ASSIGNMENT 2

SUBJECT : ARTIFICIAL INTELLIGENCE
SUBJECT CODE : CSC3206
LECTURER : DR. RICHARD WONG TECK KEN
DEADLINE : 20th JULY 2023

NO.	NAME	STUDENT ID
1.	Khoo Siong Hoe	21072319
2.	Lee Boon Bing	18109082
3.	Muhammad Idraki Radzi Bin Mohd Radzi	20025821
4.	Tan Javier	19017219

TABLE OF CONTENTS

1.0 INTRODUCTION

Based on the given requirements as well as the problem description of the assignment, the proposed application was developed using the Python programming language and its corresponding libraries such as “Matplotlib” and “NumPy”. The proposed application is mainly operated by the user in the Command-Line Interface (CLI) and its output is mostly seen in the CLI as well with the exception of certain advanced features. Besides that, the proposed application is a maze solver that uses the informed A* Search algorithm in order to find the paths needed to effectively visit certain locations within the maze and will only stop after clearing all the conditions stated within the problem description of the assignment. There are four main files that the application uses, and these files handle the menu, the search algorithm, the logic of the maze, and the maze visualization feature. The main purpose of this report is to document the various features of the proposed application as well as its capability to cater to different maze configurations such as flexible coordinates and changeable limits. Furthermore, the documentation will also describe any and all concepts related to data structures and algorithms used within the proposed application. Additionally, the report will contain the results of the given assignment problem and will contain an in-depth evaluation in terms of its overall performance and outcome.

2.0 PROBLEM DESCRIPTION

This section of the report will be focused on the in-depth analysis of the problem stated within the given assignment and will also contain vital assumptions as well as restrictions that must be considered when proposing suitable search algorithms. The main objective that the proposed search algorithms must accomplish is to find the shortest path that is able to visit all the specific nodes, each of them containing their own individual values, and to store the values however, the stored values must not exceed a predefined numerical limit and if the predefined numerical limit has been reached or is close to reaching, the algorithm must visit another set of specific nodes in order to reset the stored value back to zero.

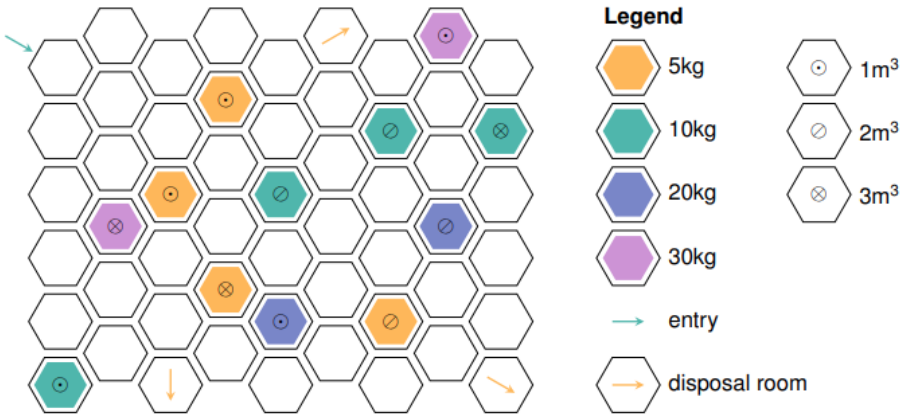

Figure 1 Maze Arrangement & Layout

In Figure 1, the assignment's maze layout and the required nodes to visit are shown. From the figure the movement and shape of the maze is understood where the shape is in the form of hexagonal nodes arranged in an arrangement of six rows and nine columns. The only movements or actions allowed within the hexagonal nodes are upwards, downwards, and four diagonal movements composed of north-east, north-west, south-east and south-west. Next, the maze also clearly labeled as well as identified each of the nodes with the necessary information such as the starting node, the 'rubbish room' nodes, the 'disposal room' nodes and the regular nodes. Each of the 'rubbish room' nodes contain values corresponding to the weight as well as the size of the 'rubbish', and there are three 'disposal room' nodes laid out at the edges of the maze.

The assignment has also stated certain prerequisites and assumptions that must be considered when selecting and evaluating the proposed search algorithms. The prerequisites are that the limit of the stored value must not exceed forty for an attribute known as "kilogram" and also five for another attribute labeled as "cubic meters". One assumption that is understandable from the assignment is that the layout of the nodes as well as their corresponding roles and values are known beforehand, meaning the proposed application is able to capitalize on the number of remaining 'rubbish room' nodes and continuously execute itself until all the nodes are visited. The next assumption that can be deduced from the assignment is that the final destination must be one of the 'disposal room' nodes due to the requirement of clearing all the 'rubbish' within the maze, hence the aforementioned attributes, 'kilogram' and 'cubic meters' requires to be zero in order to terminate the algorithm.

