**harita, ekran görüntüsü, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.**

**CENG – 3511**

**ARTIFICIAL INTELLIGENCE MIDTERM PROJECT**

**Smart Path Finder with Leaflet and Dijkstra’s Algorithm**

SÜLEYMAN EMRE PARLAK – 210709005

**Instructor :** Prof. Dr. Bekir Taner Dinçer

**Teaching Assistant :** Selahattin Aksoy

**INTRODUCTION**

Smart Path Finder is a map-based path finding application that finds the shortest path between two points using the Dijkstra algorithm. By selecting the start and end points on the map, the user sees the shortest path found by the algorithm and accesses its details. In addition, the A\* Search algorithm has been added to the project to make performance comparisons.

**SETUP INSTRUCTION**

* Clone the repository. ( git clone https://github.com/paitblack/CENG\_3511\_Artificial\_Intelligence\_Midterm\_Project.git)
* Open any IDE you preferred (Visual Studio etc.)
* Install required packages (if any).
* Command "python app.py" on the terminal (run the app).
* Click the link provided in the terminal to launch the app in your browser.

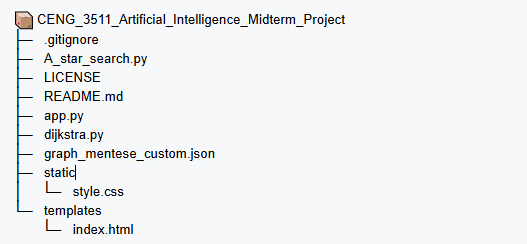
**TOOLS USED**

* Python — Backend logic, algorithms' implementations and route calculations
* Flask — Lightweight web framework for serving the application
* HTML & CSS — Frontend structure and styling
* JavaScript (inline in HTML) — Frontend interaction logic
* Leaflet.js — Interactive map visualization library

**GOAL**

* Determining the shortest path between two selected points on the Leaflet.js map using the Dijkstra algorithm.
* Displaying this determined path on the map with a polyline.
* Presenting the determined path to the user with data such as distance, steps, execution time, path.
* Creating a user-friendly interface.
* Encouraging the user to use it again and again (allow to user edit points).
* Add A\* search algorithm and compare performance.
* Calculate the roads and real distances.

**PROJECT FOLDER STRUCTURE**



1. **A\_star\_search.py :** Contains the implementation of the A\* pathfinding algorithm using heuristic search.
2. **app.py :** Main Flask application file handling web server and route calculations.
   1. Initialization
      * Loads graph data for pathfinding.
      * Creates Flask app.
   2. Core Functions
      * find\_nearest\_node(): Finds closest graph node to user’s clicked coordinates using geodesic.
      * /route : Serves index.html (Leaflet map interface).
   3. /route Endpoint
      * Input : Accepts POST with start/end coordinates.
      * Steps:
        + Finds nearest nodes to user’s points.
        + Runs Dijkstra’s and A\* Search algorithms.
        + Returns path details
        + Errors : Handles identical nodes and calculation failures.
   4. Output
      * JSON with path data for Leaflet to display
   5. Execution
      * Runs in debug model

1. **dijkstra.py :** Implementation of Dijkstra's shortest path algorithm.
2. **graph\_mentese\_custom.json :** Custom graph data file containing nodes, edges and coordinates.
3. static/

**style.css:** Stylesheet for the web interface.

1. templates/

**index.html:** Main webpage template with map and results display.

* + 1. Map Setup
       - Uses Leaflet.js to display OpenStreetMap.
       - Custom CSS styles the map and results panel.
    2. User Interaction
       - First click : Adds green marker (start point).
       - Second click : Adds red marker (end point) and triggers route calculation.
       - Clear function : Removes markers/path on new clicks
    3. Route Calculation
       - Sends coordinates to /route via POST request.
       - Displays the path (red line), distance, steps and execution time.
    4. Dynamic UI
       - Animates the path for visual feedback
       - Updates results panel in real time.

