Analysis Report: Routing Module

Overall Project: All-In-One Urban Mobility Platform

Key Responsibilities: Algorithm design, API integration, Performance Testing

1. Context

The "All-In-One Urban Mobility Platform" requires a core module capable of calculating routes for users (pre-trip routing). This module is the foundation for the "Planning" feature of the entire project, providing suggested routes (e.g., fastest, shortest) based on different modes of transport. This component is responsible for fetching raw map data, processing it into a computable graph structure, and implementing efficient routing algorithms.

2. Problem Definition

The core problem is: How to calculate and find the shortest path between any two coordinates (longitude, latitude) within the Ho Chi Minh City area, distinguishing between vehicle types (car, motorbike, bicycle, walking) based on static map data from OpenStreetMap (OSM).

Stakeholders

- **Routing Team:** Responsible for building, testing, and maintaining the algorithm's accuracy.
- **UI/UX & Integration Team:** An internal "client" that needs a stable API from this module to call and display results (route, distance) to the end-user.
- **End-users:** Expect to receive an accurate, reasonable, and traffic-compliant route.

Objectives

- Successfully build 4 separate road network graphs for 4 vehicle types from OSM data.
- Successfully implement the A* (A-star) algorithm capable of finding the shortest path on the graph.
- Ensure the average response time (calculation) for a routing query is under 3 seconds (after the graph is loaded).
- Route accuracy: 100% compliance with defined rules in the graph (e.g., one-way streets).

Input Data (Input)

- **Input (from external APIs):**
  - Raw map data from OpenStreetMap (via Overpass API).
  - Geocoding data from Nominatim (to convert addresses).
- **Input (directly from user):**
  - Start location (text format, e.g., "Saigon Central Post Office").
  - End location (text format, e.g., "University of Science").
  - Vehicle selection (to choose the corresponding JSON data file).
- **Input (for the algorithm core - post-processing):**

    - Start coordinates (Latitude, Longitude) - (converted by `choose_location`).
    - End coordinates (Latitude, Longitude) - (converted by `choose_location`).

Expected Output

- A list of coordinate pairs (Latitude, Longitude) representing the detailed route from start to end.
- Total distance of the route (in meters or km).
- A `shortest_path_map.html` file visualizing the route on a map (`folium`).

Constraints

- OSM data is static; it **does not include real-time traffic conditions** (unlike a Google Maps API solution).
- OSM data is very large and complex; the graph must be filtered and built carefully.
- Loading the entire graph into memory can be resource-intensive (RAM).
- The A* algorithm requires a good heuristic (estimation) function.

3. Guiding Questions

3.1 What do we know?

- We can get free, high-quality map data from OpenStreetMap (OSM).
- Overpass API is a powerful tool for querying and extracting this OSM data.
- Classic routing algorithms (Dijkstra, A*) are well-suited for this problem.
- The `geopy` library can convert text addresses to coordinates (Nominatim).
- The `folium` library can be used to quickly visualize results.
- The `haversine` formula is used to calculate the "as-the-crow-flies" distance between two GPS coordinates.

3.2 What do we want?

- A script (`DownloadMapDataHCMC.py`) to download and save OSM data.
- A function (`build_graph_from_json`) to convert raw JSON data into a graph structure (adjacency list) that Python understands.
- A function (`astar`) that efficiently implements the A* algorithm using `heapq`.
- A function (`find_nearest_node`) to "map" a user's GPS coordinates to the nearest node (intersection) on the graph.
- Integrate all of this into one master function (`find_shortest_path`).

3.3 What are the rules or constraints?

- The graph must correctly handle road attributes, especially `oneway=yes`.
- If a road is two-way, edges must be added in both directions.
- Calculations must be based on actual distance (calculated by `haversine` between nodes), not the heuristic distance.
- The solution must run using Python and open-source libraries.