3.0 IMPLEMENTATION DESCRIPTION

3.1 Main Menu Implementation

The main menu is generated and handled with the ‘prompt_menu_choice` method. To display the menu, the program iterates over the enumerated list, presenting each option along with its corresponding number. Using an enumerated list along with string formatting to generate the menu options allows for simple editing or addition of options, without having to write or edit individual print statements. The main menu is presented using numbers to represent the option and to accept user input, making it simpler for users to identify and enter the desired option. The program incorporates general error handling for the menu option input. When the user enters a value to select a menu option, the program validates the input to ensure it is an integer within the acceptable range. If the user enters a non-integer or an integer outside the range of the menu options, an error message is displayed which prompts the user to enter a valid integer. The error handling system ensures that the program will keep asking for input until a valid choice is provided. This approach prevents any unexpected or incorrect inputs from causing disruptions in the menu flow.

The first three menu options, namely "Enter bin location", "Enter rubbish locations," and "Enter disposal locations," allow users to input coordinates to add elements within the maze. Each option requires the user to enter the corresponding coordinates and validates the input. Selecting the "Enter bin location" option allows the user to define the coordinate of the bin in the maze, with error handling ensuring the input is a valid coordinate. Similarly, when "Enter rubbish locations" is chosen, users can add multiple rubbish locations with their respective weights and sizes. The program validates each input, updates the maze, and handles scenarios where the entered location is already present in the maze. The "Enter disposal locations" option allows users to input rubbish disposal locations, checking for duplicates or conflicts with existing rubbish or bin locations. Option four, “Read locations from text file”, allows users to enter a preset maze from a text file. The user points towards a text file, and if the file exists and is properly formatted, the program reads the locations from the file and updates the maze layout and variables. The option “Display maze visualization” generates a visualization of the maze. The visualization displays the bin location, disposal locations, and rubbish locations in the maze. The program retrieves the relevant information from the stored variables and uses the Matplotlib library to create the visual representation. The sixth menu option is "Start maze solver." Upon selecting this option, the program prompts the user to enter the weight and size capacities for the bin. Error handling is implemented to validate user input, as well as to ensure

that the maze is solvable, by checking if the bin capacities are greater than or equal to the rubbish with the highest capacities. If the bin location, rubbish location or disposal location has not been set, the user is prompted to input the coordinates and properties for the variables respectively. Once the necessary data is obtained, the program passes the maze data, bin capacity, and locations to the `RubbishCollector` algorithm. The algorithm runs and performs rubbish collection based on the provided data, optimizing the collection process and printing the path results. Option seven is "Reset maze locations". This option resets all the maze locations, including rubbish, bin, and disposal locations. The program clears the stored variables related to the locations, reverting the maze to its initial state. The final option on the menu is "Quit application". This option terminates the program and exits the main menu loop. A closing message is displayed before terminating.

3.2 A* Search Implementation

This part of the paper provides a thorough examination of an A* Search algorithm-based garbage retrieval procedure. Firstly, it will start by going through the 'RubbishCollector' class's constructor, the '__init__' method, initializes the variables required for the garbage retrieval procedure. The technique establishes the bins' weight and size limits and the locations of the garbage collection and disposal facilities. The maximum values of the x and y coordinates of the disposal and trash sites are considered when determining the size of the maze using the 'determine_maze_size' method that is implemented in the code. The method returns the size of the maze after adding buffer space to allow for the bin's movement. Based on the supplied size, the 'create_empty_maze' method constructs an empty maze that is represented as a 2D list of zeros. Now for the next method that will be talked about it is the 'get_rubbish' method which incorporates the A* Search algorithm to find the shortest path of the rubbish's location starting from the bin while avoiding any disposal location or previously visited rubbish locations. The paths retrieved are sorted to their respective length and the rubbish location. In this method, the bin's state is updated according to the rubbish being retrieved in the final path and then the bin's current weight and size will be adjusted correspondingly. The 'get_disposal' method uses an A* search to determine the closest disposal site from where the bin is currently located. The current weight and size are reset to zero, and the bin's position is updated. These procedures guarantee effective garbage collection and disposal, and the code publishes the current weight and size of the bin as well as movement instructions, assisting in process oversight.

