Computer Science NEA

2023

Maze Navigating CAr

21233752 - Connor Kearney

 | [Company address]

Contents

[1 Analysis 4](#_Toc126220572)

[1.1 Problem identification 4](#_Toc126220573)

[***Table 1 - Identification of stakeholders*** 4](#_Toc126220574)

[***Table 2 - Computational Methods*** 5](#_Toc126220575)

[1.2 Research 7](#_Toc126220576)

[1.2.1 Existing similar solutions 7](#_Toc126220577)

[1.2.2 Features of the proposed solution 12](#_Toc126220578)

[1.2.3 A brief explanation of the LEGO MINDSTORMS EV3 unit 13](#_Toc126220579)

[1.2.4 Limitations of my proposed solution 14](#_Toc126220580)

[1.3 Requirements 15](#_Toc126220581)

[1.3.1 Software requirements 15](#_Toc126220582)

[1.3.2 Hardware Requirements 15](#_Toc126220583)

[1.3.3 Stakeholder requirements 15](#_Toc126220584)

[1.4 Success Criteria 16](#_Toc126220585)

[*Table 10 – Success Criteria* 16](#_Toc126220586)

[2 Design 17](#_Toc126220587)

[2.1 Overview of system 17](#_Toc126220588)

[2.1.1 System Navigation Chart 17](#_Toc126220589)

[2.1.2 System Layer Diagram 20](#_Toc126220590)

[2.2 Algorithms and further breakdown 21](#_Toc126220591)

[2.2.1 Car State Diagram 21](#_Toc126220592)

[2.2.2 Dijkstra’s Algorithm Flow Chart 23](#_Toc126220593)

[2.2.3 Search Algorithm Flow Chart 24](#_Toc126220594)

[2.2.4 Car’s Menu Flowchart 26](#_Toc126220595)

[2.3 Class layouts 28](#_Toc126220596)

[2.3 Variables, Classes, and Data Structures 29](#_Toc126220597)

[2.3.1 Descriptions 29](#_Toc126220598)

[2.3.2 Validation 32](#_Toc126220599)

[2.4 User Interface 33](#_Toc126220600)

[2.4.1 Car’s UI 33](#_Toc126220601)

[2.4.2 Desktop UI 34](#_Toc126220602)

[2.5 Sample Code 35](#_Toc126220603)

[2.5.1 Menus 35](#_Toc126220604)

[2.5.2 Search Algorithm 38](#_Toc126220605)

[2.5.3 Dijkstra’s 40](#_Toc126220606)

[2.6 Log File Handling 41](#_Toc126220607)

[***Figure 13 – Log file example*** 41](#_Toc126220608)

[2.7 Testing During Development 42](#_Toc126220609)

[2.7.1 Dijkstra’s Algorithm 42](#_Toc126220610)

[2.7.2 Search Algorithm 42](#_Toc126220611)

[2.7.3 Car’s Menu Algorithm 43](#_Toc126220612)

[2.7 Post Testing 43](#_Toc126220613)

[2.7.1 Menu Testing 43](#_Toc126220614)

[2.7.2 Search Algorithm Testing 44](#_Toc126220615)

[2.7.3 Dijkstra’s Algorithm Testing 45](#_Toc126220616)

[2.7.4 “Run All” Algorithm Testing 46](#_Toc126220617)

[2.7.5 Log File Testing 46](#_Toc126220618)

[2.7.6 Motor Control Testing 47](#_Toc126220619)

[3 Implementation phase 48](#_Toc126220620)

[3.1 Iteration 1 48](#_Toc126220621)

[3.1.1 Dijkstra’s 48](#_Toc126220622)

[3.1.2 Car’s Menu 52](#_Toc126220623)

[3.2 Iteration 2 53](#_Toc126220624)

[3.2.1 Dijkstra’s 53](#_Toc126220625)

[3.3 Iteration 3 55](#_Toc126220626)

[Appendix 55](#_Toc126220627)

[Easy Test Mat 55](#_Toc126220628)

[Medium Test Mat 56](#_Toc126220629)

[Difficult Test Mat 57](#_Toc126220630)

[Motor Control Test Mat 58](#_Toc126220631)

[Test Sheet Print Out 59](#_Toc126220632)

# 1 Analysis

## 1.1 Problem identification

Robots that can navigate small or restricted areas are invaluable in modern day engineering. They allow project managers and developers to fully map out an area, potentially for development or removal of rubble. Other applications of self-learning algorithms apply to many other sectors of the world, such as car development with calculating the path of least resistance for airflow or calculating the shortest distance between stations in a city. The nature of self-learning algorithms means they can adapt and evolve in a large array of ideas.

To continue this trend, I will hopefully be exploring a small part of this in self driving cars. I will specifically be using a car to navigate a maze using several sensors to simulate the ability for cars to detect a path to follow. I will be exploring this by creating a small vehicle and attaching either a distance sensor or a colours sensor to follow a specific path, further expanding the path and even adding different routes.

Table 1 below presents a breakdown of the stakeholders who might be involved in such a project where a self-driving car would need to navigate and solve a maze.

### ***Table 1 - Identification of stakeholders***

|  |  |  |
| --- | --- | --- |
| **Stake Holder** | **Stakeholder Description** | **How will they use the system and why is it appropriate to them?** |
| Myself (The Designer) | I will be working directly on the project as the principal developer. Therefore, I will be responsible for the development, testing and main design of the system. | Will want the system to be:   * User-friendly – The users should be able to easily setup and use the system without confusion * Efficient – The system should return the most efficient path each time * Independent – The system should be able to navigate the maze without user interference. |
| Engineer (Final user) | An engineer will be using this project to navigate and survey small spaces (e.g., tunnels, service ducts). | Will want to the system to be:   * Robust – The system should not malfunction whilst operating. * Reliable – The system should consistently return the correct values.   Will want to have a remote connection possible with real time diagnostics. |
| Apprentice (Final user) | An apprentice will be using the system to learn how to use and manage the system. | Will want the system to be:   * Easy to learn – The system shouldn’t be too complicated for new users. * Simplistic – The system should be simple so inexperienced apprentices can use it. |

Table 2 below outlines the different computational methods that will be used during the production of the solution.

### ***Table 2 - Computational Methods***

| **Feature of the system** | **Relevance of feature to the system** | **Computational Method Adopted to achieve feature** |
| --- | --- | --- |
| Calculations | The system must carry out a series of calculations, e.g.   * What distance has the car travelled * How long has the car been travelling * What direction has the car travelled * What is the velocity of the car * Calculation of the shortest route from ‘A’ to ‘B’ | * Distance computation = Summation of different parts of route length to calculate final length. * Velocity computation =  The car must track how long it has travelled for over the calculated length. * The system must record an accurate map of the area * The car will find the shortest route from A to B, then follow that route |
| Sorting | Different sets of data need to be sorted e.g.   * Route length | * Finding the shortest route by comparing different routes in a list. |
| Comparing data | Data needs to be compared e.g.   * Comparison of route lengths * Route and colours (using sensors) * ‘Blind’ dead end paths compared with viable routes | * Routes must be compared to each other to find optimal route * Potential routes should be compared to find viable routes (as compared to dead ends) * Comparison of previous sets of data for quicker search times |
| Storing data | Data must be stored e.g.   * Route lengths * The speed variable | * The car needs to store the different decisions it makes * Needs to be able to store the route to compare future routes |
| Searching data | The system must be capable of searching for data to optimise its output e.g.   * Searches for the minimum route length in a list * Areas to ignore during line searching algorithms | * The algorithm will search through the different routes to find the most efficient one * The previous routes will be searched to match to current route. |
| Decomposition | This is a key computational method whereby the system design is broken down e.g.   * Different algorithms * Different menus | * The algorithms will be clearly listed and presented, and each algorithm will be broken down * The different menus will be split up * The graph will be broken down into different lengths |
| Abstraction | Removes the unnecessary information to efficiently present a model of the solution | * The decisions, paths and corners will be presented as vertices on a graph |

## 1.2 Research

### 1.2.1 Existing similar solutions

#### 1.2.1.1 Example 1. AI in LEGO EV3 Maze-Driving Robot by Tony—K

##### 1.2.1.1.1 Goal of solution:

To navigate a maze, finding the optimal path, then executing the optimal path.

