Data Structures and Algorithms - Assignment

Module Name - Data Structures and Algorithms
Module Code - COMP1002
Authored Name - Kulappu Arachchige Thisal Dilmeth
Curtin ID - 22383055
Date - 26.10.2024

Introduction

The function of the production code is to develop an Autonomous Vehicle Management System that will efficiently manage a fleet of autonomous vehicles and support real-time queries about vehicle status and location. The user can provide locations and create paths between those locations to create road networks. And user is able to deploy or remove vehicle from that road network as well as manipulate vehicle data.
The program will give recommendations to the user based on the battery level of a vehicle and the vehicles distance to its destination.

The program consists of 8 main operations

1. Adding Locations.
2. Adding Roads between locations.
3. Getting neighbouring locations of a given location.
4. Displaying the Road Network.
5. Checking if a road exists between two given locations.
6. Adding and Removing Vehicles.
7. Displaying the list of Vehicles.
8. Sorting the Vehicle list based on two criteria.

1. Adding Locations.

For this operation, the program utilises the **addVertex()** function located in the Graphs class file. This will add the location the user has given in to the vertex list. To avoid any complications the **addVertex()** function checks if the location the user has entered already exists in the vertex list. For this the **addVertex()** function uses the **hasVertex()** function. It will go through the vertex list and if a match is found it returns the boolean value 'true'.

2. Adding Roads Between Locations.

The **addEdge()** function, located in the Graphs class, is used for this operation. It takes in 3 parameters – **[location 1, location 2, distance]**. Before making the connection the program does error handling to avoid any unexpected behaviour. First it checks if both locations exists in the vertex list using **hasVertex()**. Then it makes sure that the entered distance is a non-negative number.

The **addEdge()** function first get the corresponding Graph Nodes to those given locations using the **getVertex()** function. It goes through the vertex list until a match is found which

then it will return that vertex. If a match is not found it will simply return a null value. Back in the Graphs.java file, the programs checks if both vertexes have non-null values. Then it
uses the **addEdge()** function located in the GraphNode class. In that function it creates an instance of an Edge class. The Edge class has the following attributes.

- Destination (GraphNode).
- Distance (Double).

Then this edge object is stored in list located in the GraphNode class. For the list we use a Linked List data structure for easy traversal and dynamic size.

3. Getting neighbouring locations of a given location.

The user is prompted to enter the name of the location and just like the previous functions it checks the existence of that location in the vertex list. Then the program uses the **displayAdjList()** function located in the Graphs class to display the adjacency list. It displays the names of the neighbouring locations and the distance to those locations.
It uses the **getAdjacent()** function with returns the edge list of a given Graph Node and use accessors such as **getLocation()** and **getDistance()**.

4. Displaying the Road Network.

This function uses the **displayList()** function in the Graph class to display the road network as a list. It will print each location with its adjacent locations and the distance to those locations. This also uses the **getAdjacent()** function and use accessors such as **getLocation()** and **getDistance()**.

5. Checking if a road exists between two given locations.

First it checks the existence of both locations and the use the **Breadth First Search (BFS)** traversal method, located in Graph.java, to find the path between the two given locations. If a path is found the **BFS()** function will return a true Boolean value. Otherwise, it will return false. The **BFS()** function uses stack and queue data structures. This traversal method goes layer by layer, starting from a given location and check all nodes in the current depth and moves to next depth level and checks all nodes in that level until the destination node is found.

6. Adding and Removing Vehicles

This function uses the HashTable data structure to store and manage each vehicle instance.

Like in all other functions, error handling is done here. First the program checks if the vehicle ID is already in the HashTable using the **find()** function located in the VehicleHashTable class. The it checks the locations and validates the user battery level. Then the program creates a vehicle with the user given attributes and stores that in the HashTable using the **put()** function in the VehicleHashTable class. It use the vehicle ID to create a hash index using the **Fowler-Noll-Vo (FNV-1)** hash function. In the put function if a collision occurs when entering the vehicle object, it uses double hashing to prevent it. Additionally if the load factor of the is high, meaning the HashTable is almost full, it uses the **resizeUp()** function to increase the size of the HashTable. The vehicle is also stored in an array of Objects as it will necessary later operations in the program.

