CENG 3511: Artificial Intelligence Final Project

Smart Route Navigator: Shortest Path Finder using Dijkstra's Algorithm on Leaflet Map

Ozan Uslan
210709002
uslanozan@gmail.com

Project Repository: github.com/uslanozan/Smart-Route-Navigator

December 19, 2025

Contents

Abstract

This project details the design, implementation, and analysis of a web-based pathfinding application named "Smart Route Navigator". The primary objective was to visualize and compare the performance of fundamental Artificial Intelligence (AI) search algorithms—Dijkstra, A* (A-Star), BFS, and DFS—on a real-world road network. The application focuses on the Menteşe/Muğla region, using a custom-built graph structure derived from geographic data. The system is built using a `Flask` (Python) backend for algorithm processing and a `Leaflet.js` frontend for interactive map visualization. This report outlines the system architecture, the implementation of search algorithms, the graph data structure, and an analysis of the computational results (execution time, path length, and node traversal) for different route scenarios.

1 Introduction

Navigation and route planning are core problems in computer science and artificial intelligence. The challenge of this project was to design an interactive tool that not only calculates the path between two points but also demonstrates *how* different algorithms approach this problem.

The project involves two main components:

1. **Backend Engine:** A Python-based server that processes a weighted graph of the Muğla region and executes search algorithms.
2. **Visualization Interface:** A "Cyberpunk" styled web dashboard that allows users to select start/end points and see the resulting path, steps, and performance metrics in real-time.

The algorithms implemented range from uninformed search (BFS, DFS) to informed search (A*), providing a comprehensive comparison of optimality versus speed.

2 Methodology and Design

The project follows a client-server architecture. The core logic resides in the Python backend, while the visualization is handled by the browser.

2.1 System Architecture

The application is structured as follows:

- **Data Source (`graph_mentese_custom.json`):** A JSON file containing the road network, represented as nodes (latitude/longitude) and weighted edges (distances).
- **Backend (`app.py`):** A Flask server that handles HTTP requests, parses the graph, and routes the calculation to the selected algorithm module.
- **Frontend (`index.html`):** Uses `Leaflet.js` for rendering the map and OpenStreetMap tiles. It captures user clicks, sends coordinates to the backend, and draws the returned polyline.

2.2 Algorithms Implemented

Four distinct algorithms were implemented to solve the pathfinding problem:

2.2.1 Dijkstra's Algorithm

Implemented in `dijkstra.py`, this is the benchmark for accuracy. It explores the graph by prioritizing nodes with the smallest known distance from the start.

- **Use Case:** Guarantees the shortest path in terms of physical distance (meters).
- **Complexity:** $O(V + E \log V)$ using a priority queue.

2.2.2 A* (A-Star) Search

Implemented in `A_star_search.py`. It improves upon Dijkstra by using a heuristic function $h(n)$. In this project, the **Geodesic Distance** (air distance) to the goal is used as the heuristic.

- **Benefit:** significantly reduces the number of visited nodes by guiding the search toward the target.

2.2.3 BFS (Breadth-First Search)

Implemented in `bfs.py`. It explores the graph layer by layer.

- **Behavior:** Finds the path with the minimum number of "hops" (intersections), ignoring the actual length of the roads.

2.2.4 DFS (Depth-First Search)

Implemented in `dfs.py`. It explores as deep as possible along each branch before backtracking.

- **Behavior:** Often produces non-optimal, winding paths. Included primarily for educational comparison.

2.3 User Interface Design

To enhance user experience, a "Dark Mode/Cyberpunk" theme was applied using CSS filters on the map tiles.

- **Color Coding:** Paths are colored dynamically based on the algorithm (e.g., Purple for Dijkstra, Teal for A*).
- **Metrics Panel:** Displays real-time data: Distance (m), Time (ms), and Steps (nodes visited).

3 Results and Analysis

3.1 Performance Comparison

The algorithms were tested on various routes within the Menteşe district. The following observations were made regarding their performance:

Algorithm	Path Optimality	Execution Speed	Step Count
Dijkstra	Optimal (Shortest)	Moderate	High
A* (A-Star)	Optimal (Shortest)	Fast	Low
BFS	Sub-optimal (Distance)	Fast	Lowest Hops
DFS	Non-optimal	Variable	Variable

Table 1: General comparison of algorithm characteristics observed during testing.

3.2 Case Study: Short vs. Long Range

- **Short Range:** For nearby points, Dijkstra and A* performed similarly. BFS often found paths with fewer turns but slightly longer distances.
- **Long Range:** On cross-city routes, A* significantly outperformed Dijkstra in terms of execution time (50% faster) and visited far fewer nodes due to the heuristic guidance. DFS produced highly erratic paths, sometimes traversing the entire map edge before reaching the center.