##### 1.2.1.1.2 Description of the Solution:

Multiple sensors are attached to a Lego EV3 car that can navigate a maze. The maze is first navigated inefficiently to crudely map out the maze and store the different directions it took. The directions are then converted into simplistic versions, reducing three instructions to one where possible. When placed back at the entrance, the car will navigate the optimal route back to the end.

##### 1.2.1.1.3 Features of the solution:

* Uses the left-hand wall following method
* Stores its path and eliminates dead ends
* Finds combinations in the movements in the maze and replaces them with simplified instructions
* Uses three different sensors as ev3 cannot manage more than one type of sensor
* Uses a 2-wheel chassis to move through the maze
* Uses a colour sensor to detect when the maze is finished

In table 3 below the positives and negatives of this solution are stated to gauge how effective the solution was.

##### *Table 3 - Positives and Negatives of the solution for Example 1:*

|  |  |
| --- | --- |
| Positives | Negatives |
| Efficient solution for solving the maze | Turning takes a large amount of space |
| Simple to reproduce for end users | Will be inefficient for larger mazes |
| Reduces the movements for optimal path | Does not allow for multiple ends |
|  | Does not allow for loops in the maze |

Figure 1 below shows an extract of the code used to program the solution. The language used is based on drag-and-drop principles, so would not be suitable for my solution.

##### *Figure 1 - A sample of code from solution Example 1:*

Graphical user interface, application, table, Excel

Description automatically generated

#### 1.2.1.2 Example 2. Design of a maze solving robot using Lego MINDSTORMS by B.J.S van Putten

##### 1.2.1.2.1 Goal of solutions:

To use LEGO Mindstorms RCX 2.0 to navigate a maze made of black on white line patterns.

##### 1.2.1.2.2 Description of the solution:

Uses the Mindstorm RCX 2.0 unit to explore the capabilities and restrictions of Lego hardware by developing a maze solving robot. The program designed will be able to efficiently navigate different maze scenarios and optimise a route through the maze, pushing the limits of the computing power of the RCX 2.0 unit. This identified the different restrictions of the unit and delivered solutions to the problems.

##### 1.2.1.2.3 Features of the Solution:

* Uses a programming language call NQC (Not Quite C)
* Uses the follow the wall method to navigate the maze
* Uses two motors set up to be fully rotational to move
* Uses a light sensor to understand where the maze is
* Simple program outline to follow
* Efficient for the capabilities of the RCX 2.0 unit
* Optimises the information gathered in the first run

##### *Table 4 - Positives and Negatives of the solution for Example 2:*

|  |  |
| --- | --- |
| Positives | Negatives |
| Simple Algorithm to follow and adapt | Unable to accurately detect dead ends and 90 degree turns |
| Once the track is solved, can solve it efficiently | Battery of the robot can reduce accuracy |
|  | For much larger mazes, the RCX will run of memory |
|  | Cannot be easily reproduced |

##### 1.2.1.2.4 Code from the solution

The code presented below is an extract of the code used in the solution. This code is accompanied by an image in the solutions. See figure 2.

|  |
| --- |
| Right:  A visualisation of the solution’s algorithm:  If (on black line) {  Forward (left motor);  Off (right motor);  }  Else (on white) {  Forward (right motor);  Off (left motor);  } |

Figure 2 below shows an image representation of the method used to search the maze.

##### Diagram Description automatically generated*Figure 2 – A sample image of the tracking system in solution Example 2*

### 1.2.2 Features of the proposed solution

#### 1.2.2.1 Initial Concept of my solution considering my research

My initial concept will be using the EV3 unit to follow a maze comprised of black lines on white background.

There are two different algorithms that exist that can navigate from A to B in a maze. These are presented in Table 5 below under the headings ‘Dijkstra’s’ and ‘Floyd’s’ algorithms respectively.

##### *Table 5 – Two algorithm approaches.*

|  |  |  |
| --- | --- | --- |
|  | Dijkstra’s | Floyd’s |
| Description | A simple algorithm that will compute the shortest distance from a single vertex to any other vertex | A complex, multistage algorithm that will compute the shortest distance from any vertex to any other vertex |
| Positives | Simple to code the algorithm  Much more efficient to run | Simple to implement using arrays  Returns the shortest Hamiltonian cycle |
| Negatives | Cannot be used when multiple goals must be achieved | Uses a large amount of memory |
| Efficiency |  |  |
| Suitability | Is suited towards smaller mazes with only a single goal | Is suited towards larger mazes where multiple goals must be achieved |

##### 1.2.2.1.1 Final comparison between algorithms

Due to its much more efficient nature, and not needing to compute the distance between many goals, I will be using Dijkstra’s algorithm to find the shortest path through the maze. This is different to the previous solutions, as they both use the left or right-hand wall hugging method before simplifying their results. My solution will be more durable than the other solutions, which have a 20% success rate (from solution 2).

##### 1.2.2.1.2 Sensor configuration

I have chosen to use a single colour sensor to navigate the maze for now. This is because it will be much more efficient to build and modify the maze compared to physical walls. This also allows me to calculate the result of Dijkstra’s algorithm beforehand to ensure the algorithm is working as intended. Also, this method is much more cost effective than other methods such as up-right walls, allowing me to develop the algorithm and work out any flaws.

##### 1.2.2.1.3 Software choice

Furthermore, I will be using EV3DEV software and OS to run my code. This allows me to use Bluetooth to communicate with the car and upload my program with ease. This means I am using MicroPython to program and control the car, as I am familiar with python and will be able to develop my code using Visual Studio Code.

The user will be able to interact with the car using a GUI interface located on the LCD display of the EV3 unit.

### 1.2.3 A brief explanation of the LEGO MINDSTORMS EV3 unit

Since I have decided to use an EV3 set to execute this project, I will explain the aspects of the EV3 that are useful in this project, and some that limit the potential of my final product. I will be further comparing the EV3 to a previous unit and stating why features of the EV3 are more suited to my project.

The specifications of the EV3 and NXT units are presented in Table 6 below.

#### *Table 6 – EV3 and NXT specifications*

|  |  |  |
| --- | --- | --- |
|  | EV3 | NXT |
| Display | 178x128 pixel | 100x64 pixel |
| Main Processor | 300 MHz | 48MHz |
| Main Memory | 64MB RAM, 16MB flash | 64KB RAM, 256KB flash |
| Extra Features | USB host port, Wi-Fi (through dongle), BT | BT |
| Ports | 4 sensors, 4 motors | 4 sensors, 3 motors |
| weight | 215g | 213g |

#### 1.2.3.1 Comparison of the EV3 and NXT and final choice

As a result of the low RAM, the limited CPU speed and smaller screen resolution the NXT is not as suitable for this project as the EV3, however if an OS allows EV3DEV code to be run on the NXT, then if the algorithm is efficient enough then the code should still run on the NXT.

