Project Report

LUO Weisheng 10756

#Overview

Our project is dedicated to developing a fully functional Paris map navigation system aimed at providing users with comprehensive geographic information services and a convenient travel experience. The project is implemented using a front-end and back-end separation architecture to ensure the system's efficiency and scalability.

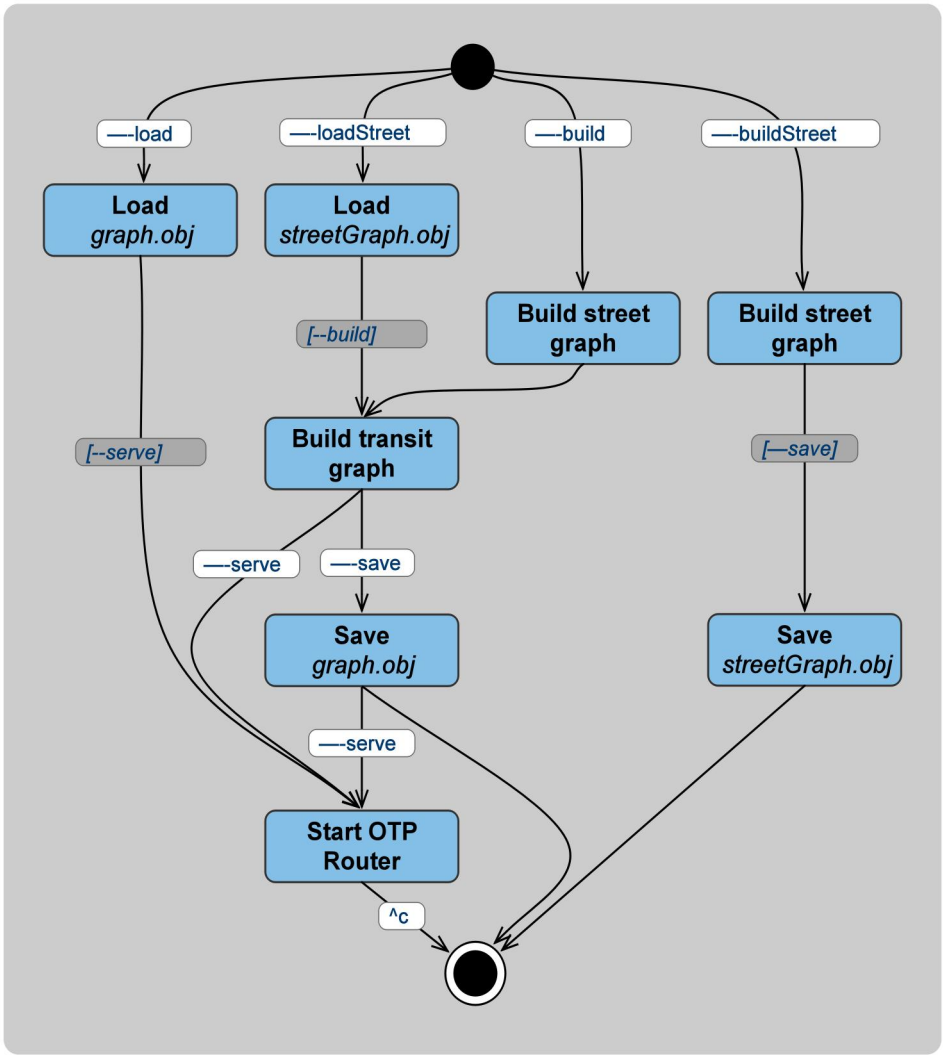
In the front-end, we adopted the Vue framework and combined it with the Element Plus component library to build an intuitive, modern user interface that supports multi-terminal access. Through this interface, users can perform location searches, obtain shortest route plans, and visually view relevant information on the map. Additionally, we integrated the Baidu Map plugin to provide powerful map display capabilities, supporting various map view switches and detailed geographic information displays.

For the back-end, we chose OpenTripPlanner2 as the core tool. This powerful open-source platform not only supports the loading of basic maps and street maps but can also handle complex public transportation route maps. It offers flexible route planning functions, supporting various path planning algorithms, and can calculate the optimal route according to user needs. This tool also has high scalability, allowing for further development of additional features, such as multi-modal travel planning and real-time traffic information integration.