3.4 What's missing?

- **Real-time traffic data:** This is the biggest shortcoming compared to services like Google Maps.
- The route is the **"shortest path," not the "fastest path"** based on traffic conditions.
- **Speed limits:** The current graph is based only on distance. To calculate ETA (Estimated Time of Arrival), data on speed limits (`maxspeed`) for each road segment is needed.
- **Graph Database:** Loading large JSON files into memory will not scale to a national level. A graph database solution (e.g., Neo4j) would be necessary for a larger version.

4. Resource Requirements

4.1 Hardware Requirements

- **4.1.1 Development Team:**
  - Personal computer (Laptop/PC) with a minimum of 8GB RAM (16GB recommended) to process and load the JSON files (which can be several hundred MB) into memory to build the graph.
- **4.1.2 Deployment Infrastructure (For API):**
  - A server (or container) to run the Python application (e.g., Flask, FastAPI) as a backend.
  - Requires sufficient RAM to "hold" (cache) the built graph in memory, avoiding the need to read and rebuild the graph for every API call.

4.2 Software Requirements

- **4.2.1 Development Tools and Environment:**
  - **Language:** Python 3.x.
  - **IDE:** VS Code (or equivalent).
  - **Source Control:** Git, GitHub.
- **4.2.2 Libraries (Dependencies):**
  - `requests`: To call the Overpass API.
  - `json`, `math`, `heapq`: Standard Python libraries for data handling, math, and priority queues (for A*).
  - `geopy`: For address geocoding.
  - `folium`: For visualization (demo).
  - `overpy`: (Optional) A specialized library for working with the Overpass API (though your code uses `requests` directly).

5. Third-Party Resources and Services

- **5.1. OpenStreetMap (OSM):**
  - **Requirement:** Base map data source.
  - **Cost:** Free.
- **5.2. Overpass API (e.g., overpass.kumi.systems):**
  - **Requirement:** Public API service to query and extract OSM data.
  - **Cost:** Free, but has rate limiting and query timeout limits.
- **5.3. Geocoding Service (Nominatim):**

- **Requirement:** Integrated within the `geopy` library to convert addresses (`start_text`, `end_text`) to (lat, lon).
- **Cost:** Free, but has strict frequency limits (usually 1 request/second).

6. Requirements Definition

6.1 Functional Requirements

- **FR1 (Data Download):** The system must be able to download road network data for HCMC (ID: 3601973756) from the Overpass API.
- **FR2 (Data Filtering):** Queries must accurately filter 4 road types for: Cars, Motorbikes, Bicycles, and Pedestrians (as in `DownloadMapDataHCMC.py`).
- **FR3 (Graph Building):** The system must read the downloaded JSON file and build a graph structure (adjacency list) in memory.
- **FR4 (One-Way Handling):** When building the graph, the system must check the `oneway` tag. If 'yes', create only a one-way edge; otherwise, create edges in both directions (`build_graph_from_json`).
- **FR5 (Distance Calculation):** The weight of each edge in the graph must be the actual geographical distance, calculated using the `haversine` formula.
- **FR6 (Node Snapping):** The system must provide a function (`find_nearest_node`) to find the nearest node in the graph to any given GPS coordinate pair.
- *FR7 (A Algorithm):** The system must implement the A* (`astar`) algorithm to find the shortest path, using `haversine` as the heuristic function.
- **FR8 (Geocoding):** The system must allow users to input text-based addresses and convert them to coordinates (`choose_location`).

6.2 Non-Functional Requirements

- **NFR1 (Performance):** The calculation time of the `astar` function (after the graph is loaded) must be under 3 seconds for typical inner-city queries.
- **NFR2 (Accuracy):** The generated route must be logically correct (no entering forbidden roads, no going against one-way streets).
- **NFR3 (Maintainability):** Code must be clearly organized (separating functions: `haversine`, `astar`, `build_graph_from_json`...).
- **NFR4 (Reliability):** Must handle exceptions: node not found (`find_nearest_node` returns None), or no path found between two points (`astar` returns an empty array).