The ‘avoid_location’ method locates garbage that cannot be collected because it is too large or heavy for the bin. When collecting trash, this list of sites is used to prevent gathering trash that cannot be accommodated. The movement directions necessary to go to each place along the pathways are clearly displayed using the ‘print_directions’ method. It shows the movement order from the initial location to each succeeding position, making it easier to comprehend how the bin moves while being retrieved and disposed of. The garbage retrieval and disposal procedure are carried out via the ‘run_algorithm’ method until all garbage has been collected. It determines the capacity of the bin and then calls the appropriate procedures, as necessary. The sum of the lengths of each path in the ‘final_path’ list, minus one (except for the starting point of each path), is used by the procedure to determine the overall path cost. The total number of nodes expanded during the A* search and the overall frequency of disposal location visits are reported. The ‘final_path’ list is iterated over by the code to output the final path, which offers a thorough breakdown of the actions made during the garbage retrieval process.

The algorithm obviously had some modifications and improvements done to the original A* Search. One of the changes made was that the tie breaking for the f-values is done based on the g-values. In the Node class there is the ‘__it__’ method which will determine the tie-breaking through their specific g-values if their f-values are the same. With the prioritization of the g-values the algorithm may be able to discover a shorter path earlier as it will focus on expanding nodes that are closer to the starting position first. After that the algorithm does not use any sort of Heap operations. Rather than using the original way of storing the f-values this code will start searching for the node that has the lowest f-value with the usage of the built in ‘min’ function. By approaching the problem in this way, the node retrieval method for the node with the lowest f-value becomes more efficient and a heap would not need to be maintained. Next, the addition as well as the removal of the respective nodes from the ‘closed_list’ was made efficient. The membership checking is allowed to be made efficient ‘if child in closed_list’ because the ‘closed_list’ is now implemented as a set. The addition of the nodes to the ‘closed_list’ and the checking of the membership in the ‘closed_list’ is now done constantly allowing for a speedup in terms of an execution in comparison to a list-based execution. Lastly the overall memory usage of the algorithm was reduced because of the implementation of set to the ‘closed_list’ no duplicate entries will be entered and therefore the memory consumption is reduced. The f-values are stored in the ‘open_list’ as ‘open_list’ is implemented into a dictionary and they would not be duplicated in each node reducing memory usage.

The heuristic function, or specifically the Manhattan distance will be talked about now. The Manhattan distance is the heuristic function employed in this implementation. The sum of the absolute differences in the x and y coordinates between two points is used to determine the Manhattan distance, sometimes referred to as the L1 norm. The ‘distance’ function in the code determines the biggest absolute difference between two places' X and Y coordinates to determine the Manhattan distance between them. This heuristic can be used for A* Search since it is acceptable, which means it never overestimates the actual cost to achieve the goal. Other than that, this code uses offset rows and columns as the foundation for its movement logic. A grid is used to depict the maze, and each cell can be thought of as a position. The function examines the current row's parity to find a node's neighbors. The ‘even_row_offsets’ list is used by the code to describe the potential offsets for adjacent spots if the current row is even. Similar to this, the list of ‘odd_row_offsets’ is used when the current row is odd. Depending on the row parity, these offset values reflect the different movements of moving up, down, left, right, and diagonally. The algorithm ensures that movements are in line with the grid layout and adhere to the criteria by considering the offset rows and columns. Then lastly the part of the code that will be talked about is how the nodes are expanded and the return paths are generated. The code offers two operations for counting the number of nodes expanded throughout the search and returning paths. The ‘return_path’ method builds the path from the start node to the goal node by tracing back through the parent pointers starting with the current node (the goal node). To get the right order, the places are added to a list and then the list is inverted. The path is returned by the function as a list. The path and the total number of nodes expanded throughout the search are both returned by the ‘astar’ function. The start and goal nodes are made, and the open and closed lists are set up. The open list is implemented as a dictionary with the nodes serving as the keys and the f-values that go with them serving as the values. The algorithm advances by iteratively expanding the neighbors of the node with the lowest f-value on the open list and updating their g, h, and f values. The ‘return_path’ function is used to get the path and the number of enlarged nodes if the goal node is discovered. An empty list is returned in the absence of a path, indicating that there is not a legitimate route connecting the start node and the goal node.

3.3 Maze Solver Implementation

The code creates a maze of dynamic size based on the given data. The maximum X and Y coordinates of the disposal and rubbish locations are determined to establish the maze's size. Additional buffer space is added to accommodate the movement of the bin. The maze is then produced as a 2D list of zeros, indicating vacant cells.