**DIJKSTRA’S ALGORITHM VS A\* SEARCH ALGORITHM**

**Hypothesis:** A\* algorithm will outperform Dijkstra in pathfinding efficiency for this application, demonstrating:

1. Faster computation (due to heuristic-guided search)

2. Fewer visited nodes (targeted exploration)

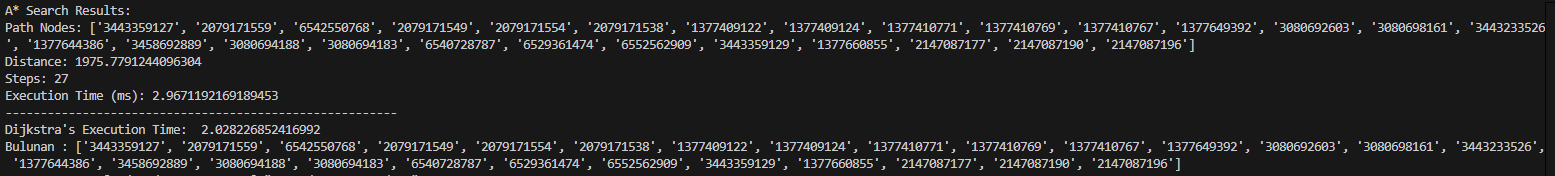
3. Lower execution time (especially for distant nodes)

...while maintaining equal path optimality when using admissible heuristics (Haversine distance).\*

**Rationale:**

A\* reduces search space by prioritizing nodes closer to the goal (heuristic + actual cost), whereas Dijkstra explores uniformly in all directions until reaching the target. This trade-off favors A\* in most spatial pathfinding scenarios.

**Results of our Application :**

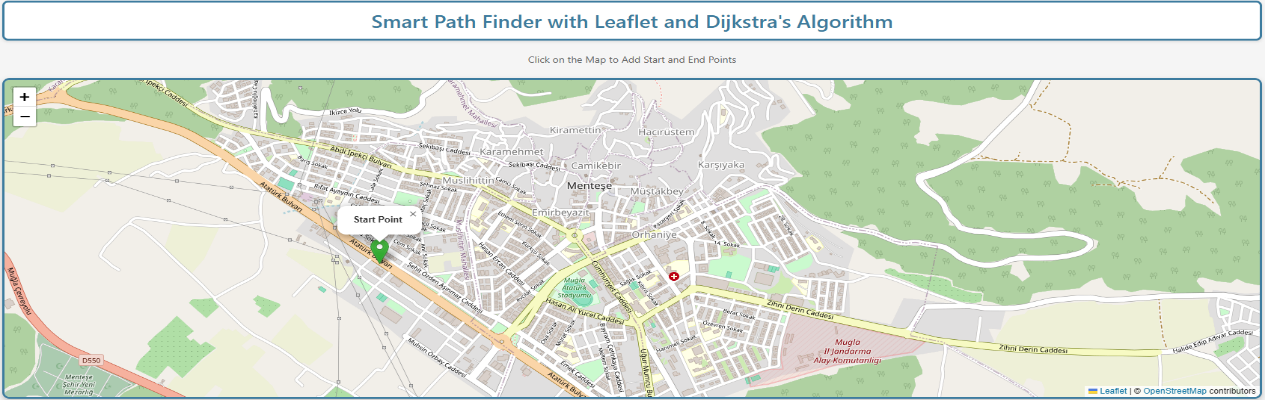
****

1. Both algorithms successfully reached the target point.
2. Both algorithms followed the same path.
3. Dijkstra's algorithm worked faster than A\* algorithm.