7. Work Breakdown Structure

7.1. OSM API Integrate (OpenStreetMap API Integration)

- **Objective:** Fetch raw map data from OpenStreetMap and integrate a geocoding service to get input coordinates.
- **Responsible Files:** `DownloadMapDataHCMC.py`, `FindShortestPathInHCMC.py`
- **Sub-tasks (Tasks):**
  - **Overpass API Querying (Data Querying):**
    - Write 4 specialized Overpass queries to filter road data in HCMC (area: 3601973756) for 4 vehicle types:
      - `get_roads_query_hcm_car()` (Car)
      - `get_roads_query_hcm_motorbike()` (Motorbike)
      - `get_roads_query_hcm_bicycle()` (Bicycle)

        - `get_roads_query_hcm_walk()` (Pedestrian)
    - **Data Extraction:**
        - Implement the `extract_raw_data_from_OSM()` function using `requests` to call the Overpass API, fetch JSON data, and save it to corresponding files (car_data.json, motorbike_data.json...).
    - **Geocoding Integration:**
        - Implement the `choose_location()` function using `geopy.Nominatim` (a service based on OSM data) to convert text addresses (e.g., "Saigon Central Post Office") into coordinates (latitude, longitude).

7.2. Routing Algorithm (Routing Algorithm Design)

- **Objective:** Build a graph from raw OSM data and implement an efficient routing algorithm (A*) on that graph.
- **Responsible File:** `FindShortestPathInHCMC.py`
- **Sub-tasks (Tasks):**
    - **Graph Building:**
        - Implement the `build_graph_from_json()` function to read the JSON file (downloaded in 7.1) and convert it into a graph structure (adjacency list).
        - This function must process 'nodes' (vertices), 'ways' (edges), and most importantly, the `oneway` tag.
    - **Edge Weight Calculation:**
        - Implement the `haversine()` function to calculate the geographical distance (actual distance) between two nodes.
        - This will be the weight of the edges in the graph.
    - **Node Snapping:**
        - Implement the `find_nearest_node()` function to "map" GPS coordinates (from 7.1) to the nearest existing node (intersection) in the built graph.
    - **Algorithm Core:**
        - Implement the `astar()` (A-star) algorithm using `heapq` (priority queue).
        - This algorithm uses `haversine` (as-the-crow-flies distance to destination) as its heuristic function to optimize search speed.
        - Implement the `dijkstra()` function (for comparison or as a fallback).
    - **Master Function:**
        - Implement the `find_shortest_path()` function to link all preceding steps: Load data -> Build graph -> Find start/end nodes -> Run A* -> Return route.

7.3. Performance Testing

- **Objective:** Ensure the routing module operates stably and meets the defined Non-Functional Requirements (NFRs), especially NFR1 (response speed).
- **Responsible File:** (New file needed, e.g., `test_performance.py`)
- **Sub-tasks (Tasks):**
    - **Graph Load Testing:**
        - Measure the time and peak RAM usage when the `build_graph_from_json()` function processes the data files (especially the largest one, e.g., `walk_data.json`).

- **Algorithm Latency Testing:**
  - Write a script to automatically run the `astar()` function with 100 random coordinate pairs (including short inner-city and long suburban routes).
  - Calculate the average, maximum, and minimum response times.
  - Objective: Verify average response time is < 3 seconds (meeting NFR1).
- **Accuracy/Sanity Testing:**
  - Use the `visualize_path_on_map()` function to visually inspect 5-10 sample routes.
  - Objective: Ensure routes are reasonable and logically sound (e.g., respecting defined one-way streets).

8. Execution Plan

Below is the execution plan based on the new work breakdown:

No.	Task	Objective	Expected Output	Status
1	**OSM API Integrate**	Fully integrate APIs for raw data (Overpass) and geocoding (Nominatim).	JSON files (`car_data.json`...) and `choose_location()` function operating stably.	Completed
2	**Routing Algorithm**	Implement and finalize the A* routing core and graph-building logic from A-Z.	`find_shortest_path()` function returns an accurate route.	Completed
3	**Performance Testing**	Measure and verify the algorithm's performance and accuracy against NFRs.	Test report confirming average query time < 3 seconds and route accuracy.	Next