There are several steps in the trash retrieval process, and it starts by the fact that many variables are initialized by the code, including the end path, the total number of enlarged nodes, and the weight and size of the bin. Secondly, based on their locations, the ‘get_rubbish’ function obtains the trash. It uses the A* Search algorithm to determine the shortest route that avoids both the disposal locations and the garbage locations from the bin's current location to each trash location via the ‘avoid_locations’ method which is able to store all the rubbish locations that exceeds the bin current capacity inside a list. The total number of nodes expanded throughout the search is recorded by the code. Thirdly, according to their lengths and the locations of the trash, the searched trash pathways are sorted. In the fourth step, each trash path is determined based on the bin capacity size and weight variables, and if the trash does not exceed the capacity, the trash is added to the final path list, and the size and weight of the bin are changed appropriately. The garbage is eliminated from the list of remaining garbage dumps. After that, the fifth part of the steps is that the code prints the current weight and size of the bin after the trash has been added, along with movement instructions to the trash site. The steps necessary to get to the location of the trash are printed by the ‘print_directions’ method which is used to calculate the X and Y coordinates differences in order to determine the actual movements of the bin, and it is also able to display the current coordinates of the rubbish as well. The sixth step is that when the bin is full, it calls the ‘get_disposal’ method to go on to the disposal stage. The seventh step is the ‘get_disposal’ method runs an A* Search from the bin's present location to each disposal location to determine which disposal location is closest. The position of the bin is then updated by choosing the shortest route. The bin's weight and dimensions are both set to zero. Lastly, the code publishes the current weight and size of the bin along with moving instructions to the disposal point, much like the trash retrieval step.

These steps are all executed within the ‘run_algorithm’ method by using a simple while loop to continually perform the methods until there is zero rubbish left inside the maze. The conditions of the arguments inside the while loop are that if the current weight or size is not equal to the bin’s capacity, the ‘get_rubbish’ method will execute and when the current weight or size is the same as the capacity, the ‘get_disposal’ method will execute instead.

3.4 Maze Visualization Implementation

This section pertains to the function in the program responsible for creating a graphical representation of a maze and its contents. It uses the `matplotlib` library to draw the maze using hexagons and labels each hexagon with relevant information.

The “draw_hexagon” method is defined with the parameters plot axis, center, size, color, text label, font size, and optional parameters for rubbish size and weight which determines the properties of the hexagon generated. This method is used to draw the individual hexagons on the subplot. Within the function, the coordinates of the hexagon points are calculated using trigonometry, then labelled and filled with color based on the passed parameters. The main function in the code is “create_hexagon_maze” and takes a “data” parameter, which is the dictionary containing information about the maze, including maze size, disposal locations, rubbish locations, bin location, rubbish weight, and rubbish size. If the bin location is not provided, it prints an error message and returns. To generate the maze, the code determines the maze dimensions by identifying the maximum X and Y coordinates. A subplot is then defined with appropriate dimensions to accommodate the maze visualization. Subsequently, a loop is initiated to iterate over the Y and X coordinates of the maze in reverse order. With each iteration, the center coordinates for the corresponding hexagon are calculated based on the current x and y values. The color and text label for the hexagon are determined based on its object type. If the location corresponds to a rubbish location, the size and weight of the rubbish are extracted and passed to the draw_hexagon function. After iterating over all the coordinates, the window title is set to "Maze", and the plot window is shown, which allows the code execution to continue without blocking the console. The resulting maze visualization is displayed to the user.

4.0 DYNAMIC CONFIGURATION DISCUSSION

In this section, the dynamic capabilities present within the developed application are described and elaborated upon in further detail in order to properly convey their functionality, their methods of execution as well as their interactions with other important functions. As a disclaimer, the dynamic configuration ability of the application is only limited to the difference in weight and size values of the nodes, the location of the nodes themselves, the capacity limit of the rubbish bin itself, and the size of the hexagon maze. This means that the shape of the hexagon maze is not changeable and is fixed in which the hexagons are shaped with a flat top and bottom and are arranged in an offset of even columns being lower than the odd columns.

The first dynamic capability of the application that will be discussed is the customizable node locations which allows the user to insert the specified nodes such as “Bin”, “Disposal” and “Rubbish” in any coordinates of the hexagon maze by either inputting the X-coordinates and Y-coordinates in the application via option 1 to 3 for the respective nodes or inputting them prior to the start of the application in the form of a text file and loading the nodes’ coordinates into the application by selecting option 4. To implement this dynamic feature into the developed application, each of the required nodes were stored into appropriate variables such as an empty list or a None type in order to populate or manipulate the information via updates, insertions and deletions later on. The information stored within these variables were set to the tuple format that corresponds to the X-coordinates and Y-coordinates. The ‘Rubbish’ and ‘Disposal’ node variables were created as empty list due to the reason that there are multiple instances of these two types of nodes, and the ‘Bin’ node variable was set as a None type because only one ‘Bin’ node can exist at one coordinate in the maze. Additionally, these node variables were initialized using the Python default constructor method.