The entire system design fully considers user needs and practical application scenarios, not only realizing the core functions but also reserving space for future expansion, making it adaptable to evolving user requirements and technological advancements.

#Project Structure

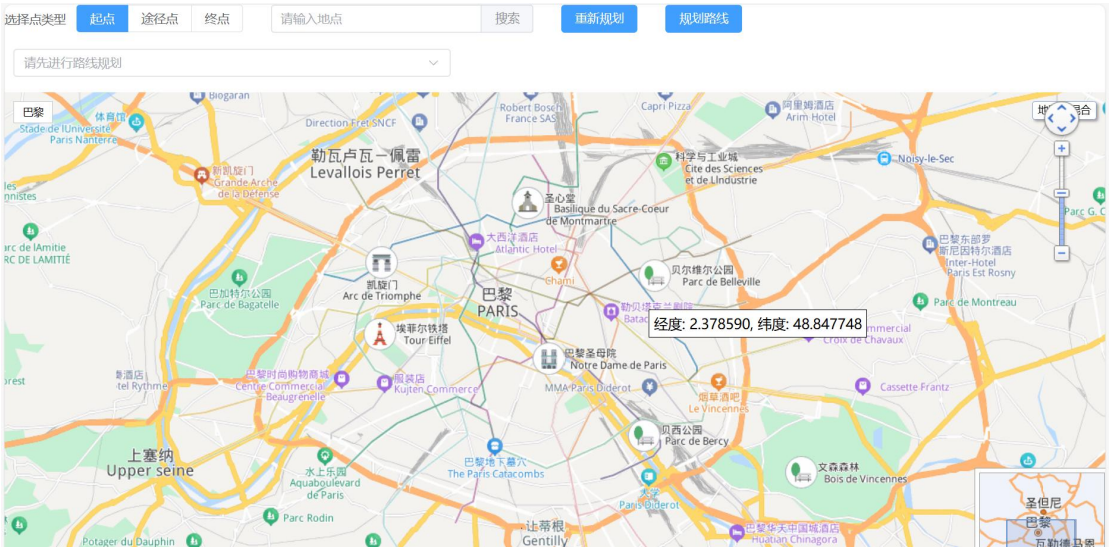
The project is structured with a front-end and back-end separation model. For the front-end, we used the Vue framework and integrated Baidu Map's API for interface display. Real-time updates of views are achieved by sending GET requests to Baidu Map's servers. After users provide a starting and ending point, a boundary check filters out invalid data. Then, we use the promise-based HTTP client library axios to send requests to the back-end server for route planning. Once the back-end server receives the starting and ending points (assuming the graph has already been constructed on the server), it will use the $A^{\ *}$ algorithm (as the project standardizes on using the $A^{\ *}$ algorithm for path planning). After obtaining the planning result, the front-end filters out duplicate routes and continues to call Baidu Map's API for interface updates. This is the execution flow of our project. To better illustrate the back-end server's execution logic, we have provided a flowchart:



The open-source tool OpenTripPlanner we adopted uses Graphhopper as the graph node search engine, with the core using the $A^{\ *}$ algorithm for shortest route planning. Below, we will provide a detailed introduction of both the front-end and back-end.

#front-end

Our front-end is designed based on the **Vue** framework, and uses the **Element Plus** component library to build the interface. Our front-end interface effect is shown in the figure:



Our front-end interface offers the following key features:

Main Features:

Loading State (v-loading): The el-container element utilizes the v-loading directive. When the loading variable is set to true, a loading indicator and the text "Please wait, route planning in progress..." are displayed, informing users that the route planning is underway.

Input Form: The form includes radio buttons for selecting point types (starting point, waypoint, destination), an input box for searching locations, and buttons for resetting and generating routes. The form is built using the el-form and related Element Plus components.

Map Display: The baidu-map component renders the Baidu Map and incorporates various map controls such as map type selection, navigation control, and a thumbnail map to enhance user interaction.