As for the EV3, the larger RAM and much faster CPU speed are definite bonuses, as well as the increased screen resolution. The larger RAM will allow me to store many more data values in a list or an array, allowing the car to perform the search on larger mazes. The fast CPU means I can run many operations per second without the EV3 slowing down considerably, meaning that the maze can be traversed quicker, and the sensors will be more accurate in their timings. The screen size, whilst better than the NXT, is still very small, and will only allow me to have 4 different lines with a readable font, so navigation will have to be optimised to account for this.

The use of LEGO allows the project to be easily expandable and if any aspect does break then parts can be easily replaced. However, if the concept of this project were to be applied to a real-life scenario, then a much sturdier and durable option should be considered as LEGO may not be acceptable in some circumstances.

### 1.2.4 Limitations of my proposed solution

The main limitation in my proposed solution is the necessity of a set up maze. This means that the car will not be very applicable to the open world in its finished state as of the previously stated specifications. This will result in the product potentially not meeting the system requirements. One solution to this will be that the colour sensor could be swapped out for an ultrasonic or IR sensor.

Further limitations of this design include the inability for the end user to change any of the code or perform any maintenance whilst in the field without any extra equipment, which means the car will not be stand-alone.

One further limitation is that this solution will have to be very memory efficient. The EV3 has only 64MB of RAM. This means that any longs lists, or arrays of data must be stored efficiently, and the car will only be able to navigate a small to medium sized maze as larger mazes may take up too much RAM, so some optimisation **must** be used.

## 1.3 Requirements

### 1.3.1 Software requirements

**Brickman OS** – This is the OS that is loaded onto an SD card so the EV3 can understand and interpret the code, as well as interfacing with the motors.

**Visual Studio Code** – Used to edit and load the code onto the car

**EV3Dev extension** – This is the extension that communicates with the car to transfer the code to the car

### 1.3.2 Hardware Requirements

**EV3 Brick and base set** – This will be what the car is built from, so will be needed to reproduce the final product.

**Micro SD Card** – The Brickman OS will need to be installed onto some medium for the brick to understand. The EV3 unit has a built-in micro-SD card reader which can be booted from.

**A Maze or path to follow and solve** – The car must have something to follow and solve to work as intended.

**A computer with Bluetooth or USB** – This allows for the program to be loaded onto the car

### 1.3.3 Stakeholder requirements

Table 7 below discusses the different requirements that are dictated by the design of the system.

#### *Table 7 - Design*

|  |  |
| --- | --- |
| Requirement | Explanation |
| Simple Menu | The menu must be simple so that the users can easily understand how to navigate and use it |
| Fully independent | The car being fully independent, past pressing the “run” button, means that it will allow operation to be hassle free |
| Simple Instructions with screenshots | This allows users to understand each function of in the menu |

Table 8 below discusses the different requirement surrounding the functionality of the system.

#### *Table 8 - Functionality*

| Requirement | Explanation |
| --- | --- |
| Use of the EV3 buttons to interact with the software | The software on the EV3 unit will require that buttons of the unit be pressed to begin operation. This allows the user to navigate the different menus and run different subroutines |
| Easy setup from box to final product | The setup process must be easy for users to recommend and continue to use the product in mass and to deploy remotely |
| The ability to remote into the EV3 unit | The ability to receive information and see the EV3 screen remotely will allow for the users to actively see the state of the car |
| The ability to change the speed of the motors | Users may wish to change the speed as the car is traversing different areas, so be able to change the speed will be useful |
| Different menus for stats and settings of the car | The user may wish to see the different values that the EV3 unit has stored or is using to allow for more in-depth debugging or logging |

Table 9 below discusses the hardware and software requirements for the project.

#### Table 9 - Hardware and Software

|  |  |
| --- | --- |
| Requirement | Explanation |
| A computer with USB or Bluetooth | This will allow the user to load the code and OS onto the EV3 brick and interact with it once the code is loaded |
| Brickman OS image | This is the OS that will be used to understand and process the code that will move the car |
| Micro SD Card | This will hold the Brickman OS image as well as any code loaded onto the car |
| Visual Studio Code with EV3DEV | This will be needed to load code from the computer onto the car. However, this can be avoided by creating a premade OS image of Brickman that includes all the code which can be loaded directly onto the car’s storage |
| Lego EV3 Unit | This will be the main ‘brain’ of the car. All algorithms and subroutines will be processed using this. |

## 1.4 Success Criteria

Table 10 states the chosen success criteria for the project and states the conditions under which the criteria have been successfully reached.

### *Table 10 – Success Criteria*

|  |  |
| --- | --- |
| Criteria | Evidence of completion |
| Simple User interface | Have stakeholders use the GUI and assess if they can navigate the menus efficiently and with little hassle. |
| Algorithm that can navigate mazes consistently | The algorithm should complete the maze at least 90% of the time. At least 10 tests should be run on a medium sized maze |
| Memory efficient design | The code and algorithms use no more than 75% of the available memory (48MB). |
| The ability to stop the program | A button that can turn off the EV3 |
| The ability to remotely connect to the system | Screen shots of the remote system capability |
| The ability for the car to be stand alone | The car completes at least 90% of mazes without interference. At least 10 tests should be run. |

# 2 Design

## 2.1 Overview of system

The system will be broken down into several different aspects that comprise the UI in the front end and the algorithms in the back end. Each diagram is an abstraction and decomposition of the system, allowing me visual aids with my programming.

### 2.1.1 System Navigation Chart

Figure 4 (below) is a visual representation of the navigation system for the user. It describes the different elements of the UI and how each one will interact with the user.

This chart is split into two keys sections, i.e., Car Menus AND Desktop Menu/UI. The initial sub-section, “Car Menus”, presents the four key visual components of the car’s user interface (i.e. visible to the user). The main menu sub-section presents the key visual aspects of the desktop user interface.   
Each red subsection implies that a menu will be interacting with the lower sections. Each green subsection implies a subroutine will be executed once the user interacts with it, returning the previous menu once completed.

The chart is broken down in such a way that allows me to visually aid my programming, so the structure of the menu system stays intact and robust. The chart is an abstraction of the high-level aspects of the system.

**Justification**

The chart breaks down the system into a clearer format for the client to understand the links between the different menus. This means the client can more clearly visualise the final navigation system.

#### ***Figure 4 – System Navigation Chart***

Diagram

Description automatically generatedKey: Menu, Subroutine, Misc.

### 2.1.2 System Layer Diagram

The diagram in figure 3 is a visualisation of the flow of data between subsections of the system, with each layer representing a different type of subsection. This chart allows me to visually see a decomposition and abstraction of the system including subroutines and data structures so I can quickly implement the different areas of my project.

Figure 3 also displays the interaction of the subsystems between different layers, such as the Main Menu leading to individual screens, each with a unique set of subroutines. These subroutines then interact with the data structures and variables within the code, and how one subroutine can interact with another object in the system.

**Justification:**

This diagram visually decomposes the different high level and low-level aspects of the system, and shows how they can interact with each other, allowing a potential client to fully understand the importance of each component in the system.

Chart, diagram

Description automatically generated***Figure 3 – Layer Diagram***

## 2.2 Algorithms and further breakdown

### 2.2.1 Car State Diagram

Figure 5 visually maps out each state that the car’s system can be in, showing the different paths that the state may follow during operation. The right section of the diagram shows the changes in state when the “execute subroutine” path is taken, with the left section of the diagram showing the changes in state when the “search subroutine” path is taken. The diagram enables me to accurately detail the general actions taken during a subroutine and how the car should behave at certain points.