Furthermore, these variables were then stored into a Python Dictionary, which is the equivalency of a HashMap, named ‘data’ using a function that also returns the ‘data’ variable. This function is used whenever the ‘data’ dictionary variable needs to be updated with the latest maze information that the user has input into the application. The ‘data’ dictionary is used to create and update the variables within the “maze_solver.py” file with the similar names, and these variables inside the “maze_solver.py” file were used so that the algorithm is able to decide on the paths it needed to take in order to efficiently visit the ‘Rubbish’ as well as the ‘Disposal’ nodes of the maze. The “maze_visualization.py” file also used the ‘data’ dictionary variable to create a dynamic graphical representation of the maze.

Moving on, the second dynamic feature of the developed application is the customizable rubbish weights and sizes when the user inputs a 'Rubbish' node. This feature is clearly demonstrated in the second option of the application or when the user enters the maze information inside a text file and uploads the information using the fourth option of the application. This dynamic feature contains many similar concepts and structures as the previous dynamic feature with the exception of its relation to the rubbish location variable. The weights and sizes of the 'Rubbish' nodes are being stored into an empty list respectively and are linked to the rubbish location variable via their index numbers. These two variables were also stored into the 'data' dictionary variable in order to be used by other methods as well as arguments such as deciding the times needed to travel to a 'Disposal' node, calculating the current weight and size of the rubbish bin, and also using them as error exception handling to prevent the user from inputting the rubbish bin capacity too low which causes the maze solver algorithm to stop functioning.

The third dynamic feature of this application is its capability to allow the user to change the rubbish bin limits when performing the maze solver algorithm. This feature is only seen when the user selects the sixth option which is to start the maze solver algorithm in order to find the paths needed to visit all the 'Rubbish' nodes within the predefined hexagon maze. The variables needed to implement this feature are two integer variables, each correlating with the rubbish weight and size, and are also initially set as zero. Both of these variables are later stored into the 'data' dictionary and used to update other variables with similar names. The reason as to why the rubbish bin capacity is a dynamic feature is because the minimum values allowed to be input by the user changes according to the rubbish weights and sizes within the maze.

Besides that, the fourth dynamic feature that the developed application contains is the size of the hexagon maze which changes depending on the locations of the 'Bin', 'Rubbish' and 'Disposal' nodes. In order to achieve this feature, two methods were created, one of them is used to determine the X-axis and Y-axis limits of the maze by calculation via checking both the 'Disposal' and 'Rubbish' nodes' coordinates as well as adding some buffer space for both X-axis and Y-axis. The maze size determining method will return the calculated axis limits as a tuple and a variable will be created to store the values. Next, a method is constructed and is used to create the maze according to the maze size variable values, and the created maze is then stored into a separate variable as a list containing the rows and columns of the maze. This

concept is performed in two instances, one in the maze solver algorithm file and the other inside the maze graphical visualization file with some minor alterations.

By implementing these dynamic features into the developed application, the application is able to handle many different maze configurations such as nodes placements, nodes values, maze size, and search algorithm conditions. However, these dynamic features are not able to accommodate different maze shapes and arrangements due to their limited capabilities. To conclude, the developed application is considered to be a dynamic maze solver algorithm that is suitable for hexagon shaped mazes where the hexagons must be rotated to have flat tops and bottoms.

5.0 RESULT & EVALUATION

For this section of the report, the results that the developed application produced based on the assignment specifications and requirements is shown below by directly taking from the terminal in the form of CLI, and the results are also accompanied by the graphical representation of the maze which can be seen in Figure 2. The results from the CLI will be evaluated based on their effectiveness in solving the assignment problem, their achievement in following the optimization requirement of least amount of time visiting the 'Disposal' node, and their overall weaknesses as well as strength in the perspective of both programmers and users.