When removing a vehicle it first checks if the vehicle HashTable is empty. And then check for the existence of the vehicle. Then it uses the **remove()** function in the VehicleHashTable class to remove the vehicle from the HashTable. And if the load factor of the HashTable is below a certain limit the program uses the **resizeDown()** function to decrease the size of the HashTable to save memory space.

7. Displaying the list of Vehicles.

First the program checks if the vehicleHashTable is empty. Then it uses the **display()** function in the HashTable class to display the HashTable as a list. It will first print the Vehicle Id followed by all of the attributes of the vehicle. The null entries in the HashTable are not displayed as it is not important and will only clutter the output.

8. Sorting the Vehicle list based on two criteria.

The vehicle can be sorted according to two criteria.

- Battery level.
- Distance to Destination.

Battery level

- The program uses the array of Objects stated earlier for this operation. The program will use the **sortBatteryLevel()** function in the Vehicle class to sort the Object array in descending order using a QuickSort algorithm. It will also display the sorted array using a simple for loop.
- Then it uses the **find_vehicle_with_highest_battery()** function in the Vehicle class to get the vehicle with the highest battery life.

Distance to Destination

- First we start with the error handling. Then the program uses the **sortDistance()** function in the Vehicle class. In the **sortDistance()** function we take all the data from the VehicleHashTable and store them in a heap using a for loop and the **add()** function in the Heap class. And then we call the **heapSortThis()** function which is a wrapper method for the **heapsort()** function.

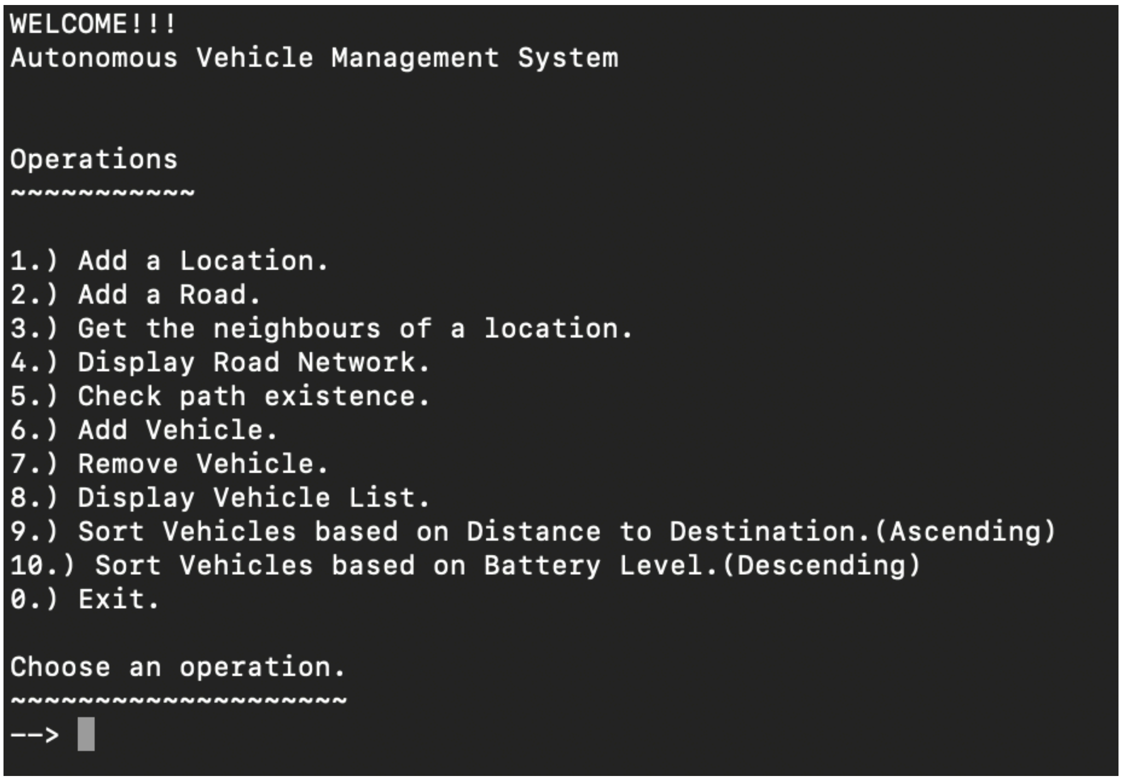
- In the **heapsort()** function, it first heapifies the array using the **heapify()** function. Then in the **sortDistance()** function in the Vehicle class after sorting the heap, using a simple for-loop the program prints the array. In ascending order.

