

Final assignment report

COMP5008 - Semester 2, 2024



October 27, 2024

bUI tHAO PHUONG

21875768

# INTRODUCTION

## Problem statement:

In this project, we have to build an Autonomous Vehicle Management System. We have the Road Network, Vehicle Hash Table, and the Vehicles.

In the Road Network, we should be able to add the locations and the roads, retrieve the neighbors, display the graph, and check for the path existence.

Vehicle Hash Table is where we manage all the vehicles. We perform actions such as inserting the vehicles, deleting the vehicles, searching and displaying the vehicles.

The Vehicle class stores the vehicles’ information like the ID, locations, destination, distance, and battery level. It can also have actions like set and get location, set and get distance, set and get battery level to retrieve necessary information about the vehicles.

## Purpose of the assignment:

The purpose of the assignment is to implement the methods learnt during the lectures and practical sessions. The implemented methods are Hash Table, Graph, Heap Sort, Quick Sort, Linked List, Queue and Stack (in Graph). This Assignment is the synthetic of the weekly practical exercises in those topics, which helps me revise and practice more on those topics.

This report discusses how the classes and methods are developed, including their implementations, test cases, and efficiency analysis.

# Class and Method Implementations:

## Graph Class:

### Purpose

This class represents the Road Network. It is used to store all the vertices (locations) and edges (road), and all the actions related on that road network.

### Methods

The Graph class is implemented based on weighted Graph methods.

#### b.1. addLocation(self, location):

This method is used to add location into the road network. It uses the Graph’s addVertex() method in DSAGraph() to add new location. Each location is represented by a vertex on the graph. Graph uses Linked List to store the vertices. Each time a new vertex is added, it will insertLast() to that Linked List.

#### b.2. addRoad(self, location1, location2, distance):

This method is used to add road between two locations and also add distance between those two. The road is represented by an edge on the graph. We import method addEdge(self, label1, label2, weight) from DSAGraph(). This edge connects two vertices (locations) to produce a path. The weight on each edge stands for the distance between two locations.

#### b.3. retrieve\_neighbors(self, location):

This method retrieves the locations which are adjacent to the input location. We use getAdjacent() to find the adjacent vertices on the graph, which is the neighbors of one location.

#### b.4. displayGraphL(self):

This method is used to display the graph in a list form. It uses the method displayAsList() in DSAGraph() to display all locations and their neighbors.

#### b.5. displayGraphM(self):

displayGraphM(self) method displays all locations and their neighbors in a matrix form by using displayAsMatrix() in DSAGraph().

#### b.6. is\_path(self, source, destination):

is\_path() checks whether there is a path between source and destination. It returns True if there is a path between the two locations, otherwise it returns False. We use Breath First Search to traverse from the source to find the destination. If it completes traversing and can find the destionation (source == destination), the method returns True. If it completes traversing but cannot find the destination, it means there is no path between source and destination. Therefore, the method returns False.

We have also checked the case in which the source or the destination do not exist. In that case, we will print a message “Source/Destination not found!”.

If the user enters a source which is the same as the destination, the method prints “That’s the same place!”.

#### b.7. find\_all\_path(self, source, destination):

In this method we store all paths the vehicle can possibly go to if it wants to get from source to destination.

We create a new linked list called PathLinkedList() to store all the paths. We call the method dfs\_find\_path() to traverse to all edges (paths) possible to get to the destination.

#### b.8. dfs\_find\_path(self, current, destination, visited, path, distance):