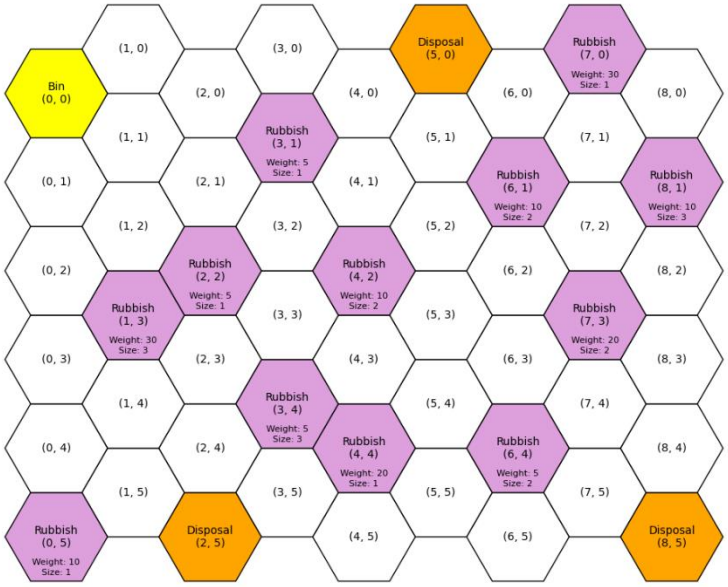

Figure 2 Maze Graphical Representation

5.1 Command-Line Interface Result

```
Fetching Rubbish:
Path: (0, 0) -> (1, 3)
Current Weight: 30 / 40
Current Size: 3 / 5
Direction:
Move       Movement       From        To          Coordinates
=============================================================================
1:       DOWN            (0, 0)      (0, 1)      (0, 0) to (0, 1)
2:       DOWN            (0, 1)      (0, 2)      (0, 1) to (0, 2)
3:       BOTTOM-RIGHT    (0, 2)      (1, 3)      (0, 2) to (1, 3)
=============================================================================

Fetching Rubbish:
Path: (1, 3) -> (2, 2)
Current Weight: 35 / 40
Current Size: 4 / 5
Direction:
Move       Movement       From        To          Coordinates
=============================================================================
1:       TOP-RIGHT       (1, 3)      (2, 2)      (1, 3) to (2, 2)
=============================================================================

Fetching Rubbish:
Path: (2, 2) -> (3, 1)
Current Weight: 40 / 40
Current Size: 5 / 5
Direction:
Move       Movement       From        To          Coordinates
=============================================================================
1:       TOP-RIGHT       (2, 2)      (3, 2)      (2, 2) to (3, 2)
2:       UP              (3, 2)      (3, 1)      (3, 2) to (3, 1)
=============================================================================

Going to Nearest Disposal:
Path: (3, 1) -> (5, 0)
Current Weight: 0 / 40
Current Size: 0 / 5
Direction:
Move       Movement       From        To          Coordinates
=============================================================================
1:       TOP-RIGHT       (3, 1)      (4, 0)      (3, 1) to (4, 0)
2:       TOP-RIGHT       (4, 0)      (5, 0)      (4, 0) to (5, 0)
=============================================================================

Fetching Rubbish:
Path: (5, 0) -> (6, 1)
Current Weight: 10 / 40
Current Size: 2 / 5
Direction:
Move       Movement       From        To          Coordinates
=============================================================================
1:       BOTTOM-RIGHT    (5, 0)      (6, 0)      (5, 0) to (6, 0)
2:       DOWN            (6, 0)      (6, 1)      (6, 0) to (6, 1)
=============================================================================
```

```
Fetching Rubbish:
Path: (6, 1) -> (4, 2)
Current Weight: 20 / 40
Current Size: 4 / 5
Direction:
Move       Movement       From        To          Coordinates
========================================================================
1:       BOTTOM-LEFT     (6, 1)      (5, 2)      (6, 1) to (5, 2)
2:       BOTTOM-LEFT     (5, 2)      (4, 2)      (5, 2) to (4, 2)
========================================================================

Fetching Rubbish:
Path: (4, 2) -> (4, 4)
Current Weight: 40 / 40
Current Size: 5 / 5
Direction:
Move       Movement       From        To          Coordinates
========================================================================
1:       DOWN            (4, 2)      (4, 3)      (4, 2) to (4, 3)
2:       DOWN            (4, 3)      (4, 4)      (4, 3) to (4, 4)
========================================================================

Going to Nearest Disposal:
Path: (4, 4) -> (2, 5)
Current Weight: 0 / 40
Current Size: 0 / 5
Direction:
Move       Movement       From        To          Coordinates
========================================================================
1:       BOTTOM-LEFT     (4, 4)      (3, 5)      (4, 4) to (3, 5)
2:       BOTTOM-LEFT     (3, 5)      (2, 5)      (3, 5) to (2, 5)
========================================================================

Fetching Rubbish:
Path: (2, 5) -> (0, 5)
Current Weight: 10 / 40
Current Size: 1 / 5
Direction:
Move       Movement       From        To          Coordinates
========================================================================
1:       TOP-LEFT        (2, 5)      (1, 5)      (2, 5) to (1, 5)
2:       BOTTOM-LEFT     (1, 5)      (0, 5)      (1, 5) to (0, 5)
========================================================================

Fetching Rubbish:
Path: (0, 5) -> (3, 4)
Current Weight: 15 / 40
Current Size: 4 / 5
Direction:
Move       Movement       From        To          Coordinates
========================================================================
1:       TOP-RIGHT       (0, 5)      (1, 5)      (0, 5) to (1, 5)
2:       TOP-RIGHT       (1, 5)      (2, 4)      (1, 5) to (2, 4)