Dynamic Markers: Based on user clicks and search results, dynamic markers are generated on the map.

Route Planning: The getRoute function handles the route generation logic, creating routes between the selected points and sending the request to the routing service (such as OpenTripPlanner). It processes the returned routes, removing duplicates and optimizing factors such as time, cost, and the number of transfers.

Polyline Rendering: Routes are displayed on the map as polylines in various colors, with different colors representing different routes. Users can select a specific route through a dropdown menu (el-select), and the corresponding polyline will be highlighted on the map.

Coordinate Labels: The map displays the latitude and longitude of the mouse position in real-time, assisting users in accurately selecting points.

#Back end

In transportation application scenarios, whether it's for consumer-facing travel services or applications geared toward travel analysis, there is often a need to perform shortest path or route planning, as well as public transit transfer algorithms. These tasks are typically accomplished by invoking open APIs from Baidu Maps, as Google services are not accessible in mainland China, leaving Baidu Maps as the alternative.

However, the usage limits imposed on open map APIs can often fail to meet the actual computational demands. In such cases, we can opt for **OpenTripPlanner2** to perform offline computations. Below, we will use walking navigation as an example to introduce our back-end process, covering aspects such as data input, graph structure construction, and the A* algorithm.

#date

In our project, we utilized OpenStreetMap (OSM) as the data input. OSM is a highly detailed map dataset and a global, freely accessible online map collaboration project. Its goal is to create a world map that can be freely edited and is open to everyone, providing a convenient navigation solution for mobile devices. OSM data is open-source, allowing users to download and use this global map database for free.