- Then using the **find_nearest_vehicle()** function to get the vehicle with the shortest distance to their destination.

How to run the code

1. Open the terminal and compile all the java file.
2. Run the Menu file by entering **java Menu** in the terminal.
3. Follow the prompts given by the program.

Sample Output



```
WELCOME!!!
Autonomous Vehicle Management System


Operations
~~~~~~~~~~~

1.) Add a Location.
2.) Add a Road.
3.) Get the neighbours of a location.
4.) Display Road Network.
5.) Check path existence.
6.) Add Vehicle.
7.) Remove Vehicle.
8.) Display Vehicle List.
9.) Sort Vehicles based on Distance to Destination.(Ascending)
10.) Sort Vehicles based on Battery Level.(Descending)
0.) Exit.

Choose an operation.
~~~~~~~~~~~~~~~~~~~~
-->
```

Vehicle Class

The vehicle class is used to create vehicle object by using the data given by the user. It provides various methods to set and retrieve these attributes, as well as functions for vehicle management and sorting

Attributes:

- vehicleID (String)
- currentLocation (String)
- distance (double)
- destination (String)
- batteryLevel (double)
- heap (Heap)

Functions:

- **Getters and Setters for each of the attributes.**
- **setDistance() -** Uses Depth First Search (DFS) to find a path between two locations and find the total distance between them. It takes in a Graph object and two locations as parameters.
- **sortDistance() -** This takes in a VehicleHashTable object as a parameter. It take all of the data in the HashTable and inserts them into a heap one by one. Then it calls the heapSort() function and sorts the heap. The using a for-loop it displays the heap. This returns a Heap object.
- **find_nearest_vehicle() -** This takes in a Heap object as a parameter and displays the value of the element at the top of the heap.
- **sortBatteryLevel() -** This takes in an Object array as a parameter. Then it counts the number of non-null elements in the array assigns that to an int variable name ‘count’. After it creates a new Object array with the length of ‘count’. Then this array is filled by all of the non-null elements in the passed in(parameter) array. Then it uses QuickSort to sort that array and displays it using a simple for loop. This returns an object array.
- **find_vehicle_with_highest_battery() -** This takes in an Object array and displays the element at the top of the array (index 0).
- **removeVehicle() -** This takes an array and a string key and removes that key from the array and returns that array.

VehicleHashTable Class

The VehicleHashTable class manages vehicle entries using a hash table. It supports basic hash table operations such as insertion, retrieval, removal, and resizing when the load factor crosses specific thresholds.

Attributes:

- hashArray (DSAHashEntry[])

Functions:

- **Getters and Setters:**
  - **getHashTable() -** Returns the HashTable array.
  - **getArraySize() -** Returns the current size of the hash array.
  - **getIndexValue(int i) -** Gets the value at a specified index.
  - **getIndexKey(int i) -** Gets the key at a specified index.
- **findNextPrime() -** This takes in an integer parameter. Calculates the next prime number of the passed in integer. The prime number is what we used to declare the size of the hash array. The HashTable's array size is always a prime, which reduces collisions.
- **get() -** Gets the value of a given key. Uses double hashing to handle collisions. It loops using a while loop until it either finds the key or has gone through the whole array.
- **getEntry() -** This works a same as the get() function but this return the whole DSAHashEntry of a given key.
- **find() -** Checks if a given key exists in the HashTable, using double hashing to handle collisions.
- **put() -** Inserts a new HashEntry with a given key and value or updates the value if the key already exists in the HashTable. Handles collisions with double hashing and resizes the HashTable when the load factor is above 0.70.
- **remove() -** Removes the entry of the given key, marking it as "formerly used" in the HashTable by changing the state to -1. The table resizes down if the load factor is below 0.40.