3:       TOP-RIGHT       (2, 4)      (3, 4)      (2, 4) to (3, 4)
========================================================================
```

```
Going to Nearest Disposal:
Path: (3, 4) -> (2, 5)
Current Weight: 0 / 40
Current Size: 0 / 5
Direction:
Move       Movement       From        To          Coordinates
=============================================================================
1:       BOTTOM-LEFT     (3, 4)      (2, 4)      (3, 4) to (2, 4)
2:       DOWN            (2, 4)      (2, 5)      (2, 4) to (2, 5)
=============================================================================

Fetching Rubbish:
Path: (2, 5) -> (6, 4)
Current Weight: 5 / 40
Current Size: 2 / 5
Direction:
Move       Movement       From        To          Coordinates
=============================================================================
1:       TOP-RIGHT       (2, 5)      (3, 5)      (2, 5) to (3, 5)
2:       BOTTOM-RIGHT    (3, 5)      (4, 5)      (3, 5) to (4, 5)
3:       TOP-RIGHT       (4, 5)      (5, 5)      (4, 5) to (5, 5)
4:       TOP-RIGHT       (5, 5)      (6, 4)      (5, 5) to (6, 4)
=============================================================================

Fetching Rubbish:
Path: (6, 4) -> (7, 3)
Current Weight: 25 / 40
Current Size: 4 / 5
Direction:
Move       Movement       From        To          Coordinates
=============================================================================
1:       TOP-RIGHT       (6, 4)      (7, 4)      (6, 4) to (7, 4)
2:       UP              (7, 4)      (7, 3)      (7, 4) to (7, 3)
=============================================================================

Going to Nearest Disposal:
Path: (7, 3) -> (8, 5)
Current Weight: 0 / 40
Current Size: 0 / 5
Direction:
Move       Movement       From        To          Coordinates
=============================================================================
1:       DOWN            (7, 3)      (7, 4)      (7, 3) to (7, 4)
2:       BOTTOM-RIGHT    (7, 4)      (8, 4)      (7, 4) to (8, 4)
3:       DOWN            (8, 4)      (8, 5)      (8, 4) to (8, 5)
=============================================================================
```

```
Fetching Rubbish:
Path: (8, 5) -> (8, 1)
Current Weight: 10 / 40
Current Size: 3 / 5
Direction:
Move       Movement       From        To          Coordinates
================================================================================
1:       UP              (8, 5)      (8, 4)      (8, 5) to (8, 4)
2:       UP              (8, 4)      (8, 3)      (8, 4) to (8, 3)
3:       UP              (8, 3)      (8, 2)      (8, 3) to (8, 2)
4:       UP              (8, 2)      (8, 1)      (8, 2) to (8, 1)
================================================================================

Fetching Rubbish:
Path: (8, 1) -> (7, 0)
Current Weight: 40 / 40
Current Size: 4 / 5
Direction:
Move       Movement       From        To          Coordinates
================================================================================
1:       TOP-LEFT        (8, 1)      (7, 1)      (8, 1) to (7, 1)
2:       UP              (7, 1)      (7, 0)      (7, 1) to (7, 0)
================================================================================

Going to Nearest Disposal:
Path: (7, 0) -> (5, 0)
Current Weight: 0 / 40
Current Size: 0 / 5
Direction:
Move       Movement       From        To          Coordinates
================================================================================
1:       BOTTOM-LEFT     (7, 0)      (6, 0)      (7, 0) to (6, 0)
2:       TOP-LEFT        (6, 0)      (5, 0)      (6, 0) to (5, 0)
================================================================================

Final Path:
Path 1: (0, 0) -> (0, 1) -> (0, 2) -> (1, 3)
Path 2: (1, 3) -> (2, 2)
Path 3: (2, 2) -> (3, 2) -> (3, 1)
Path 4: (3, 1) -> (4, 0) -> (5, 0)
Path 5: (5, 0) -> (6, 0) -> (6, 1)
Path 6: (6, 1) -> (5, 2) -> (4, 2)
Path 7: (4, 2) -> (4, 3) -> (4, 4)
Path 8: (4, 4) -> (3, 5) -> (2, 5)
Path 9: (2, 5) -> (1, 5) -> (0, 5)
Path 10: (0, 5) -> (1, 5) -> (2, 4) -> (3, 4)
Path 11: (3, 4) -> (2, 4) -> (2, 5)
Path 12: (2, 5) -> (3, 5) -> (4, 5) -> (5, 5) -> (6, 4)
Path 13: (6, 4) -> (7, 4) -> (7, 3)
Path 14: (7, 3) -> (7, 4) -> (8, 4) -> (8, 5)
Path 15: (8, 5) -> (8, 4) -> (8, 3) -> (8, 2) -> (8, 1)
Path 16: (8, 1) -> (7, 1) -> (7, 0)
Path 17: (7, 0) -> (6, 0) -> (5, 0)

Search Summary:
Total Path Cost: 40
Total Nodes Expanded: 2188
Total Times Disposal Visited: 5
```

