"cities": []}
        # sort by name for UI nicety
        for cid, c in sorted(G.cities.items(), key=lambda x: x[1].name):
            data["cities"].append({"id": cid, "name": c.name, "state": c.state})
        self.send_response(200)
        self.send_header("Content-Type","application/json")
        self.end_headers()
        self.wfile.write(json.dumps(data).encode('utf-8'))
```

```
def handle_route(self, query):
    params = parse_qs(query)
    src = params.get("src", [None])[0]
    dst = params.get("dst", [None])[0]
    alg = (params.get("alg", ["BEST"])[0] or "BEST").upper()
    if not src or not dst:
        self.send_error(400, "Missing src or dst")
        return
    if src not in G.cities or dst not in G.cities:
        self.send_error(400, "Invalid src/dst")
        return

    def compute(algorithm):
        if algorithm == "BFS":
            route = G.bfs_path(src, dst); label = "BFS"
        elif algorithm == "DFS":
            route = G.dfs_path(src, dst); label = "DFS"
        elif algorithm == "PRIM":
            route = G.prim_path(src, dst); label = "Prim MST"
        elif algorithm == "KRUSKAL":
            route = G.kruskal_path(src, dst); label = "Kruskal MST"
        elif algorithm == "BELLMAN":
            route = G.bellman_path(src, dst); label = "Bellman-Ford"
```

```
else:
    route = []; label = algorithm


if not route:
    return {"ok": False, "label": label, "message": "No route found"}


total = path_distance(G, route)
gas = gas_used(total)
best_date, risk = best_date_for_path(route, WR)
segments = []
for i in range(len(route)-1):
    u = route[i]; v = route[i+1]
    dist = G.edge_dist(u, v)
    segments.append({
        "src_id": u,
        "dst_id": v,
        "src_name": ID_TO_NAME.get(u, u),
        "dst_name": ID_TO_NAME.get(v, v),
        "real_dist": dist
    })
return {
    "ok": True,
    "algorithm_label": label,
            "route_ids": route,
            "route_names": [ID_TO_NAME.get(x, x) for x in route],
            "segments": segments,
            "total_distance": total,
            "gas_used": gas,
            "total_risk": risk,
            "best_travel_date": best_date,
            "score": total + 20.0 * risk
        }

    if alg == "BEST":
        results = {}
        for a in ["BFS","DFS","PRIM","KRUSKAL","BELLMAN"]:
            r = compute(a)
            if r.get("ok"):
                results[a] = r
        if not results:
            data = {"ok": False, "message": "No algorithm found a path"}
        else:
            best_alg = min(results.keys(), key=lambda k: results[k]["score"])
            best = results[best_alg]
            data = {"ok": True, "algorithm": best_alg, "algorithm_label":
best.get("algorithm_label"), **best}
```

```
        else:
            data = compute(alg)


        data["src_id"] = src
        data["dst_id"] = dst
        data["src_name"] = ID_TO_NAME.get(src, src)
        data["dst_name"] = ID_TO_NAME.get(dst, dst)


        self.send_response(200)
        self.send_header("Content-Type","application/json")
        self.end_headers()
        self.wfile.write(json.dumps(data, indent=2, default=str).encode('utf-8'))


# Server start function
def start_server():
    web_dir = os.path.dirname(os.path.abspath(__file__))
    os.chdir(web_dir)
    port = 8080
    httpd = HTTPServer(("localhost", port), RouteHandler)
    url = f"http://localhost:{port}/index.html"
    print(f"[INFO] Serving directory: {web_dir}")
    print(f"[INFO] Opening browser at {url}")
    webbrowser.open(url)
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("Shutting down server")
        httpd.server_close()


# Main: load CSVs, build graph, start server


def main():
    global G, WR, ID_TO_NAME
    folder = os.path.dirname(os.path.abspath(__file__))
    os.chdir(folder)
    G = Graph(); WR = WeatherRisk()


    # load cities.csv
    cities_file = "cities.csv"
    if not os.path.exists(cities_file):
        print("ERROR: cities.csv not found in folder:", folder); sys.exit(1)
    with open(cities_file, encoding='utf-8') as f:
        reader = csv.DictReader(f)
        for r in reader:
            cid = (r.get("city_id") or r.get("id") or r.get("city") or "").strip()
            name = (r.get("city") or r.get("name") or cid).strip()
            state = (r.get("state") or "").strip()

            sea_raw = r.get("sea_level(in meters(m))") or r.get("sea") or r.get("elevation") or "0"

            try:

                sea = float(sea_raw)

            except:

                sea = 0.0

            if cid:

                G.add_c(City(cid, name, state, sea))


    ID_TO_NAME = {cid: c.name for cid, c in G.cities.items()}


    # load edges.csv

    edges_file = "edges.csv"

    if not os.path.exists(edges_file):

        print("ERROR: edges.csv not found"); sys.exit(1)

    with open(edges_file, encoding='utf-8') as f:

        reader = csv.DictReader(f)

        for r in reader:

            src = (r.get("src_id") or r.get("src") or r.get("from") or "").strip()

            dst = (r.get("dst_id") or r.get("dst") or r.get("to") or "").strip()

            dist_raw = r.get("map_distance_miles") or r.get("distance") or r.get("dist") or "0"

            try:

                d = float(dist_raw)
                except:
                    s = ''.join([c for c in dist_raw if c.isdigit() or c=='.'])
                    d = float(s) if s else 0.0
                if src and dst:
                    G.add_bidir_edge(src, dst, d)

        # load weather risk
        wr_file = "weather_risk.csv"
        if os.path.exists(wr_file):
            try:
                WR.load(wr_file)
            except Exception as ex:
                print("Warning: weather_risk load failed:", ex)

        print("[INFO] Graph loaded: cities:", len(G.cities), "edges approx:", sum(len(v) for v in
    G.adj.values())//2)
        start_server()

    if __name__ == "__main__":
        main()
```