- **getNumOfEntries() -** Returns the number of non-null entries in the HashTable.
- **getLoadFactor() -** Calculates and returns the load factor (ratio of non-null entries to the total table size).
- **hashFunction() -** Calculates the hash index for a given key using the **Fowler-Noll-Vo (FNV-1)** hashing algorithm.
- **stepHash() -** Calculates the secondary hash value for a key, used for double hashing to handle collisions.
- **resizeUp() -** Increases the HashTable size to the next prime number greater than twice the current size. Re-inserts and hashes all current entries into the new array.
- **resizeDown() -** Reduces the HashTable size to the next prime number smaller than half the current size. Re-inserts and hashes all current entries into the new array.
- **display() -** Displays the key and associated vehicle data (destination, current location, distance, and battery level) for each entry in the hash table.

DSAHashEntry Inner Class

Attributes:

- key (String)
- value (Object)
- state (int) —> 0 - [empty], 1 - [in use], -1 - [previously used]

Functions:

- **Getters and Setters for each of the attributes.**
- **Two constructors -** One sets the key and value to null and the state to 0. The other constructor takes in the key and the value as parameters and sets the state to 1.

Edge Class

The Edge class stores a destination and the distance to that destination. This allows us to store edge data when an edge is created between two graph nodes (locations).

Attributes:

- destination (GraphNode)
- distance (double)

Functions:

- **Getters and Setters for each of the attributes.**
- **Constructor -** Takes in the destination and the distance as parameters and assigns that to the attributes.

QuickSort Class

This takes in an array and uses a quick sort algorithm to sort the data and returns the array.

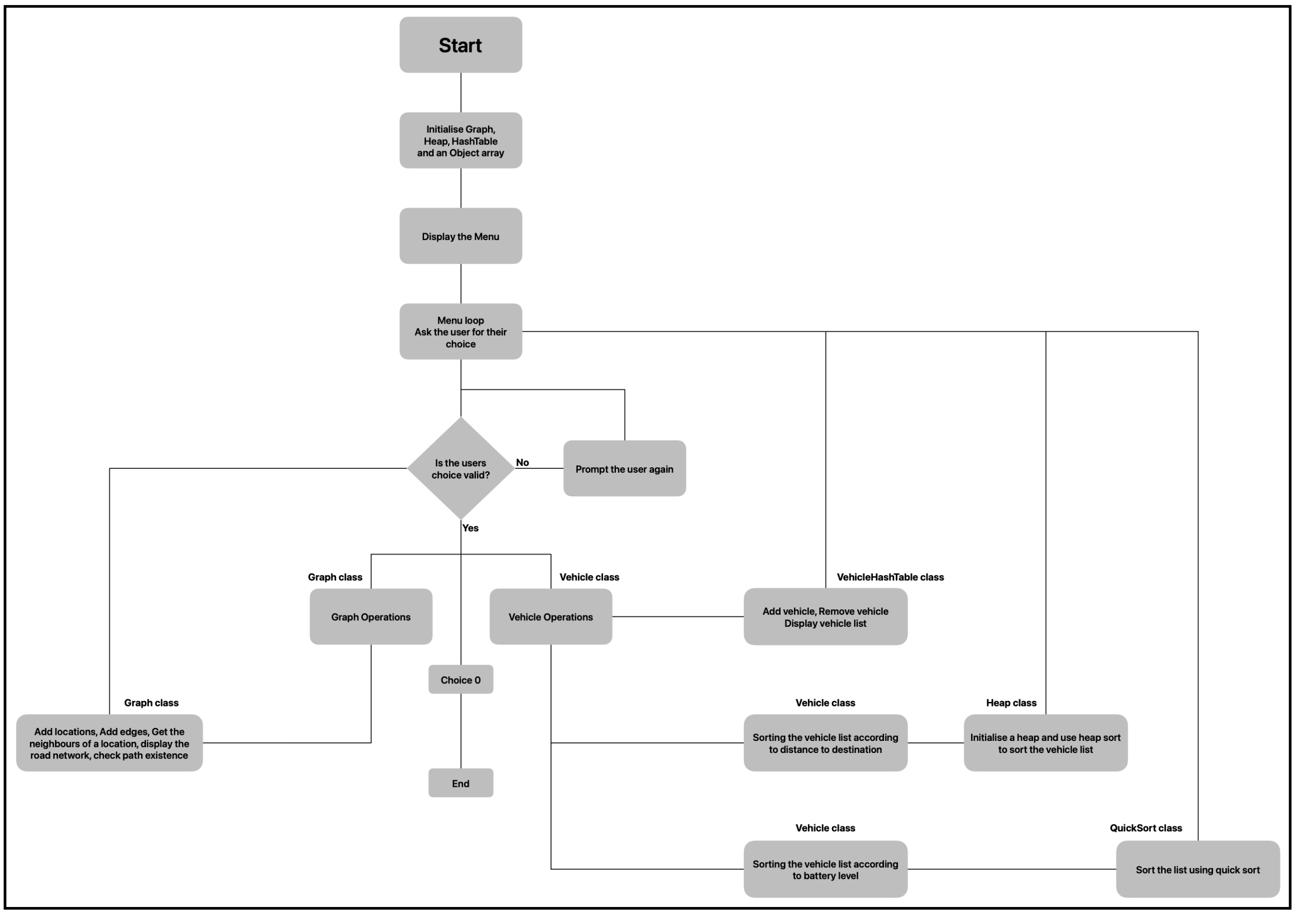
Functions:

- **quickSort() -** This takes in 3 parameters - [array to be sorted, left index, right index]. This is a recursive quick sort function that divides the array into two sub arrays and applies quickSort() to each sub array. First it calculates the pivot index (usually the middle index). Then it calls the doPartitioning() function to rearrange the elements based on the pivot index and calculates and returns the new pivot index.

- **doPartitioning() -** This takes in 4 parameters - [array to be sorted, left index, right index, pivot index]. This rearranges elements so that the elements with the higher values than the pivot value are stored on the left and the lower values are store on the right. First it swaps the pivot value with the right index value. Then declares current index equal to left index to keep track of the boundaries of the left sub array. Then it loops through the array and if element i has a larger value than the pivot val, that element is swapped with the current index element and current idx gets incremented. The the pivot value is placed at the correct sorted position and the new pivot index is initialised to current index and returned.

All other classes are generic data structure classes. The other data structures are Graphs, Linked Lists, Queues, Stacks and Heaps.

Flowchart of the systems architecture



Start
Initialise Graph, Heap, HashTable and an Object array
Display the Menu
Menu loop Ask the user for their choice
Is the users choice valid?
No
Prompt the user again
Yes
Graph class
Graph Operations
Vehicle class
Vehicle Operations
VehicleHashTable class
Add vehicle, Remove vehicle Display vehicle list
Choice 0
End
Graph class
Add locations, Add edges, Get the neighbours of a location, display the road network, check path existence
Vehicle class
Sorting the vehicle list according to distance to destination
Heap class
Initialise a heap and use heap sort to sort the vehicle list
Vehicle class
Sorting the vehicle list according to battery level
QuickSort class
Sort the list using quick sort

Traceability Matrix

Operation	Menu Choice	Methods involved	Expected outcome	Status
Initialising	-	-	Initialises Graph, Heap, VehicleHashTable and an Object array.	**P**

1.
1.1 Add location.
	1	process()
		switch() case 1:
		graph.addVertex()
			Adds a new vertex (location) to the graph (road network).	**P**

1.2 Add existing location.	1	process()
		switch() case 1:
		graph.hasVertex()
			Should prompt you again for another valid location.	**P**