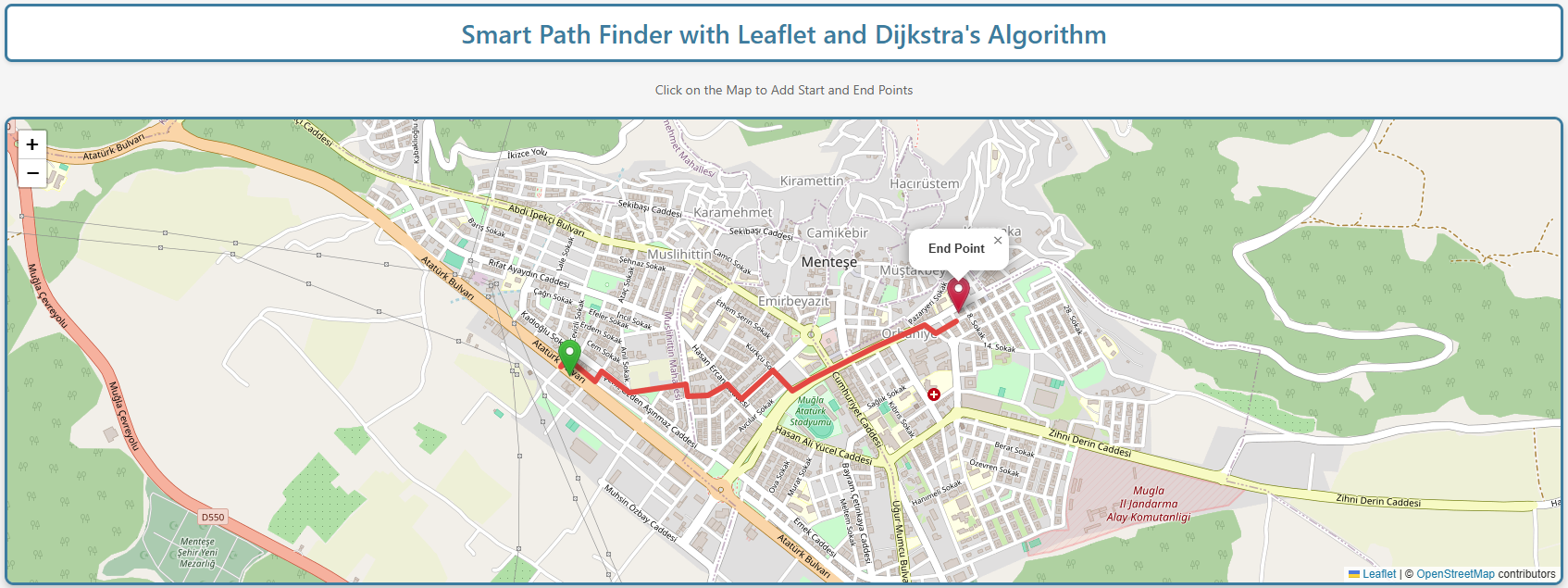
Possible reasons for item 3 :

1. Small Graph Size
   * With few nodes (<100), Dijkstra's simplicity (no heuristic calculations) may outperform A\*'s overhead.
   * A\*'s heuristic advantage scales with graph complexity.
2. Heuristic Computation Cost
   * Haversine distance calculations (in A\*) are computationally heavier than Dijkstra's basic edge-weight sums.
   * For small graphs, this overhead may negate A\*'s search-space reduction.
3. Implementation Factors
   * Priority queue inefficiencies in A\* (e.g., frequent `f\_cost` updates).
   * Suboptimal heuristic tuning (e.g., overestimating distances).
4. Dense vs. Sparse Graphs
   * If most nodes are interconnected, A\*’s heuristic provides less benefit.
   * Dijkstra’s uniform expansion may coincidentally inspect fewer nodes.
5. Hardware/Environment
   * Single-run variances (background processes, caching).