This diagram shows the possible states that can occur during the operation of the system, and the transition between these states.

**Justification:**

Figure 5 provides a clear and concise visual representation of the system behaviour during operation, making it easy for a potential client to understand how this system will work. The diagram allows for the representation of complex behaviour in an easy to understand and communicate way.

#### ***Figure 5 – State diagram***



### 2.2.2 Dijkstra’s Algorithm Flow Chart

#### Diagram Description automatically generated***Figure 6 – Dijkstra’s algorithm Flow chart***

*This section loops through each of the possible 4 directions that arc could be, and if the current weight of the vertex is more than the new weight, than the weight is updated.*

*This section means the algorithm will iterate through all the vertices until the end node is permanent. This means all the different paths are explored and their lengths compared to find the best one.*

Figure 6 is an abstraction of the widely used Dijkstra’s algorithm used for network traversal. The algorithm used is an efficient and robust algorithm, which means the system will be able to run even with minor flaws in the detection algorithm or the environment. The structure of the algorithm allows for either recursion or iteration to be used, giving a flexible approach in its coding.

### 2.2.3 Search Algorithm Flow Chart

Figure 7 is an abstraction of the search algorithm that is going to be used. The search algorithm is used to create the initial network of corners used to represent the maze so the car can solve a path for it. The search algorithm involves different techniques to ensure that the system will be able to detect if there is a corner or the car has simply missed the line. This allows the system to remain robust and efficient, whilst also being reliable.

#### ***Figure 7 – Search algorithm flow chart***

*This subroutine checks if there is a black line between start\_point and end\_point. The 2 variables represent angles of a circle around the car. Positive and negative .*

### 2.2.4 Car’s Menu Flowchart

#### ***Figure \_\_\_ - Menu Flowchart***



*This section handles the user inputs each time the screen refreshes. The up and down buttons move the pointer, and the centre button selects the current item.*

## 2.3 Class layouts



Figure \_\_\_ shows the different classes involved in the project with their interactions. Each class has its name in red, properties in green and methods in blue. A “+” denotes a public attribute or method and a “-” denotes a private attribute or method.

The main class in this project is called “Main”. This allows me to centrally manage all the different objects through one object. Any “Main” object will have several different “Menu” objects, each containing the information for the menus displayed onto the EV3 screen. Along with this, a unique “Car” object will be created within the runMain () method of the “Main” object. This object is for storing all the information relevant to the operation of the car, such as the EV3 brick, the motors, the sensors, and several subroutines.

The “Car” object will have a “Graph” object which is made up of many “Vertex” type objects. Each “Vertex” object will represent a different corner or dead end. The “Graph” object has the Dijkstra’s algorithm as a public method, as the code in the “Car” object will be able to execute this.

## 2.3 Variables, Classes, and Data Structures

### 2.3.1 Descriptions

|  |  |  |  |
| --- | --- | --- | --- |
| **Variables** | **Appears in** | **Description** | **Justification** |
| Start\_point | Search Algorithm | States the angle the car should turn from | Allows modularity in the system as a subroutine can be used |
| End\_point | Search Algorithm | States the angle the car should turn to | Allows modularity in the system as a subroutine can be used |
| Increment | Search Algorithm | States the angle the car should turn before checking for a line. | Allows for a variable increment rather than fixed |
| Dir | Dijkstra’s | A counter used to check each direction of a vertex | Allows for looping using an integer value, so the system will be more efficient |
| Weight from start | Dijkstra’s | The weight of a vertex from the start vertex in the maze | Allows the priority queue to be stored, and is used to calculate the shortest path |
| Start label | Dijkstra’s | The label of the start vertex in the maze | Used to backtrack through the network to find the final path the car must take |
| End label | Dijkstra’s | The label of the final vertex in the maze | Used to tell Dijkstra’s algorithm when to stop |
| Weight of arc | Dijkstra’s | The weight of an arc connecting 2 vertices | Needed for updating the weight of each vertex in the network |
| Permanent | Dijkstra’s | A Boolean to determine if a vertex has a permanent label | Will help to determine if a vertex has been made |
| Pointer | Menu Algorithm | Used to determine where the arrow should point in a menu | A fixed value won’t allow me to move where the pointer arrow is |
| IsActive | Menu Algorithm | Used to loop the menu output | Allows me to easily stop the main loop from anywhere inside the class, and recursion is not a viable replacement for this loop |

|  |  |  |  |
| --- | --- | --- | --- |
| **Classes** | **Appears in** | **Description** | **Justification** |
| Vertex | Search Algorithm, Dijkstra’s | Used to store information about corners | Gives the system modularity and consistency by using a single class to represent a repeating occurrence. |
| Graph | Search Algorithm, Dijkstra’s | Used to store and process all vertices in the maze | Creates a simple class that can store vertices consistently |
| Car | Search Algorithm, Menu Algorithm | Used to interact with the motors and sensors | Allows me to have validation and consistency in the way the motors and sensors behave, as well as store all the movement subroutines |
| Main | Menu Algorithm | Stores the runMain () subroutine | Python does not include a “main” function, so this class allows me to mimic techniques from other languages such as C# |
| Menu | Menu Algorithm | Stores the information needed to output a menu | Creates an easy template to fill in a menu. This means the code is modular |

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structures** | **Appears in** | **Description** | **Justification** |
| Graph | Search Algorithm, Dijkstra’s | An abstraction of the maze using corners as vertices and arcs as paths on the ground | Abstracts any unnecessary information from the maze, allowing for an easy understanding of the maze. Also, Dijkstra’s requires a graph to work on |
| Priority Queue | Dijkstra’s | A queue that is first sorted by some value (weight from start), before being enqueued or dequeued | Needed for Dijkstra’s to determine which vertex to visit next. |
| List | Dijkstra’s | A simple dynamic data structure that will store all the permanent vertices | Needed to separate the permanent vertices from the other vertices without deleting them completely |
| Log Files | Logging | Some data will be stored in log files. | Will be useful when trying to debug errors in the system. Also, will be used for storing serialized objects through pickle. |

### 2.3.2 Validation

| **Variable** | **Description of Validation** | **Justification for validation** |
| --- | --- | --- |
| Start\_point | Integer between -360 and 360 | The angle of turning may need to be negative, so a negative lower bound is needed |
| End\_point | Integer between -360 and 360 | The angle of turning may need to be negative, so a negative lower bound is needed |
| Increment | Integer | As turning in floats will be difficult to get precise |
| Dir | Integer between 0 and 3 | As there are only 4 directions Dijkstra’s needs to look in |
| Weight from start | Positive float greater than weight before | The replacing weight should be less than |
| Start label | Unique | If there is a label applied to more than one vertex, than Dijkstra’s won’t work |
| End label | Unique | If there is a label applied to more than one vertex, than Dijkstra’s won’t work |
| Weight of arc | Positive float value | Arc weights can’t be negative, as they represent distances |
| Pointer | Positive integer between 0 and 3 | There are only 4 options in a menu, if pointer goes beyond 3, the arrow pointer on screen will be too low |

## 2.4 User Interface

The user interface will be split into two sections, visible on the car and visible on a desktop.

### 2.4.1 Car’s UI

Figure 8 contains the design choice for the 4 different menus that are going to be active on the car. The user interface of the car must be simple and user-friendly, which is demonstrated in the above diagrams. Due to hardware restrictions, a maximum of 4 lines can be printed onto the screen at any time. This does however contribute to the overall user experience as the user is not overwhelmed by choices. Each menu will have a title to tell the user where they are, and up to 4 different operations they can execute, including exiting back the main menu or exiting the program.

