COMP5008 - Project Report

Autonomous Vehicle Management System (AVMS)

Unit Coordinator

Dr. Mahbuba Afrin

Project Report by

Nima Dorji Moktan : 21522132

October 27, 2024

Contents

1 Implementation of Autonomous Vehicle Management System (AVMS)

1.1 Introduction

The Autonomous Vehicle Management System (AVMS) is designed to manage a fleet of vehicles, track their locations, and help navigate between different locations or destinations. As urban mobility increases, effective vehicle management systems become essential for logistics, transportation services, and personal use. This report outlines the implementation strategy, detailing the class structures, methods, functionality and algorithms used in AVMS. Furthermore, it discusses the use case and challenges encountered during development. It also provides a flowchart illustrating the system's architecture, and analyzes the efficiency of the implemented algorithms while suggesting potential improvements.

1.2 Implementation Strategy

1.2.1 Class Structure

The AVMS is built upon a modular architecture consisting of several interconnected classes, each responsible for specific functionalities. The whole AVMS is implemented using Python programming language. The primary classes defined are: **Vehicle Class**

- **Purpose:** Represents an individual vehicle and holds essentials about its state, including location, distance to the destination, and battery level.
- **Attributes:**
  - vehicle_id: Unique identifier for the vehicle.
  - location: Current geographic location of the vehicle (City in this case).
  - destination: Intended destination of the vehicle (Destination city).
  - distance_to_destination: Distance remaining to reach the destination.
  - battery_level: Current battery percentage.
- **Methods:** Includes getters and setters for vehicle attributes, allowing encapsulation and data integrity.
  - setLocation(): Sets the vehicle's current location.
  - setDestination(): Sets the destination.
  - setDistanceToDestination(): Sets distance to the desired destination.
  - setBatteryLevel(): Sets battery level for the vehicle.
  - getLocation(): Gives the current location of the vehicle.
  - getDestination(): Returns the destination for a given vehicle.
  - getDistanceToDestination(): Returns the distance to the destination.
  - getBatteryLevel(): Returns battery level for the given vehicle.

VehicleHashTable Class

- **Purpose:** Implements a hash table to manage vehicles efficiently, allowing for quick addition, removal, and search operations.

- **Attributes:**
  - hashArray: Array to store vehicle entries.
  - size: Size of the hash table.
- **Methods:**
  - add_vehicle(): Adds a vehicle to the hash table.
  - remove_vehicle(): Removes a vehicle using its ID.
  - get_vehicle(): Retrieves a vehicle by ID.
  - display_all_vehicles(): Displays all vehicles in the hash table.

HEAP Class

- **Purpose:** A max-heap implementation used to manage vehicle distances for sorting vehicles based on proximity to their destinations.
- **Attributes:**
  - array: Array to hold heap elements.
  - count: Current number of elements in the heap.
- **Methods:**
  - add(): Adds a new vehicle to the heap.
  - remove(): Removes the vehicle closest to its destination.
  - heap_sort(): Sorts vehicles by distance using heapsort.

DSAGraph Class

- **Purpose:** Represents the road network as a graph, where locations are vertices and roads are edges.
- **Attributes:**
  - vertices: A list of vertices representing cities (locations).
  - vertex_labels: An array storing the labels of the vertices.
  - directed: Boolean indicating if the graph is directed or undirected.
- **Methods:**
  - addVertex(): Adds a new vertex for a location.
  - addEdges(): Connects two vertices with an edge.
  - getNeighbors(): Retrieves adjacent vertices for a given location.
  - displayGraph(): Displays the entire graph structure, showing connections between locations.

Supporting Data Structure [1]

- SinglyLinkedList: Used to store the adjacency list for each vertex.
- Stack: Used in graph traversals (e.g., Depth-First Search).
- Queue: Used for graph traversals (e.g., Breadth-First Search).

1. The Custom Data Structures are used from practical sessions

1.2.2 Method Implementation

Each class encapsulates its functionality through well-defined methods, ensuring that the system adheres to the principles of Object-Oriented Programming (OOP).