#OSM data structure

OSM data is composed of spatial data and attribute data. The spatial data primarily includes three basic elements: Nodes, Ways, and Relations. These elements together form a complete map representation:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.8.8 (2798843 spike-07.openstreetmap.org)"
copyright="OpenStreetMap and contributors"
attribution="http://www.openstreetmap.org/copyright"
license="http://opendatacommons.org/licenses/odbl/1-0/">
 <bounds minlat="31.3618700" minlon="121.8012200" maxlat="31.3620700"
maxlon="121.8015200"/>
 <node id="7603536329" visible="true" version="1" changeset="86368470"
timestamp="2020-06-08T19:00:07Z" user="Reboot01" uid="9856191" lat="31.3618772"
lon="121.8012886"/>
 <node id="7603536330" visible="true" version="1" changeset="86368470"
timestamp="2020-06-08T19:00:07Z" user="Reboot01" uid="9856191" lat="31.3619399"
lon="121.8014870"/>
 <node id="7603536331" visible="true" version="1" changeset="86368470"
timestamp="2020-06-08T19:00:07Z" user="Reboot01" uid="9856191" lat="31.3620542"
lon="121.8014375"/>
 <node id="7603536332" visible="true" version="1" changeset="86368470"
timestamp="2020-06-08T19:00:07Z" user="Reboot01" uid="9856191" lat="31.3619915"
lon="121.8012390"/>
 <way id="813970381" visible="true" version="1" changeset="86368470"
timestamp="2020-06-08T19:00:07Z" user="Reboot01" uid="9856191">
  <nd ref="7603536329"/>
  <nd ref="7603536330"/>
  <nd ref="7603536331"/>
  <nd ref="7603536332"/>
  <nd ref="7603536329"/>
  <tag k="building" v="yes"/>
 </way>
</osm>
```

Node: Stores the latitude and longitude information on the map and is used to represent locations such as landmarks, shops, or restaurants. These nodes can also be used as part of a way.

Way: Consists of an ordered sequence of nodes, which can be represented as open lines (e.g., streets) or closed polygonal areas (e.g., parks, buildings). Ways are used to depict linear features (like roads) or area features (like lakes, forests).

Relation: Defines the relationships between different nodes, ways, or other relations. They are used to represent complex map features, such as a boundary composed of multiple ways, or to describe the intricate relationships of highways or railway lines.

Attribute Data

In OSM, attribute data is described using Tags. Tags exist as key-value pairs that assign specific meanings and properties to each spatial data element. For example, highway=residential is used to define a residential road, while building=yes indicates a building. These tags cannot exist independently; they must be attached to basic elements such as nodes, ways, or relations.

Other OSM Features

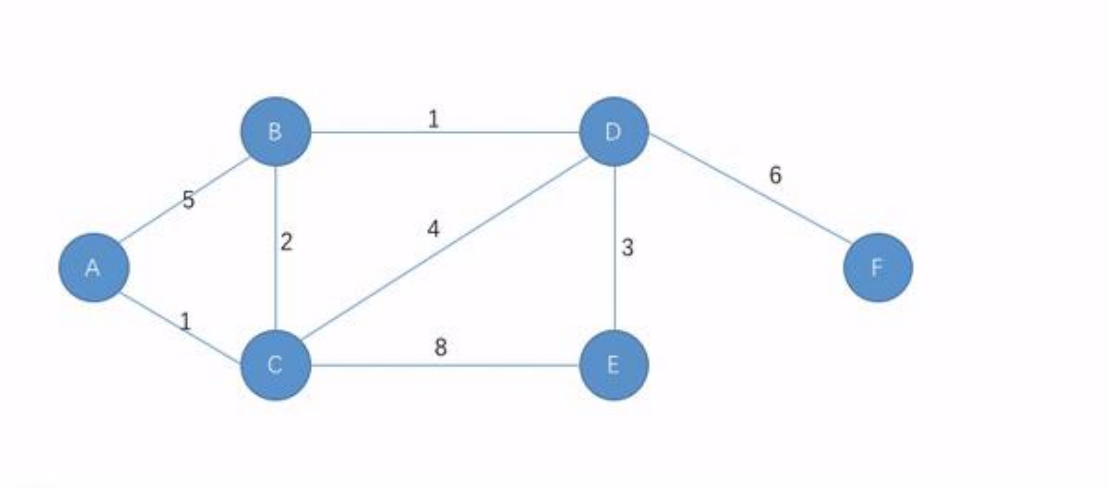
In addition to basic map data, OpenStreetMap.org also includes map annotations and GPS tracks. These resources help users report errors or outdated information on the map and provide real-world track data to improve map accuracy.

Finally, OSM map layers are rendered by different rendering engines based on these basic data elements. Different engines can produce different styles of map imagery, known as tiles, which offer users a variety of visual effects and data presentations.

By leveraging OSM, our project can utilize this open and continuously updated map resource to deliver precise and diverse map services, providing users with richer geographic information and a better navigation experience.

Constructing the Graph Structure

To convert OSM data into a graph structure, nodes in the graph represent locations on the map (expressed in latitude and longitude), while edges represent roads. On this basis, we can introduce weights to represent distance, travel time, transportation costs, and other factors to facilitate the calculation of the shortest path or minimal cost. This can be represented as an undirected weighted graph (though in practice, the data also needs to consider different public transportation routes, and there may be issues such as missing road networks in the original data). Here, we provide a brief introduction focused on the core path planning algorithm of the OpenTripPlanner tool, as illustrated below.



Logic of A*

Unlike the undirected graph mentioned above, a real transportation network map differs in that the way to expand the sub-nodes in a real transportation network is to find the nearest connected nodes to the parent node, rather than the four nodes directly above, below, left, and right of the node. The distance between these nodes is not fixed at 1 but is calculated based on the latitude and longitude of the two nodes. Suppose there are two points, A and B, with latitudes and longitudes $(lat1, lon1)$ and $(lat2, lon2)$ respectively. The formula S represents the distance (in meters) between the two points, where a represents the difference in latitude between A and B $(lat1 - lat2)$, and b represents the difference in longitude between the two points $(lon1 - lon2)$. The value 6378.137 represents the Earth's radius (in kilometers). Using this formula, the calculated distance accuracy and the actual map distance deviation are within ±0.2 meters:

$$S = 2 \times \arcsin\left(6378.137 \times \sqrt{\sin^2(a/2) + \cos(lat1) \times \cos(lat2) \times \sin^2(b/2)}\right)$$

The **A* algorithm** is a widely used pathfinding and graph traversal algorithm that efficiently finds the shortest path between two points. It combines features of both **Dijkstra's Algorithm** and **Greedy Best-First Search**, using a heuristic to guide its search and improve performance.

The main steps of the A* algorithm are as follows:

1. **Initialization:**
    - Create two sets:
        - **Open Set:** Contains nodes to be evaluated (starts with the initial node).
        - **Closed Set:** Contains nodes that have already been evaluated.
    - Initialize the **g-score** (the cost from the start node to the current node) for the start node as 0.
    - Calculate the **f-score** (estimated total cost from start to goal through the current node) for the start node using the heuristic function.

2. **Loop through Open Set:**
    - While the Open Set is not empty:
        - Select the node in the Open Set with the lowest f-score; this node is considered the **current node**.
        - If the current node is the goal node, reconstruct the path and terminate the algorithm.
        - Move the current node from the Open Set to the Closed Set.
        - For each neighbor of the current node:
            - If the neighbor is in the Closed Set, skip it.
            - Calculate the tentative g-score for the neighbor.
            - If the neighbor is not in the Open Set or the tentative g-score is less than the previously recorded g-score:
                - Update the neighbor's g-score to the tentative g-score.
                - Calculate the neighbor's f-score using the updated g-score and heuristic estimate.
                - Set the current node as the neighbor's parent node.
                - If the neighbor is not in the Open Set, add it.

3. **Heuristic Function:**
    - The heuristic estimates the cost to reach the goal from a given node. A common heuristic for geographical data is the **Haversine formula**, which calculates the great-circle distance between two points on a sphere given their longitudes and latitudes. This provides an admissible and consistent heuristic, ensuring the optimality of the A* algorithm.

    Haversine Formula:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

    - **Where:**
        - d is the distance between the two points.
        - r is the radius of the Earth (mean radius = 6,371 km).
        - ϕ_1, ϕ_2 are the latitudes of point 1 and point 2 in radians.
        - λ_1, λ_2 are the longitudes of point 1 and point 2 in radians.

4. **Path Reconstruction:**
    - Once the goal node is reached, backtrack from the goal node to the start node using the parent references to reconstruct the optimal path.

Advantages of the A* Algorithm:

- **Efficiency:** Finds the shortest path faster than many other algorithms by using heuristics to guide the search.
- **Optimality:** Guarantees the shortest path if the heuristic is admissible (does not overestimate the actual cost).
- **Flexibility:** Can be adapted for various scenarios by modifying the cost and heuristic functions.

Applications:

- **GPS Navigation:** Calculating optimal routes between locations.
- **Robotics:** Path planning for autonomous movement.
- **Game Development:** NPC movement and decision-making.
- **Network Routing:** Finding optimal data transfer paths.

Conclusion: By converting OSM data into a graph structure and applying the A* algorithm with an appropriate heuristic like the Haversine formula, we can efficiently compute optimal routes for navigation purposes. This approach allows for offline computations, overcoming the limitations of external API calls and providing flexibility to tailor the pathfinding process to specific needs and constraints.

Implementation Steps

1. **Initialization:**
   - Create the start node and goal node, initializing the g, h, and f values.
   - Create an open list (using a priority queue) and a closed list, adding the start node to the open list.
2. **Node Processing:**
   - Remove the node with the smallest f value from the open list and treat it as the current node.
   - Check if the current node is the goal node. If it is, backtrack the path and return it.
3. **Exploration of Adjacent Nodes:**
   - Calculate the adjacent nodes in four directions (up, down, left, right) from the current node.
   - For each adjacent node, check if it is a valid node (i.e., within the grid boundaries and not an obstacle).
4. **Updating Adjacent Nodes:**
   - For each valid adjacent node, if it is not in the closed list, calculate its g, h, and f values.
   - Use the `add_to_open` function to determine whether to add the adjacent node to the open list.
5. **Moving to the Closed List:**
   - Move the current node into the closed list.
   - If the open list is empty, it means no path was found, and the function should return failure.
6. **Repeat the Process:**
   - Repeat steps 2-5 until the goal node is found or the open list is empty.

This report includes a simple Python code implementation to simulate the above steps:

```
import heapq
import matplotlib.pyplot as plt
import numpy as np


class Node:
    """Node class representing each point in the search tree."""
    def __init__(self, parent=None, position=None):
        self.parent = parent              # Parent node of this node
        self.position = position        # Node's position in the maze (coordinates)
        self.g = 0                          # G-value: cost from the start node to the current node
        self.h = 0                          # H-value: estimated cost from the current node to the goal node
        self.f = 0                          # F-value: sum of G and H values, i.e., the total estimated cost

    # Compare if two nodes have the same position
    def __eq__(self, other):
        return self.position == other.position

    # Define less-than operation for comparison in priority queue
    def __lt__(self, other):
        return self.f < other.f

def astar(maze, start, end):
```

```
    """A* algorithm implementation to find the shortest path from start to end in the maze."""
    start_node = Node(None, start)    # Create the start node
    end_node = Node(None, end)          # Create the goal node
    open_list = []                      # Open list to store nodes to be evaluated
    closed_list = []                    # Closed list to store evaluated nodes
    heapq.heappush(open_list, (start_node.f, start_node))    # Add the start node to the open list
    print("Added start node to the open list.")

    # Loop as long as there are nodes in the open list
    while open_list:
        current_node = heapq.heappop(open_list)[1]    # Pop and return the node with the smallest f-value from the open list
        closed_list.append(current_node)                  # Add the current node to the closed list
        print(f"Current node: {current_node.position}")

        # If the current node is the goal node, backtrack the path
        if current_node == end_node:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            print("Goal node found, returning the path.")
            return path[::-1]    # Return the reversed path, i.e., from start to end
```

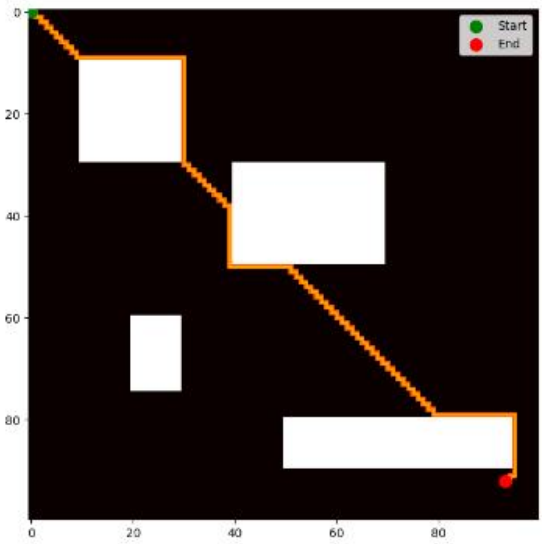
```
        # Get adjacent nodes around the current node
        (x, y) = current_node.position
        neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]

        # Traverse adjacent nodes
        for next in neighbors:
            # Ensure adjacent node is within maze bounds and not an obstacle
            if 0 <= next[0] < maze.shape[0] and 0 <= next[1] < maze.shape[1]:
                if maze[next[0], next[1]] == 1:
                    continue
                neighbor = Node(current_node, next)  # Create adjacent node

                # Skip processing if adjacent node is already in the closed list
                if neighbor in closed_list:
                    continue

                neighbor.g = current_node.g + 1  # Calculate the G-value for the adjacent node
                neighbor.h = ((end_node.position[0] - next[0]) ** 2) + \
                             ((end_node.position[1] - next[1]) ** 2)  # Calculate the H-value
                neighbor.f = neighbor.g + neighbor.h  # Calculate the F-value

                # If the adjacent node's new F-value is smaller, add it to the open list
```

```
                    if add_to_open(open_list, neighbor):
                        heapq.heappush(open_list, (neighbor.f, neighbor))
                        print(f"Added node {neighbor.position} to the open list.")
                else:
                    print(f"Node {next} is out of bounds or an obstacle.")