Based on the CLI results, the two most important pieces of information are displayed at the end and are labeled as “Final Path” and “Search Summary”. The “Final Path” section is used to display the paths needed to be taken in a specific order so that the assignment maze can be solved, and the “Search Summary” displays the three most important information such as the common search algorithm details as well as the information related to the developed application optimization requirements. All the paths taken by the A* Search algorithm is clearly labeled according to a numerical system starting from one with an increment of one each time, and these paths were arranged based on the traversal from ‘Rubbish’ node or ‘Disposal’ node to each other. The step-by-step solution to the assignment maze can be seen in the “Final Path” section in which all the paths are displayed and linked appropriately in order to make it understandable for the users.

Moving on, the “Search Summary” section has three important pieces of information that pertain to the efficiency and effectiveness of the A* Search algorithm in the perspective of various factors. The first information is the total path cost for the search algorithm to completely solve the assignment maze and the result states that the summation of all the paths is forty, where traveling to each node is equal to one. Next, the total amount of nodes expanded by the search algorithm is also stated and this information is important because the search algorithm itself can be changed or evaluated based on the calculated value, meaning this value is able to convey the overall performance of the A* Search algorithm. Lastly, the number of times the search algorithm travels to a ‘Disposal’ node is also displayed and this is due to the reason that the application was developed with the optimization requirement of visiting the ‘Disposal’ node the least number of times.

By analyzing the CLI results of the developed application, the proposed search algorithm is able to effectively achieve all the requirements of assignment as well as adhering to the optimization requirements however, the application is not able to efficiently solve the assignment because of two reasons which are the nodes expanded by the search algorithm and the total path cost. The nodes expansion of the A* Search is quite high and the reason for this is because the heuristic function is not optimal which causes the search algorithm to expand more nodes than necessary, and results in slower execution time for larger maze with more data sets. Besides that, the total path cost that is displayed by the application is also not efficient because of the decision-making logic during the path selection. The decision-making logic only selects the lowest path cost and does not consider the previous paths, meaning that for each

iteration of the path selection, the second lowest path cost is not remembered, and this causes the total path cost to be higher than necessary. The proper solution to resolve these two efficiency issues would be to implement a more complex and optimize heuristic function for the search algorithm and to improve the decision-making logic via the usage of caches or storing the second lowest path cost in a variable so that it can be compared during the path selection process.

6.0 CONCLUSION

In conclusion by using A* Search algorithm for this maze, the output of it is satisfied and reached the goal by successfully solving all the problems in this maze. In our implementation, the developed application includes dynamic features that allow users to customize various aspects of the maze solver algorithm. Users can input and customize the locations of nodes such as the bin, rubbish, and disposal sites either through the application or by loading a text file. The application supports multiple instances of rubbish and disposal sites, while only one bin can exist at a coordinate. Additionally, users can customize the weights and sizes of the rubbish nodes, which are stored in separate variables and linked to the corresponding rubbish locations. The application also allows users to change the capacity limits of the rubbish bin dynamically, which affects the algorithm's functionality. The size of the hexagon maze is determined based on the locations of the nodes, with buffer space added for movement. The application also provides a dynamic graphical representation of the maze using the matplotlib library. Overall, these dynamic features enable users to configure and solve mazes with different node locations, weights, sizes and bin capacities. Besides, the developed application also successfully solves the assignment maze and provides a clear step-by-step solution of the CLI results in this report. The "Search Summary" section displays important information such as the total path cost, the number of nodes expanded and the number of visits to the disposal node. However, there are efficiency issues in terms of the high number of nodes expanded by the A* Search algorithm and the higher-than-optimal total path cost. These issues can be addressed by improving the heuristic function and enhancing the decision-making logic during path selection. By implementing these improvements, the efficiency and effectiveness of the application can be enhanced, resulting in faster execution times and more optimal path solutions. Lastly, there are still future improvements we can make such as implementing a more advanced A* Search algorithm. This can lead to shorter and more optimal paths and also improve the overall efficiency of the maze-solving process.