2.
2.1 Add a road between locations.
	2	process()
		switch() case 2:
		graph.addEdge()
			Adds an edge between two locations. Creates a new Edge object and adds it to the adjacency list of both vertexes (locations).	**P**

2.2 Add a road between non existent locations.	2	process()
		switch() case 2:
		graph.hasVertex()
			Should prompt you again for another valid location.	**P**

3.
3.1 Get neighbours of a location.
	3	process()
		switch() case 3:
		graph.displayAdjList()
			Displays the neighbouring locations of a specified location in the graph.	**P**

3.2 Get neighbours of a non existent location.	3	process()
		switch() case 3:
		graph.hasVertex()
			Should prompt you again for another valid location.	**P**

4. Display road network.	4	process()
		switch() case 4:
		graph.displayList()
			Displays the whole graph with all the locations and their neighbouring locations with the distance to them.	**P**

5.
5.1 Check path existence between locations.
	5	process()
		switch() case 5:
		graph.BFS()
			Checks if a path exists between two specified locations using a breadth-first search.	**P**

5.2 Check path existence between non existent locations.	5	process()
		switch() case 5:
		graph.hasVertex()
			Should prompt you again for another valid location.	**P**

Operation	Menu Choice	Methods involved	Expected outcome	Status

6.
6.1 Add Vehicle.
	6	process()
		switch() case 6:
		vehicleHashTable.put()
		vehicle.setID()
		vehicle.setLocation()
		vehicle.setDestination()
		vehicle.setBatteryLevel()
		vehicle.setDistance()
			Adds a new vehicle to the HashTable with properties such as ID, current location, destination, and battery level.	**P**

6.2 Add Vehicle with same ID.	6	process()
		switch() case6:
		vehicleHashTable.find()
			Should prompt you again for a valid vehicle ID.	**P**

6.3 Non existent location and destination.	6	process()
		switch() case 6:
		graph.hasVertex()
			Should prompt you again for another valid location.	**P**

6.4 Invalid battery level	6	process() switch() case 6: Uses a try-catch and an if statement	Should prompt you again for a valid battery level.	**P**

7.
7.1 Remove Vehicle.
	7	process()
		switch() case 7:
		vehicleHashTable.remove()
			Removes a vehicle from the HashTable based on the given vehicle ID.	**P**

7.2 Remove non existent Vehicle.	7	process()
		switch() case 7:
		vehicleHashTable.find()
			Should prompt you again for a valid vehicle ID.	**P**

7.3 If the HashTable is empty	7	process()
		switch() case 7:
		vehicleHashTable.getNumOfEntries()
			Should print out that the HashTable is empty.	**P**

8.
8.1 Display Vehicle List.
	8	process()
		switch() case 8:
		vehicleHashTable.display()
			Displays the HashTable of vehicles. This does not show any of the null entries.	**P**

8.2 If the HashTable is empty	8	process()
		switch() case 8:
		vehicleHashTable.getNumOfEntries()
			Should print out that the HashTable is empty.	**P**

9.
9.1 Sort vehicles by distance to destination.
	9	process()
		switch() case 9:
		vehicle.sortDistnace()
		vehicle.find_nearest_vehicle()
			Sorts the vehicles in ascending based on their distance to destination.	**P**

9.2 If the HashTable is empty	9	process()
		switch() case 9:
		vehicleHashTable.getNumOfEntries()
			Should print out that the HashTable is empty.	**P**

Operation	Menu Choice	Methods involved	Expected outcome	Status

**10.
10.1 Sort vehicles by battery level.**
	10	process()
		switch() case 10:
		vehicle.sortBatteryLevel()
		vehicle.find_vehicle_
		with_highest_battey()
			Sorts the vehicles in descending based on their battery level.	**P**

10.2 If the HashTable is empty	10	process()
		switch() case 10:
		vehicleHashTable.getNumOfE
		ntries()
			Should print out that the HashTable is empty.	**P**