- **Vehicle Class:**
  - Provides methods for getting and setting vehicle properties.
  - Includes a __str__ method for easy string representation of vehicle details.
- **VehicleHashTable Class:**
  - Contains methods for managing vehicle data efficiently, including collision resolution through linear probing.
- **HEAP Class:**
  - Implements necessary operations for a heap, including insertion and removal while maintaining heap properties.
- **DSAGraph Class:**
  - Supports graph operations, including vertex and edge management, as well as graph traversal methods (DFS and BFS).
  - The displayGraph() method provides a visual representation of the graph's structure.

1.2.3 System Architecture

The architecture of the AVMS is structured to allow interaction between various components, ensuring modularity and clarity. Below is a UML diagram illustrating the classes and their relationships:

1.2.4 Workflow of AVMS

The Autonomous Vehicle Management System (AVMS) is designed with a modular approach, where each component interacts with others to provide a seamless vehicle management experience. Here is a brief explanation of the interactions between the various components of the AVMS:

1. Vehicle Class and VehicleHashTable
   - **Interaction:** The Vehicle class represents each individual vehicle with attributes like ID, location, destination, distance to destination, and battery level.
   - The VehicleHashTable is used to efficiently manage these vehicles. When a vehicle is added, it is stored in the hash table using its ID as the key. The VehicleHashTable allows quick lookup, insertion, and deletion of vehicles.
   - **Purpose:** This interaction ensures that vehicles can be accessed, modified, or removed quickly using their unique identifiers.
2. Vehicle Class and HEAP