#### ***Figure 8 – Car UI***

Diagram

Description automatically generated

### 2.4.2 Desktop UI

Figure 9 is a general abstraction of the proposed desktop user interface. The Desktop UI has all the features of the car’s UI in one window, allowing for the user’s to easily navigate the functionality of the car. The diagram is useful as it means I can visualise the desktop experience.

The desktop UI will have all the same features of the Car’s UI laid out in one window, allowing the user to see everything. The UI here will be split into two distinct sections, one being monitoring and the other be interactive.

The monitoring section will display all the information the car is sending to the computer, while the interactive section of the display allows for the user to change aspects of the cars settings and to run subroutines from the desktop.

#### Table Description automatically generated with low confidence***Figure 9 – Desktop UI***

## 2.5 Sample Code

### 2.5.1 Menus

Since I have been testing the Menus in python to gauge how possible the solution is to produce, I have coded a simple menu program with only one menu to navigate. This will allow me to expand from this example. This is not pseudocode as pseudocode has a very limited approach to importing specific libraries with specific syntax, and since the syntax is different between python and pseudocode, it would not make sense to use pseudocode whilst working with the import section of the code.

#### ***Figure 10 – Python code for a single menu***

|  |
| --- |
| #!/usr/bin/env pybricks-micropython  #This is the Second menu prototype, which will include running subroutines from the menu  import time  from pybricks.hubs import EV3Brick  from pybricks.ev3devices import (Motor, TouchSensor, ColorSensor,  InfraredSensor, UltrasonicSensor, GyroSensor)  from pybricks.parameters import Port, Stop, Direction, Button, Color  from pybricks.tools import wait, StopWatch, DataLog  from pybricks.robotics import DriveBase  from pybricks.media.ev3dev import SoundFile, ImageFile  # This program requires LEGO EV3 MicroPython v2.0 or higher.  # Click "Open user guide" on the EV3 extension tab for more information.  # Subroutines  def Run():  screen.clear()  screen.print("Run is running")  car.straight(1000) # forwards 1000 mm  time.sleep(1)  def LeftRight():  screen.clear()  screen.print("Right is running")  car.turn(90)  time.sleep(1)  screen.clear()  screen.print("Left is running")  car.turn(-90)  time.sleep(1)  def IDLine():  screen.clear()  screen.print("IDLine is running")  while colsense.color()!=Color.BLACK:  screen.print(colsense.color())  time.sleep(0.001)  def FindLine():  screen.clear()  screen.print("FindLine is running")  time.sleep(1)  # Create your objects here.  ev3 = EV3Brick()  screen = ev3.screen  buttons = ev3.buttons  motorA = Motor(Port.A)  motorD = Motor(Port.D)  car = DriveBase(motorD,motorA,31,190) # wheels have diameter of 31mm and a drivebase width of 190mm  colsense = ColorSensor(Port.S4)  sensorList = [colsense]  # Write your program here.  MainMenu = ["Main Menu","Run", "Left RIght", "ID Line", "Find Line"]  MainMenuLookUp = [Run, LeftRight, IDLine, FindLine]  current\_menu = MainMenu # allows for the main loop to update which menu it uses  current\_menuLookUp = MainMenuLookUp  pointer = 0  car.settings(straight\_speed=200)  while True:  # Output to screen  screen.clear()  for i in range(len(current\_menu)):  if i != 0:  if pointer+1==i:  screen.print(current\_menu[i]+" <--") # adds the cursor to the end of the text line  else:  screen.print(current\_menu[i]) # prints all the other lines in the menu  if i == 0:  screen.print("")  screen.draw\_text(75, 0, current\_menu[0])  # navigation  if Button.DOWN in buttons.pressed():  pointer+=1  time.sleep(0.3)  if Button.UP in buttons.pressed():  pointer-=1  time.sleep(0.3)  if Button.CENTER in buttons.pressed():  try:  current\_menuLookUp[pointer]()  except Exception as e:  print(e)  time.sleep(1)  time.sleep(0.1) |

### 2.5.2 Search Algorithm

The pseudocode in figure 11 is used to navigate the maze and map each corner to a vertex, creating a graph for Dijkstra’s algorithm to use.

#### ***Figure 11 – Pseudocode for the search algorithm***

|  |
| --- |
| SUB Search()  SUB CheckLine(int start\_point, int end\_point)  Increment = (end\_point - start\_point)/10  Car.TurnTo(start\_point)  current\_angle = start\_point  DO WHILE current\_angle < end\_point:  Car.turn(increment)  IF ColourSensor.Colour() == "Black" THEN  RETURN TRUE  END IF  END WHILE  RETURN FALSE  END SUB  Car.State = "Searching"  DO WHILE Car.State == "Searching"  Car.Forward()  IF CheckLine(0,0) THEN # if the line is straight ahead  PASS  ELSE IF CheckLine(-45,45) THEN # if the line is roughly ahead  PASS  ELSE IF CheckLine(45, 135) THEN  Car.Graph.AddVertex()  Car.Graph.Vertex.PathD2 = TRUE  Car.Graph.Vertex.PathD1 = TRUE # adds a right arc to the vertex  ELSE IF CheckLine(-135, -45) THEN  Car.Graph.AddVertex()  Car.Graph.Vertex.PathD2 = TRUE # adds a backwards arc to the vertex  Car.Graph.Vertex.PathD3 = TRUE # adds a left arc to the vertex  ELSE IF CheckLine(135, -135) THEN  Car.Graph.AddVertex()  Car.Graph.Vertex.PathD2 = TRUE # adds a backwards arc to the vertex  END IF  END WHILE  END SUB |

### 2.5.3 Dijkstra’s

In figure 12 I have coded Dijkstra’s algorithm. This will take a graph input and output the shortest path from the start node to the end node with labels. This code does require the specific objects created elsewhere in the program, so they will need to be fully implemented before this code can start being made.

#### ***Figure 12 – Dijkstra’s algorithm pseudocode***

|  |
| --- |
| SUB Dijkstras(Graph)  Visited = []  Unvisited = []  Unvisited[0] = Graph.List[0] # adds the first vertex to the unvisited list  Unvisited[0].Cost = 0  start = Unvisited[0]  FOR i=1 to Graph.List.Length  Unvisited[i] = Graph.List[i]  Unvisited[i].Cost = Math.inf # sets the verticies cost from start to infinity  END FOR  DO WHILE Unvisited.Empty = FALSE  Unvisited[0].perm = TRUE # makes the current Vertex permanent  currentVertex = Unvisited[0]  dir = 0  DO WHILE dir < 4 # check each direction  IF currentVertex.dirList[dir].cost > currentVertex.cost + currentVertex.dirCostList[dir] THEN  FOR EACH item IN Unvisited  IF item.label == currentVertex.dirList[dir].label THEN  item.prev = currentVertex.label  item.cost = currentVertex.cost + currentVertex.dirCostList[dir]  END IF  END FOR  END IF  END WHILE # end dir check  Visited.add(Unvisited[0])  Unvisited.delete(0)  Main.Sort(Unvisited) # sort using bubble sort  END WHILE  # Return the final path taken to get to the end  path = []  currentLabel = Graph.end.label  DO WHILE currentLabel != start.label:  path.add(currentLabel)  FOR EACH item IN Unvisited  IF item.label =  END WHILE  END SUB |

## 2.6 Log File Handling

For my log file handling and data storage I will be using Pickle, a module that allows for objects to be serialized and stored as text, then retrieved in the future from a simple text file. This will allow me to store the entire graph data structure in a text file, if not multiple graphs. From this, I can simply import the graphs and compare whilst Dijkstra’s algorithm is running.