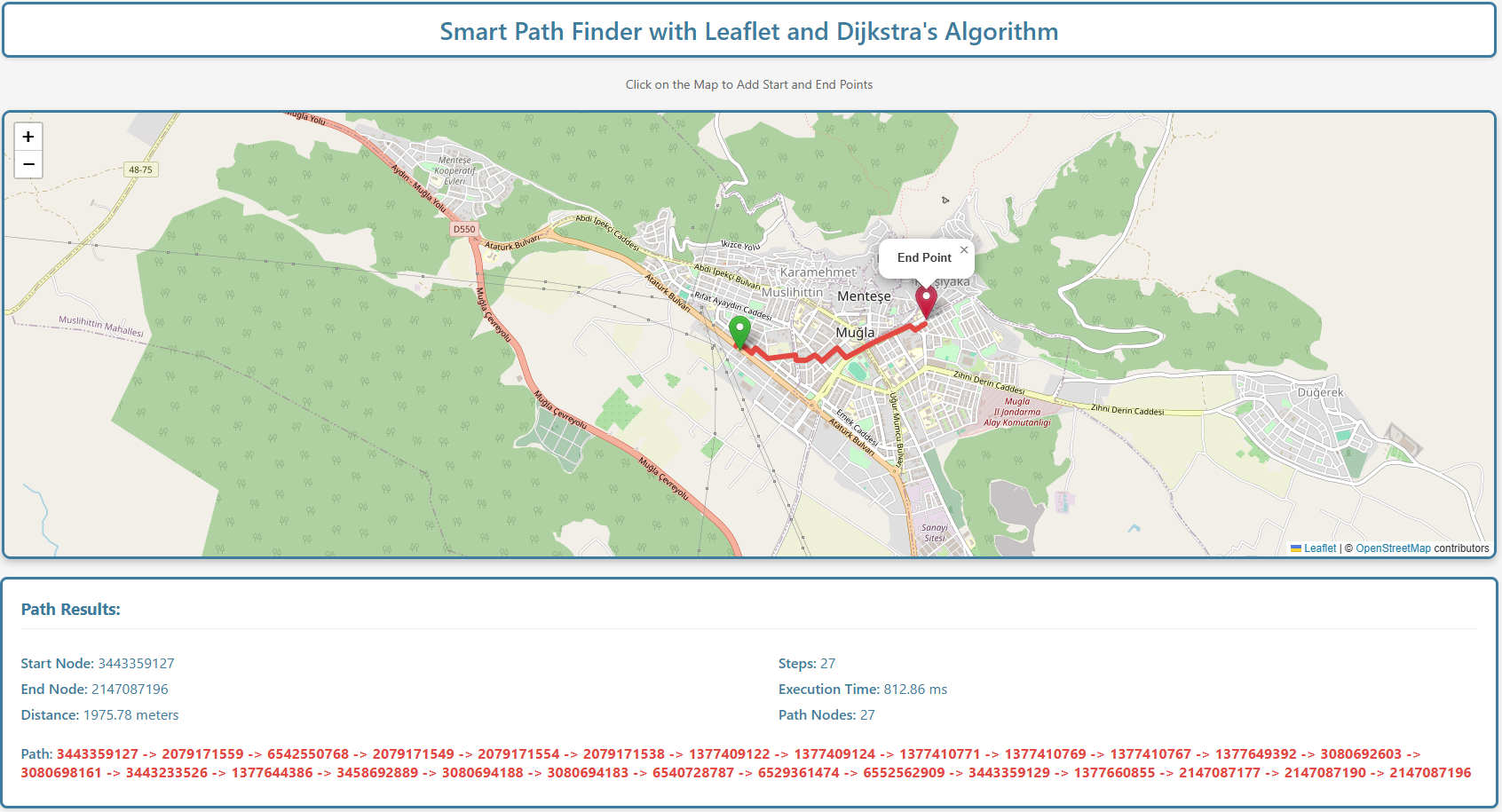
**USER GUIDE**

Click on the start point****

Click on the end point

****

You're ready to see results!

****

**If you want to create different routes, just click anywhere again.**