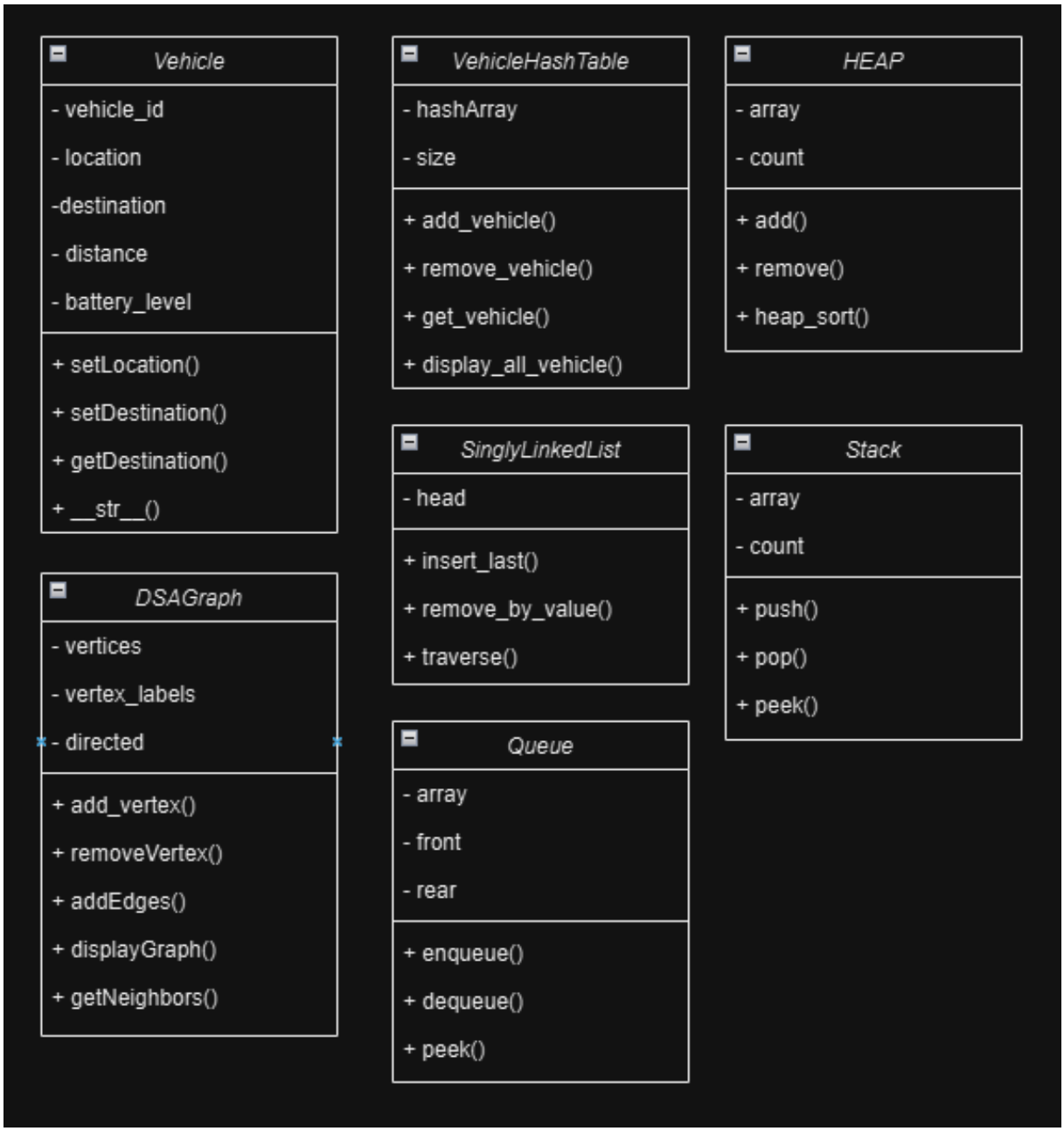

Figure 1: UML Class diagram

- **Interaction:** The HEAP class organizes vehicles based on their distance to their destination. When a vehicle is added, the distance value is used to maintain the heap's order.
- Vehicles closest to their destination are placed at the top of the heap, making it easy to quickly identify the nearest vehicle.
- **Purpose:** This ensures efficient sorting and prioritization of vehicles based on their distance to destinations.

3. DSAGraph Class and Vehicle Class

   - **Interaction:** The DSAGraph class represents the road network, where each location is a vertex, and connections between locations are edges.
   - When a vehicle is added to the system, its current location and destination are represented as vertices in the graph. An edge is created between these vertices to denote the connection or possible route.
   - **Purpose:** This interaction models the actual road network and helps visualize vehicle routes between different locations.

4. DSAGraph, SinglyLinkedList, Stack, and Queue

   - **Interaction:** The DSAGraph uses a SinglyLinkedList to store the adjacency list of each vertex, representing the neighbors of each location.
   - For graph traversal, the Stack is used in Depth-First Search (DFS), while the Queue is used in Breadth-First Search (BFS) to explore paths between locations.
   - **Purpose:** These data structures enable efficient graph operations, such as checking connectivity and finding paths between locations.

5. Integration Between Hash Table, Heap, and Graph

   - **Interaction:** The VehicleHashTable, HEAP, and DSAGraph components work together to provide comprehensive vehicle management:
     - The hash table quickly finds a vehicle based on its ID.
     - The heap prioritizes vehicles based on proximity to their destinations.
     - The graph maintains the road network structure, allowing pathfinding and navigation.
   - **Purpose:** This combined interaction enables the system to efficiently manage, locate, and track vehicles in real-time, while also understanding their positions within the broader road network.

6. User Interaction and Main Menu

   - **Interaction:** The main menu serves as the user interface, allowing users to interact with the AVMS components through a series of options. Users can add or remove vehicles, search for a vehicle, display all vehicles, visualize the road network, or sort vehicles.
   - The main menu orchestrates interactions between these components based on the user's inputs, ensuring that the right actions are triggered in response to user commands.

- **Purpose:** This interaction simplifies the process of managing the AVMS by providing a structured way to access each functionality through a user-friendly menu.