For the log files, I will be using the integrated python file manager, and simply writing lines of text with a time stamp to indicate when tasks have been successfully completed. If tasks were to fail, I can use a “Try Except” statement to catch any errors and store them as a string. Figure 13 is an example of what the log file might look like, although the design might change to better suite different situations.

### ***Figure 13 – Log file example***

|  |
| --- |
| [00:00:00] Boot  [00:00:01] Main Initialized  [00:00:02] Car Initialized  [00:00:03] Graph Initialized  [00:00:04] Menus Initialized  [00:00:10] User Input – Menu Change  [00:00:16] User Input – Variable Change  [00:00:30] User Input – Run All Sub  [00:07:26] Sub Fin – Run All |

## 2.7 Testing During Development

During development certain key aspects of the coded solution must be tested, such as loops and subroutines. Several tests must be developed to ensure that the code is working as intended and returning the correct result using the correct method.

Each test is **justifiable**, as each part has been decomposed and abstracted into its smallest components. Hence, each test ensures that each individual part of the system is working. These tests allow me to ensure that even if the correct data is output, that the correct algorithm is occurring in the code. This means that the system will be more robust if the user inputs data incorrectly.

### 2.7.1 Dijkstra’s Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Test Data | Expected Result |
| 101 | The vertex classes’ ability to instantiate and update its adjacency through the graph class. | Labels and weights are passed in. Then, the list will be iterated through, and vertex attributes are output. | All the vertex values are unchanged from their inputs. |
| 102 | Dijkstra’s loop. The shortest path is calculated. | Some vertex values with labels and weights. | The algorithm should return the shortest path from “A” to the end, as well as the total weights of each vertex. |
| 103 | Find vertex subroutine. | Vertex labels are passed in, with a network list already built | The subroutine should return the vertex object with the specified label. |
| 104 | Bubble sort algorithm | Values are passed into the bubble sort. | The list outputted should be in ascending order |

### 2.7.2 Search Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Test Data | Expected Result |
| 105 | Check line algorithm. This is the algorithm responsible for following the line. | The car placed down and will trace out the nearby adjacency of a vertex. | The car should be able to navigate the maze using the right wall. |
| 106 | Add vertex subroutine from graph class. | The car will be placed on a simple maze and trace it out. | The graph should be populated with vertexes, each with weights and adjacent vertexes. |
| 107 | Simple maze navigation without creating any vertices. Just looking at the consistency of the search algorithm. | A simple maze, increasing in vertexes each time the test is ran. | The car should be able to navigate a maze completely without adding any vertices. |

### 2.7.3 Car’s Menu Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Test Data | Expected Result |
| 108 | The draw subroutine of the menu handles how the menu’s attributes are visible on the screen. | A menu object should be created with a name and titles. | The screen should be evenly proportioned with none of the text off the screen. |
| 109 | Pointer movement is important for the user to interact with the system. | The buttons on the EV3 brick will be pressed. | The visual pointer should increment and decrement with each press of the respective buttons. |
| 110 | Menu’s linking will be important with the user experience. | Menus should be created and linked together. The menu systems should then be navigated. | The screen should update with each menu change, the pointer should return the 0. |

## 2.7 Post Testing

Each section of the solution must be testing using iterative methods to test whether they are suitable. This means that several tests must be designed and reviewed to ensure that the output shows that the section meets the success criteria.

### 2.7.1 Menu Testing

Both types of menus need to be tested to ensure that they are suitable for the end users.

#### 2.7.1.1 Car’s UI

Table 11 lays out the different tests that will be used to determine whether the car’s UI has been successful.

##### Table 11 – Tests for the Car’s UI

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Test Data | Expected Result |
| 01 | The EV3 brick will be handed to several people. They will be asked to navigate to and indicate towards different subroutines within the menu. | A random list of subroutines to find. | That the person taking part in the test can navigate to every subroutine in a random list. |
| 02 | Many buttons will be pressed to test if the program can take many inputs at once without freezing. | Button inputs. | The system should not slow down at all. |
| 03 | The pointer will be increased past 4. | - | The system should remain operational when the pointer is between 0 and 3 |
| 04 |  |  |  |

#### 2.7.1.2 Desktop UI

Table 12 lays out the different tests that will be used to determine whether the Desktop UI has been successful.

##### Table 12 – tests for the Desktop UI

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Test Data | Expected Result |
| 05 | The usability of the UI must be tested. A list of subroutines will be given, and the user must point them out. | A random list of information to find. | All the item on the list can be found within a few seconds. |
| 06 | Many buttons pressed to test how the program can deal with many inputs | Buttons inputs | The system should remain stable. |

### 2.7.2 Search Algorithm Testing

Table 13 lays out the different tests that will be used to determine whether the Search algorithm has been successful.

#### Table 13 – Tests for the search algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Test Data | Expected Result |
| 08 | Car must navigate a test mat with 4 vertices (start, 2x corner, end) | The lines on the easy mat. | The car will be able to navigate the mat. |
| 09 | Car must navigate a medium difficulty maze (start, 3-6x corner, end, dead ends) | The lines on the medium mat. | The car will be able to navigate the mat with minor difficulty |
| 10 | Car must navigate a large and very difficult maze. (Start, 8+ corners, end, dead ends) | The lines on the difficult mat. | The car will be able to navigate some of the maze but may get stuck in places. It will take a long time. |

### 2.7.3 Dijkstra’s Algorithm Testing

Table 14 lays out the different tests that will be used to determine whether Dijkstra’s has been successful.

#### Table 14 – Tests for Dijkstra’s Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Test Data | Expected Result |
| 11 | Use the output data from Test 08 to test how efficient the algorithm is. | Test 08 output graph | The algorithm should be able to output quickly. |
| 12 | Use the output data from Test 09 to test how efficient the algorithm is. | Test 09 output graph | The algorithm should be able to output moderately quickly. |
| 13 | Use the output data from Test 10 to test the extreme side of the data input. If Test 10 does not produce a valid graph, a graph may be used from elsewhere. | Test 10 output graph, or a difficult graph from another source. | The algorithm should be able to output but may take a while. |

### 2.7.4 “Run All” Algorithm Testing

Table 15 lays out the different tests that will be used to determine whether the “Run all” algorithm has been successful.

#### Table 15 – Tests for the “Run All” Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Test Data | Expected Result |
| 14 | Tests 08 and 11 | The easy test mat | The algorithm should be able to run through the maze, produce the best path and run the best path. |
| 15 | Tests 09 and 12 | The medium test mat | The algorithm should be able to run through the maze, produce the best path and run the best path. |
| 16 | Tests 10 and 13 | The difficult test mat | The algorithm might be able to run through the maze, produce the best path and run the best path. If at least halve the maze is covered and then Dijkstra’s is run successfully, then the test is successful. |

### 2.7.5 Log File Testing

Table 16 lays out the different tests that will be used to determine whether the Log file handling has been successful.

#### Table 16 – Tests for the log file handling

| Test ID | Description | Test Data | Expected Result |
| --- | --- | --- | --- |
| 17 | The system will be booted up, some random inputs will be given, and then the system will be shut down. | Any sample menu. | Everything should run smoothly, and by the end a log file with time stamps should be stored in memory. |
| 18 | The system will be booted, some buttons will be pressed, and the program will be exited unexpectedly. | Any sample menu. | The system should record the error and exit safely. |
| 19 | An object will be serialised using Pickle and stored in a file.  The object will then be loaded back in and accessed. | Any sample menu.  Any sample objects. | The object should load correctly, in the same state as when it was serialised. |

### 2.7.6 Motor Control Testing

Table 17 lays out the different tests that will be used to determine whether the motor control has been successful.