    return None  # Return None if no path is found

def add_to_open(open_list, neighbor):
    """Check and add a node to the open list."""
    for node in open_list:
        # If a node with the same position exists in the open list and has a lower G-value, do not add this node
        if neighbor == node[1] and neighbor.g > node[1].g:
            return False
    return True  # Return True to add the node to the open list if it doesn't exist

def visualize_path(maze, path, start, end):
    """Visualize the found path on the maze."""
    maze_copy = np.array(maze)
    for step in path:
        maze_copy[step] = 0.5  # Mark points on the path
    plt.figure(figsize=(10, 10))
```

```
    # Display passages in the maze as black, obstacles as white
    plt.imshow(maze_copy, cmap='hot', interpolation='nearest')

    # Extract x and y coordinates from the path
    path_x = [p[1] for p in path]  # Column coordinates
    path_y = [p[0] for p in path]  # Row coordinates

    # Plot the path
    plt.plot(path_x, path_y, color='orange', linewidth=2)

    # Plot start and end points
    start_x, start_y = start[1], start[0]
    end_x, end_y = end[1], end[0]
    plt.scatter([start_x], [start_y], color='green', s=100, label='Start', zorder=5)  # Start point as green dot
    plt.scatter([end_x], [end_y], color='red', s=100, label='End', zorder=5)  # End point as red dot

    # Add legend
    plt.legend()

    # # Hide axes
    # plt.axis('off')

    # Show the image
```

```
    plt.show()


# Set maze dimensions
maze_size = 100


# Create an empty maze, all set to 0 (indicating passable)
maze = np.zeros((maze_size, maze_size))


# Define some obstacle blocks, each as a rectangle
obstacle_blocks = [
    (10, 10, 20, 20),    # (y_start, x_start, height, width)
    (30, 40, 20, 30),
    (60, 20, 15, 10),
    (80, 50, 10, 45),
]


# Set obstacles in the maze
for y_start, x_start, height, width in obstacle_blocks:
    maze[y_start:y_start+height, x_start:x_start+width] = 1


# Set start and end points
start = (0, 0)
end = (92, 93)
```

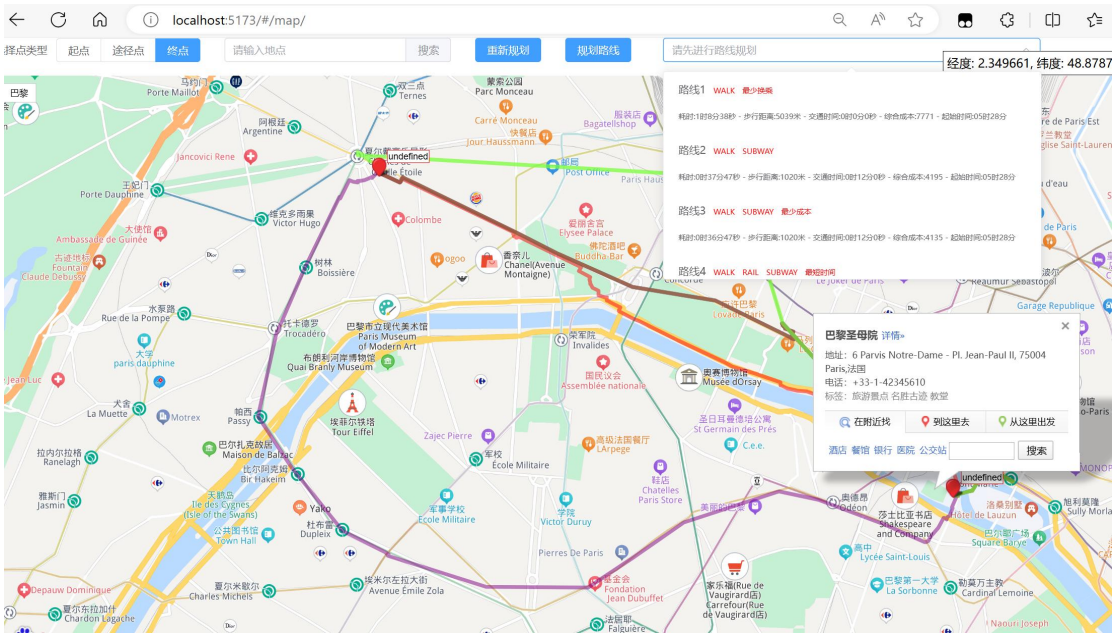
```
# Ensure start and end points are not obstacles
maze[start] = 0
maze[end] = 0