Flowchat of AVMS

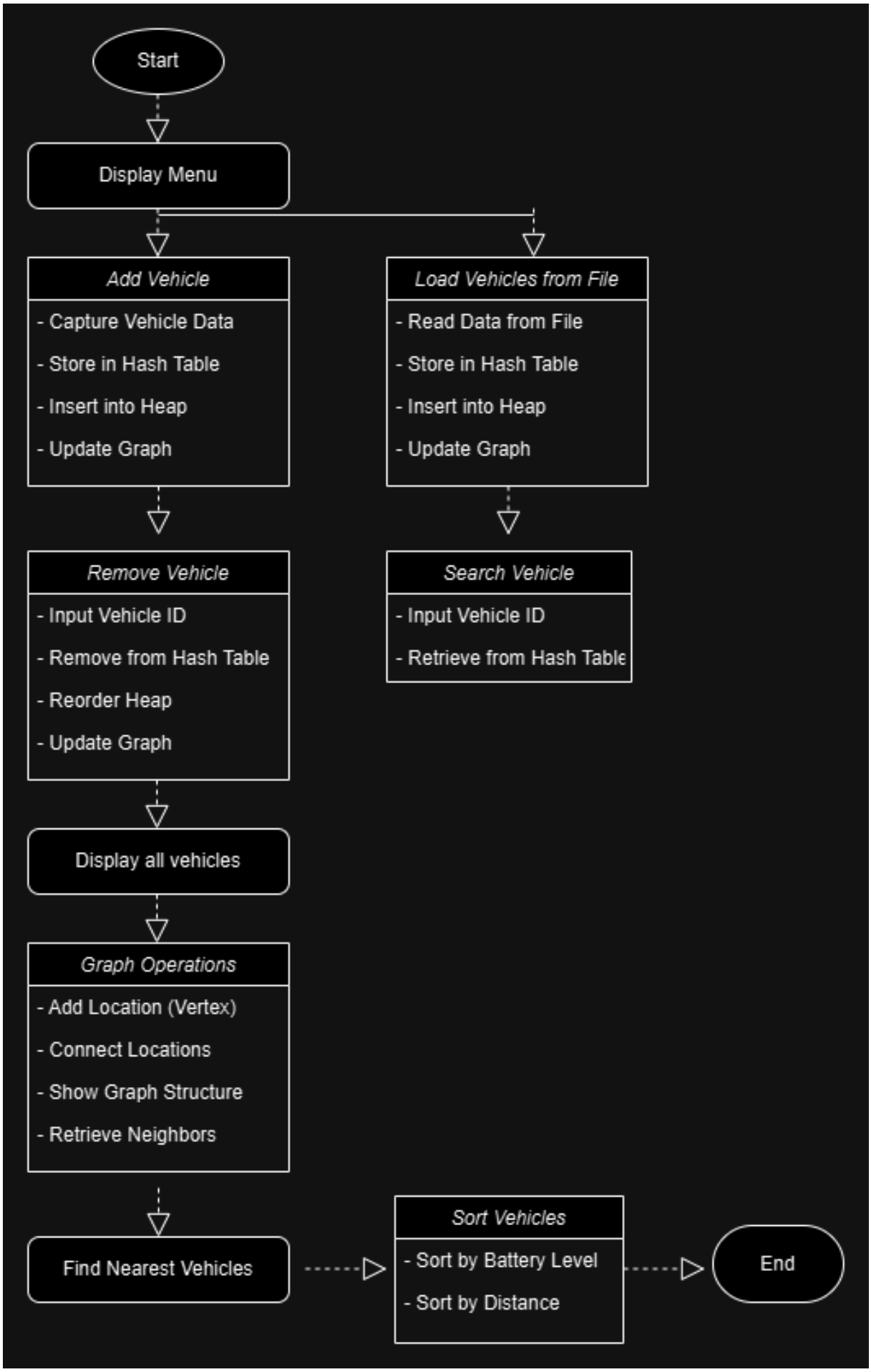

Figure 2: Flowchart of AVMS

1.3 Challenges Faced

1. **Custom Data Structure Implementation:** Developing custom data structures like SinglyLinkedList, Stack, and Queue required more time in development as I had to implement everything from scratch.

2. **Graph Management:** Ensuring that vertices and edges accurately reflected the relationships between locations in the graph was a challenge. I implemented checks in the addEdges method to ensure edges were added correctly for both directed and undirected graphs.