#### Table 17 – Tests for the motor controls

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Description | Test Data | Expected Result |
| 20 | Move 1m | - | The car should move 1m +/- 10 cm |
| 21 | The car will turn 90 one way, then back to 0 before turning 90 the other way. | - | The car should be within 5 degrees of 90 degrees from the origin line. |
| 22 |  |  |  |

# 3 Implementation phase

## 3.1 Iteration 1

Iteration 1 of the solution will contain only the bare basics of the programs. This means that the code will mimic the flow charts described above. This phase will ignore efficiency and integration between the different systems. This iteration is a prototype – it will gauge how much I am following the user requirements. This is made from the components mentioned and decomposed in the design phase. Whilst this system will show some modularity, the components will not be fully integrated with eachother.

### 3.1.1 Dijkstra’s

Dijkstra’s algorithm requires vertexes to trace and calculate a path through the maze. These vertexes must have adjacent weights and labels linking a vertex to another.

In this iteration of the vertex class, I have used individual lists for each of the directions, prefilled with None type vales. This means that even if the list is left blank, the system will not report an error. Each of these lists are then added to a dictionary using the directions as a key.

**Validation:**

In the vertex class the values must be entered in pairs (list of 2).

class Vertex():

    def \_\_init\_\_(self, label, left = [None,None], right = [None,None], rear = [None, None], front = [None, None]): # lists stored as [label, weight]

        self.label = str(label)

        self.adjacencyDict = {"Left":left, "Right":right, "Rear":rear, "Front":front} # dictionary of the adjacent verticies

        self.totalweight = 0

        self.permanent = False

The other major component of Dijkstra’s algorithm is the graph used to store the vertices. In this iteration of the graph class, a list called network is defined which will store all the vertices of the graph. Since the vertices are added directly to the list, the objects in the graph can be iterated through, crucial for Dijkstra’s in the priority queue. The graph will also have an index used for deciding which character gets assigned to a vertex.

The graph will have an add vertex subroutine, allowing for the label index to be incremented correctly, and for the instantiation of the new vertex to be validated and consistent with the others in the network.

**Validation:**

The graph class uses a subroutine to add a vertex to ensure all the values entered into the vertex constructor method are correct. The label index is also incremented by 1 here rather than other programs manually entering.

class Graph():

    def \_\_init\_\_(self, end\_label):

        self.network = []

        self.label\_index = 65 # 65 in unicode is A

        self.end = end\_label

    def add\_vertex(self, left = [None,None], right = [None,None], rear = [None, None], front = [None, None]):

        Vert = Vertex(chr(self.label\_index), left, right, rear, front)

        self.network.append(Vert)

        self.label\_index += 1

The Dijkstra’s subroutine copies the current state of the network to a priority queue, and creates an empty list called permanent which will hold all the vertexes that have become permanent (no new shorter route exists to get to the vertex). For each vertex in the queue that is not the first vertex a weight of infinity is given, to enable the bubble sort to have only the vertexes with calculated weights to become permanent.

The main loop in the algorithm then sorts and dequeues the priority queue. This vertex will then become part of the permanent list so it can be referenced and found later. Each vertex has four possible directions for an arc to exist, which are all checked, and the vertex is updated. The algorithm will stop once the length of the queue is 0.

    def Dijkstras(self):

        Queue = self.network

        Permanent = []

        dirList = ["Left", "Right", "Rear", "Front"]

        for i in range(1, len(self.network)): # setting up the initial values of the distance from the start node.

            Queue[i].totalweight = math.inf

        while len(Queue) > 0: # the main loop which checks  all the vertices

            self.bubble(Queue)

            current\_vertex = Queue.pop(0) # dequeue

            current\_vertex.permanent = True

            Permanent.append(current\_vertex)

            for i in range(0,4): # iterates from 0 to 3, stops when at 4

                dirVertexLabel = current\_vertex.adjacencyDict[dirList[i]][0] # finds the vertex in the direction

                try:

                    if self.find\_vertex(Queue, dirVertexLabel).totalweight > current\_vertex.totalweight + current\_vertex.adjacencyDict[dirList[i]][1]:

                        self.find\_vertex(Queue, dirVertexLabel).totalweight = current\_vertex.totalweight + current\_vertex.adjacencyDict[dirList[i]][1]

                except Exception as e: print(e)

        # Output the final path to follow

        for item in Permanent:

            print("\n\n {}".format(item.label))

            for direction in dirList:

                print("{}: {}".format(direction, item.adjacencyDict[direction]))

            print(item.totalweight)

The find vertex subroutine is used to iterate through the queue and return a vertex with a unique label.

    def find\_vertex(self, Queue, label):

        for item in Queue:

            if item.label == label:

                return item # returns the vertex

        try:

            return Queue[len(Queue)-2]  #  returns the final value of the queue

        except:

            print(len(Queue))

            print(label)

            for thing in Queue:

                print(thing.label)

The bubble sort used in this algorithm is standard, just checks the specific attribute total weight when sorting.

    def bubble(self, queue):

            swaps = 1

            while swaps > 0:

                swaps = 0

                for i in range(0, len(queue)-1):

                    if queue[i].totalweight > queue[i+1].totalweight:

                        queue[i], queue[i+1] = queue[i+1], queue[i] # swaps the two values round

                        swaps += 1

            return queue # returns the sorted version of queue

|  |  |  |
| --- | --- | --- |
| Test ID | Validity of Result | Comments |
| 101 | Valid, but inefficient | The way the current subroutine interacts and passes values into the vertex class repeats data 3 times. This could be replaced by a subroutine in the vertex class used for updating the adjacency of an individual vertex.  The inputs also need to be simplified, as iterating through a list to reference different values in a dictionary is inefficient. |
| 102 | Valid | The loop does work and does return the correct path through a rudimentary maze with minimal vertices. However, the loop heavily depends on the try/except condition for updating the adjacent weights. This is due to the use of NoneType values in the vertex class. |
| 103 | invalid | The find vertex subroutine only iterates through the queue. This means if a value exists but not in the queue, an error will occur if the list is less than 1 in length. This can be changed by having a precondition of the length of the queue being greater than 1 and using a null vertex with no lengths instead of accessing a random vertex. |
| 104 | Valid | The algorithm works as intended. It may be slower with a larger number of items, but it shouldn’t matter as a max of 26 vertexes will be created. |

### 3.1.2 Car’s Menu

The cars menu will be the main way the user can interact with the car’s system. This means the code needs to be efficient.