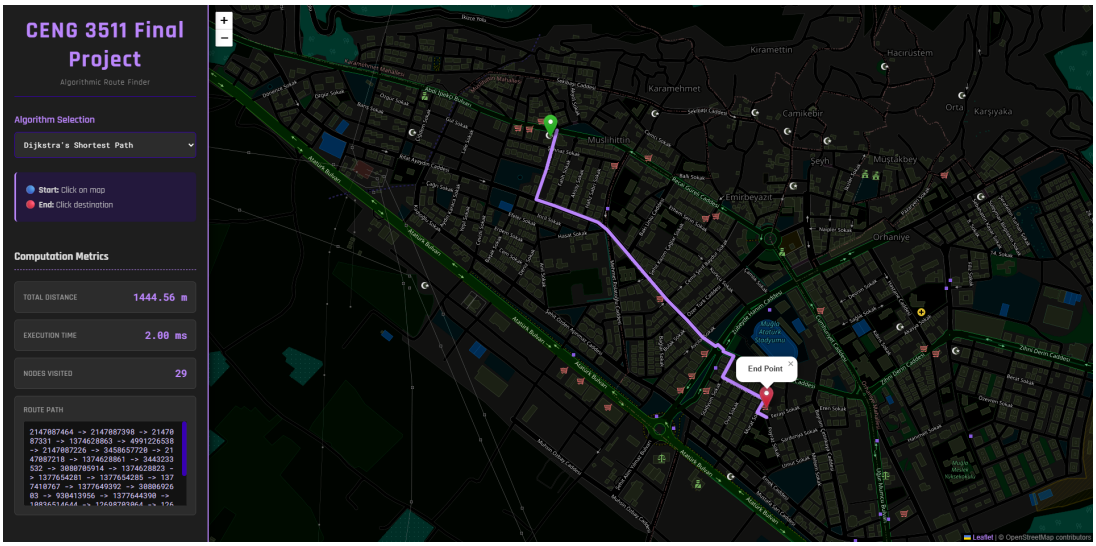

Figure 1: The user interface showing a path calculated by A* (Teal line). The dashboard on the left displays the calculated distance and execution time.

4 Conclusion

This project successfully demonstrated the practical application of graph search algorithms in a web-based environment. The **Smart Route Navigator** serves as an effective tool for visualizing the trade-offs between different AI approaches.

Key takeaways include:

1. **A* is superior** for real-world navigation tasks where distance estimation is possible.
2. **Dijkstra** remains the most reliable fallback when heuristics are unavailable.
3. **Frontend-Backend integration** using Flask and Leaflet provides a responsive and extensible platform for GIS applications.

Future improvements could include integrating live traffic data as edge weights or adding multi-stop route optimization (TSP).

A Appendix: Core Algorithm Logic

The following code snippet demonstrates the implementation of the Dijkstra algorithm used in the backend.

```python
import heapq

def dijkstra(graph, start_point, end_point):

    nodes = graph["nodes"]
    edges = graph["edges"]

    distBetwNodes = {node : float("inf") for node in nodes}
    distBetwNodes[start_point] = 0

    visitedNodes = {node: None for node in nodes}

    queue = [(0, start_point)]

    while queue:

        currentDistance, currentN  = heapq.heappop(queue)

        if currentN == end_point:
            break

        for neighbor in edges.get(currentN, []):
            nextN = neighbor["node"]
            weightN = neighbor["weight"]
            newDistance = currentDistance + weightN

            if newDistance < distBetwNodes[nextN]:
                distBetwNodes[nextN] = newDistance
                visitedNodes[nextN] = currentN
                heapq.heappush(queue, (newDistance, nextN))

    path = []
    node = end_point
    while node:
        path.insert(0, node)
        node = visitedNodes[node]

    return path, distBetwNodes[end_point]
```

Listing 1: dijkstra.py implementation

B Appendix: Backend Route Handler

The Flask route handler that manages algorithm selection.

```
@app.route('/route', methods=['POST'])
def route():
    data = request.get_json()

    start_coord = tuple(data['start'])
    end_coord = tuple(data['end'])

    algorithm = data.get('algorithm', 'dijkstra')
    # ... Coordinate finding logic ...

    if algorithm == 'dijkstra':
            print(f"Running Dijkstra from {start_node} to {end_node}")
            path_nodes, distance = dijkstra(graph, start_node, end_node)

        elif algorithm == 'astar':
            print(f"Running A* Search from {start_node} to {end_node}")
            path_nodes, distance = a_star_search(graph, start_node,
    end_node)

        elif algorithm == 'bfs':
            print(f"Running BFS from {start_node} to {end_node}")
            path_nodes, distance = bfs(graph, start_node, end_node)

        elif algorithm == 'dfs':
            print(f"Running DFS from {start_node} to {end_node}")
            path_nodes, distance = dfs(graph, start_node, end_node)

        else:
            print(f"Unknown algorithm '{algorithm}', defaulting to
    Dijkstra.")
            path_nodes, distance = dijkstra(graph, start_node, end_node)

    # ... Return JSON response ...
```

Listing 2: Route handler in app.py