3. **Sorting and Searching Algorithms:** Implementing efficient sorting and searching algorithms while maintaining simplicity posed challenges. I optimized the hash table for quick lookups and ensured that the heap maintained its properties through careful implementation of the insert and remove methods.

4. **User Interface and Interactions:** Designing an intuitive user interface for the command-line application involved multiple iterations to ensure all functionalities were easily accessible and clear. I used input validation to enhance user experience.

5. **Code management:** As the project development progressed, the number of lines of code became so huge but I managed it using a modular approach implementing the Object-Oriented programming paradigm.

1.4 Efficiency Analysis

- Time Complexity
  - Hash table operations (adding, removing, and searching vehicles): $O(1)$ on average.
  - Heapsort for sorting vehicles: $O(n \log n)$.
  - Quicksort for sorting vehicles by battery level: $O(n \log n)$.
  - Graph traversal (DFS/BFS): $O(V + E)$, where V is the number of vertices and E is the number of edges.
- Space Complexity
  - The hash table requires $O(n)$ space to store the vehicles.
  - The heap requires $O(n)$ space for storing vehicle distances.
  - The graph's adjacency list structure requires $O(V+E)$ space for storing vertices and edges.

1.5 Potential Improvements

- **Dynamic Resizing for Hash Table:** Implement dynamic resizing for the hash table to handle scenarios where it becomes full or overly sparse, improving overall performance and memory efficiency.

- **Pathfinding Algorithms:** Introduce advanced pathfinding algorithms, such as A* or Dijkstra's, for more efficient route calculations in the graph. This will enhance the system's navigation capabilities.

- **Graphical User Interface (GUI):** Transitioning from a command-line interface to a graphical user interface would give better user experience, allowing users to interact with the system more intuitively.
- **Database Integration:** Consider using a database for persistent storage of vehicles and routes. This will enhance data management capabilities, allow for larger datasets, and provide a more scalable solution.
- **Real-time Updates:** Implementing real-time tracking and updates for vehicles, perhaps using web sockets or similar technologies, would enhance the functionality of the AVMS.

2 Testing of Autonomous Vehicle Management System (AVMS)

This section outlines the test cases for the Autonomous Vehicle Management System (AVMS), detailing the expected outcomes for each component of the system. The purpose of these test cases is to ensure that all functionalities are working as intended and to identify any potential issues or bugs in the system.

2.1 Testing Procedures

1. **Unit Testing:** Each class and its methods will be tested independently to verify their functionality. This includes testing constructors, getters/setters, and specific methods for adding, removing, or displaying data.

2. **Integration Testing:** The interaction between classes will be tested to ensure that components work together as expected. This includes testing the interaction between the Vehicle, VehicleHashTable, HEAP, and DSAGraph.

3. **User Acceptance Testing (UAT):** The final system will be tested through user scenarios to ensure that the user interface is intuitive and that all functionalities are accessible and functioning correctly.

Data for Testing To conduct a comprehensive test of the Autonomous Vehicle Management System (AVMS) I have used the following data within this file as input for our test cases. The file vehicles.txt contains vehicle data formatted as follows:

V001,CityA,CityB,12.5,80
V002,CityC,CityD,8.3,90
V003,CityB,CityE,15.2,70
V004,CityA,CityF,7.5,95
V005,CityD,CityG,9.1,60

The data represents vehicles with their respective current locations, destinations, distances to destinations, and battery levels.

2.1.1 Test Case 1: Load Vehicle Data from File

- **Action:** Load the vehicles.txt file into the system.
- **Expected Outcome:** All vehicles should be successfully added to the hash table, heap, and graph. Locations should be connected properly.
- **Verification:** Display all vehicles to ensure they are correctly stored.