# Output part of the maze to confirm obstacle placement
print("View of the top-left 10x10 section of the maze:")
print(maze[:10, :10])

path = astar(maze, start, end)
if path:
    print("Path found:", path)
    visualize_path(maze, path, start, end)
else:
    print("No path found.")
```



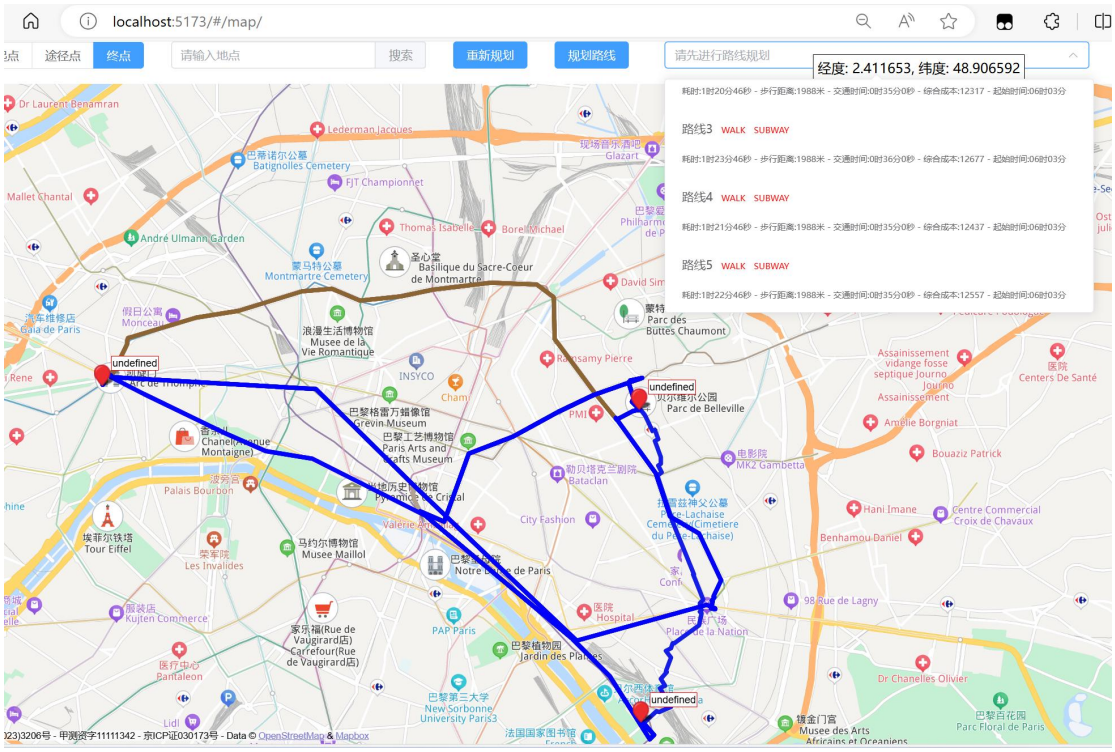
The diagram illustrates the path planning results using the A* algorithm in a two-dimensional grid maze. In the maze, the white rectangular areas represent obstacles, while the black background represents passable space. The path is shown as an orange line, starting from the green dot in the upper left corner (starting point), traversing the maze, bypassing obstacles, and finally reaching the red dot in the lower right corner (end point). This visualization clearly demonstrates the process of the algorithm finding the shortest path from the start position to the goal position while successfully avoiding all obstacles.

Project Testing

This project is built on Vue and the OpenTripPlanner tool to create an online map application capable of providing navigation for walking, driving, and public transportation. Below, we present both two-point and multi-point path planning tests.



As can be seen in the figure, after given the starting point and the end point, our system can perfectly calculate different routes and display them.



The figure shows the route planning of the starting point, the end point and the waypoint.