Exit program	0	process() switch() case 0:	Exits the program.	**P**

Test cases

Test case ID	Description	Inputs	Expected Outputs	Status
TC01	Initialize locations and add vertices	locations[]	Locations P, Q, R, S, T, U, V, W, X, Y, Z, AA, BB, CC, DD are added to the graph without errors.	**P**
TC02	Connect locations with roads (edges)	addEdge() calls	Roads are added between specified pairs of locations with distances.	**P**
TC03	Display road network as adjacency list	displayList()	Adjacency list displays all vertices and their neighbors, showing the connected road network.	**P**
TC04	Display road network as adjacency matrix	displayMatrix()	Adjacency matrix displays the network in a matrix format, showing distances between locations and 0 for unconnected locations.	**P**
TC05	Check if path exists from “P” to “T”	BFS("P", "T")	true – a path exists between “P” and “T”.	**P**

Test case ID	Description	Inputs	Expected Outputs	Status
TC06	Check if path exists from “AA” to “DD”	BFS("AA", "DD")	true – a path exists between “AA” and “DD”.	**P**
TC07	Check if path exists from “U” to “Z”	BFS("U", "Z")	false – no path exists between “U” and “Z”.	**P**
TC08	Add vehicle “RB20”	vehicleTable.put()	Vehicle “RB20” with location “T” and destination “P” is added to the VehicleHashTable with distance and battery level set correctly.	**P**
TC09	Add vehicle “RB19”	vehicleTable.put()	Vehicle “RB19” with location “U” and destination “X” is added to the VehicleHashTable.	**P**
TC10	Add vehicle “RB18”	vehicleTable.put()	Vehicle “RB18” with location “R” and destination “DD” is added to the VehicleHashTable.	**P**
TC11	Add vehicle “RB17”	vehicleTable.put()	Vehicle “RB17” with location “BB” and destination “W” is added to the VehicleHashTable.	**P**
TC12	Add vehicle “RB16”	vehicleTable.put()	Vehicle “RB16” with location “R” and destination “Z” is added to the VehicleHashTable.	**P**
TC13	Add vehicle “RB15”	vehicleTable.put()	Vehicle “RB15” with location “Q” and destination “V” is added to the VehicleHashTable.	**P**
TC14	Display Vehicle Table	vehicleTable.display()	Vehicle table displays all added vehicles with details including ID, location, destination, battery level, and distance.	**P**
TC15	Remove vehicle “RB17”	vehicleTable.remove("RB17")	Vehicle “RB17” is removed from VehicleHashTable, and no longer appears in the display.	**P**
TC16	Display Vehicle Table after removing vehicle “RB17”	vehicleTable.display()	Updated vehicle table shows remaining vehicles after removing “RB17”.	**P**

Test case ID	Description	Inputs	Expected Outputs	Status
TC17	Sort vehicles by distance to destination	sortDistance()	Vehicles sorted in ascending order of distance to their destination. find_nearest_vehicle() correctly identifies the vehicle closest to its destination.	**P**
TC18	Sort vehicles by battery level	sortBatteryLevel()	Vehicles sorted in descending order by battery level. find_vehicle_with_highest_battery() correctly identifies the vehicle with the highest battery level.	**P**

Conclusion

Overall this project was a good way to test my knowledge on data structure and algorithm. It also gave me a more deep understanding of the inner workings of the algorithms and data structures. I did have some difficulties when linking all these data structure to each other. Especially when sorting the vehicle list.

Efficiency analysis

1. **Graph Class**
    - **Path finding -** Time complexity —> O(V + E).
    - **Distance -** Time complexity —> O(V + E).
2. **HashTable Class**
    - Time complexity - Average of O(1).
3. **Sorting Algorithms**
    - **Quick Sort**
        - Time complexity —> Best/Avg case O(n log n).
        - Time complexity —> Worst case O(n^2).
    - **Heap Sort**
        - Time complexity —> O(n log n).

Potential improvements

1. **Better algorithm to find a path and the distance.**
    - We can use the Dijkstra’s algorithm for finding the shortest path to a destination and get accurate distance calculations.