2.1.2 Test Case 2: Display All Vehicles

- **Action:** Call the display_all_vehicles() method.
- **Expected Outcome:** The following vehicles data should be displayed:

Vehicle ID: V001, Location: CityA, Destination: CityB, Distance to Destination: 12.5, Battery Level: 80%
Vehicle ID: V002, Location: CityC, Destination: CityD, Distance to Destination: 8.3, Battery Level: 90%
Vehicle ID: V003, Location: CityB, Destination: CityE, Distance to Destination: 15.2, Battery Level: 70%
Vehicle ID: V004, Location: CityA, Destination: CityF, Distance to Destination: 7.5, Battery Level: 95%
Vehicle ID: V005, Location: CityD, Destination: CityG, Distance to Destination: 9.1, Battery Level: 60%

2.1.3 Test Case 3: Search for a vehicle

- **Action:** Search for the vehicle with ID V003.
- **Expected Outcome:** The system should display:

Vehicle found: Vehicle ID: V003, Location: CityB, Destination: CityE, Distance to Destination: 15.2, Battery Level: 70%

2.1.4 Test Case 4: Search for a Non-Existent Vehicle

- **Action:** Search for the vehicle with ID V010.
- **Expected Outcome:** The system should display:

Vehicle with ID 'V010' not found.

2.1.5 Test Case 5: Display Graph Structure

- **Action:** Display the graph structure after loading the vehicles.
- **Expected Outcome:** The graph should show all locations with their connections:

Location: CityA - Connected to: ['CityB', 'CityF']
Location: CityB - Connected to: ['CityA', 'CityE']
Location: CityC - Connected to: ['CityD']
Location: CityD - Connected to: ['CityC', 'CityG']
Location: CityE - Connected to: ['CityB']
Location: CityF - Connected to: ['CityA']
Location: CityG - Connected to: ['CityD']

2.1.6 Test Case 6: Vehicle Closest to Destination

- **Action:** Find the vehicle that is closest to its destination using the heap.
- **Expected Outcome:** The system should identify the vehicle V004 with a distance of 7.5 as the closes to its destination:

As per the given data.
The nearest vehicle to its destination is: Vehicle ID: V004, Location: CityA, Destination: CityF, Distance to Destination: 7.5, Battery Level: 95%

2.1.7 Test Case 7: Sort Vehicles by Battery Level

- **Action:** Sort the vehicles by their battery level using quicksort.
- **Expected Outcome:** The vehicles should be sorted in descending order of battery level:

Vehicle ID: V004, Battery Level: 95%
Vehicle ID: V002, Battery Level: 90%
Vehicle ID: V001, Battery Level: 80%
Vehicle ID: V003, Battery Level: 70%
Vehicle ID: V005, Battery Level: 60%

2.1.8 Test Case 8: Find Vehicle with Highest Battery Level

- **Action:** Identify the vehicle with the highest battery level.
- **Expected Outcome:** The system should identify vehicle V004 as having the highest battery level of 95%:

Vehicle with the highest battery level: Vehicle ID: V004, Battery Level: 95%

2.1.9 Test Case 9: Remove a Vehicles

- **Action:** Remove the vehicle with ID V002 from the system.
- **Expected Outcome:** The vehicle should be successfully removed. A subsequent search for V002 should indicate that the vehicle is not found:

Vehicle V002 removed from the system.
Vehicle with ID 'V002' not found.

2.1.10 Test Case 10: Validate Graph After Vehicle Removal

- **Action:** Display the graph structure after removing the vehicle V002.
- **Expected Outcome:** The connections involving CityC and CityD should still reflect their previous state since graph vertices are independent of vehicle removal:

Location: CityA - Connected to: ['CityB', 'CityF']
Location: CityB - Connected to: ['CityA', 'CityE']
Location: CityC - Connected to: ['CityD']
Location: CityD - Connected to: ['CityC', 'CityG']
Location: CityE - Connected to: ['CityB']
Location: CityF - Connected to: ['CityA']
Location: CityG - Connected to: ['CityD']

2.2 Conclusion

The successful development of the AVMS provided a comprehensive understanding of how to design and implement a complex system using modular principles, efficient data structures, and algorithmic strategies. It demonstrated how theoretical knowledge in data structures, graph theory, and sorting algorithms can be applied to build scalable and efficient solutions in real-world scenarios. These learnings are invaluable and can be applied to future software development projects, enhancing both design and coding skills.

3 Appendix

3.1 GitHub repository of AVMS

GitHub link