In this first iteration I will be using a Menu class to hold the data needed to display individual menus. This class will hold the name of the menu, the titles to be display and the action that each titles executes when pressed. A subroutine will handle the printing of the menu to the screen, allowing for consistency.

class Menu: # used for the different submenus in the UI

    def \_\_init\_\_(self, name, titlesList:list, subroutines:list) -> None:

        self.name = name

        self.titles = titlesList

        self.subroutines = subroutines

    def draw(self, screen, pointer): # draws onto the ev3 screen

        screen.draw\_text(115, 5, self.name)

        screen.print("")

        for i in range(len(self.titles)):

            if pointer == i:

                screen.print(self.titles[i]+" <--")

            else:

                screen.print(self.titles[i])

Python does not have a “Main” subroutine that runs initially, so to give myself modularity, and the ability to widely catch errors in the program. I have called this subroutine runMain. This also handles the user inputs for navigation. The active Boolean determines when the runMain subroutine should stop, allowing me to stop it from anywhere in the program. Here I am using a sleep method to stop the algorithm from outputting too much data, causing the program to crash.

class Main: # will hold the main section of the program. useful for dropping in different main loops

    def \_\_init\_\_(self) -> None:

        self.active = True

    def runMain(self): # the main section of the code is here vvv

        Car = car()

        MainMenu = Menu("Main", ["Run meter", "Left Right", "ID Line", "Find Line"], [Car.RunM, Car.LeftRight, Car.IDLine, Car.Findline])

        currentMenu = MainMenu

        pointer = 0

        while self.active:

            Car.screen.clear()

            currentMenu.draw(Car.screen, pointer)

            if Button.DOWN in Car.buttons.pressed():

                pointer += 1

                time.sleep(0.3)

            if Button.UP in Car.buttons.pressed():

                pointer -= 1

                time.sleep(0.3)

            if Button.CENTER in Car.buttons.pressed():

                try: # will try to run the subroutine in the current menu

                    currentMenu.subroutines[pointer]()

                except:

                    pass

            time.sleep(0.1)

Here I define the car class. The constructor method contains the instantiation of the EV3 brick and assigning the aspects of the brick such as drivebase, colour sensor and screen to individual properties of the car. Using a class allows for modularity in the program and allows me to add subroutines to the car that can be accessed by the menu class.

class car:

    def \_\_init\_\_(self):

        self.ev3 = EV3Brick()

        self.screen = self.ev3.screen

        self.buttons = self.ev3.buttons

        self.motorA = Motor(Port.A)

        self.motorD = Motor(Port.D)

        self.driver = DriveBase(self.motorD,self.motorA,31,190) # wheels have diameter of 31mm and a drivebase width of 190mm

        self.colsense = ColorSensor(Port.S4)

        self.driver.settings(straight\_speed=200) # sets the cars speed to 200mm /s

I have coded some simple placeholder movement and sensor subroutines into the car class. These allow me to do some simple tests on the hardware and understand how the code communicates with the hardware. These are ONLY placeholders. The ID line subroutine identifies when a black object is below the colour sensor.

    def RunM(self):

        self.screen.clear()

        self.screen.print("Run is running")

        self.driver.straight(1000) # forwards 1000 mm

        time.sleep(1)

    def LeftRight(self):

        dirs = ["Right", "left"]

        for i in (0,1):

            self.screen.clear()

            print (dirs[i] + " is running")

            self.driver.turn(90-180\*i)

    def IDLine(self):

        self.screen.clear()

        self.screen.print("IDline is running")

        while self.colsense.color() != Color.BLACK:

            self.screen.print(self.colsense.color())

            time.sleep(0.001)

    def Findline(self):

        self.screen.clear()

        self.screen.print("Finding Line")

        time.sleep(1)

The final two lines of the code are the instantiation of the Main object and running the runMain subroutine.

MyMain = Main() # creates the main object

MyMain.runMain() # runs the main section of the code

### 3.1.3 Search Algorithm

The search algorithm will be used to map out the maze and convert it into a graph for the path finding algorithm to compute. Currently, this iteration of the search algorithm does not fully work, as it can only navigate a straight line before stopping.

I will only be showcasing the search algorithm, as the other sections have been shown and explained in other parts. This allows me to concentrate on the specific code for the search.

def Search(self):

        self.maze.add\_vertex() # creates the initial vertex

        while self.colsense.color() != Color.BLUE:

            time.sleep(0.1)

            dir\_offset = 0

            self.driver.drive(200,0) # sets the motors to drive forward

            if self.colsense.color() == Color.RED: # go straight ahead

                print("running")

Here, we define the search algorithm. The only parameter needed for this to work is the object, as everything else is attached to that object. The initial vertex is added before the main loop, as otherwise the program would either add a vertex every time it looped, or never add an initial vertex.

Since the ending point will be blue, the ending condition has been set to when the colour sensor detects blue underneath it. Every time the loop repeats, the motors are set to running at 200mm/s with 0 degrees of rotation. The direction offset variable will be used later when the car begins to turn and directions to be absolute not relative. Since the colour of the path will be red, if the colour sensor detects red, it will just continue.

            elif self.check\_line(-45,45): # within 90\* ahead

                pass

            elif self.check\_line(45,135): # within the 90\* to the right

                self.maze.add\_vertex(left=[chr(self.maze.label\_index), self.driver.distance])

                self.driver.reset()

                dir\_offset += 1 # increments the direction offset

            elif self.check\_line(-135, -45): #within 90\* to the left

                self.maze.add\_vertex(right=[chr(self.maze.label\_index), self.driver.distance])

                self.driver.reset()

                dir\_offset -= 1

            elif self.check\_line(135,-135): # within 90\* behind

                self.maze.add\_vertex()

This piece of code determines whether there is another path to the left or right, if the car has drifted or if it has hit a dead end. If the path is cut, then a vertex is created and added to the graph. The vertex is then updated with the information of the previous vertex before the car’s distance memory is reset.

**Validation:**

The code validates if there is a vertex in any direction, and adjusts to be able to navigate to that vertex.

Several errors were encountered and not all were fixed.

|  |  |  |
| --- | --- | --- |
| Error type | Description and Comment | Fixed |
| AttributeError: 'module' object has no attribute 'inf' | The math module does not exist in the code already on the EV3. This means that the value math.inf cannot be used for comparing the weights of each arc. | Use float(“inf”) to represent infinity. |
| The EV3 not turning to correct point | The EV3 does not yet turn to the correct point, hence it does not know if there is a corner or if it is a dead end, or if it has simply gone off track. | **Needs to be brought forward to next iteration.** |
| TypeError: unsupported types for \_\_sub\_\_: 'bound\_method', 'int' | Multiple data types are being input into a subroutine. This means that the calculations in checkline are not working. | Update code to use parenthesis instead of using the function. i.e. colour -> colour() |

|  |  |  |
| --- | --- | --- |
| Test ID | Valid Result? | Comments |
| 105 | Invalid | The car does not know which direction is which. It will rotate incorrectly, then start to search. This will lead to errors if used in the final program. |
| 106 | Invalid | The car cannot turn around and follow a maze, hence it cannot create vertexes to make a graph. |
| 107 | Invalid | The car does not navigate mazes with the desired result. The car can follow a straight line but will not work once it has left the line. |

### 3.1.4 Iteration Review

## 3.2 Iteration 2

### 3.2.1 Dijkstra’s

|  |
| --- |
| import math  class Vertex():      def \_\_init\_\_(self, label):          self.label = str(label)          # left, right, rear, front          self.adjacencyLabels = ["","","",""] # dictionary of the adjacent vertices          self.adjacencyWeights = [math.inf, math.inf, math.inf, math.inf] # has all the connected arc weights set to infinity          self.totalweight = math.inf          self.permanent = False          self.previousVertex = ""          self.previousDir = 0      def update\_adjacency(self, input = [[],[],[],[]]):          for i in range(0,4):              try:                  if len(input[i]) > 0:                      self.adjacencyLabels[i] = str(input[i][0])                      self.adjacencyWeights[i] = int(input[i][1])              except:                  print("one of your values isn't the correct type")    class Graph():      def \_\_init\_\_(self, end\_label):          self.network = []          self.label\_index = 65 # 65 in unicode is A, works as an offset          self.end = end\_label          self.nullVertex = Vertex("\*")      def add\_vertex(self, left = [], right = [], rear = [], front = []):          Vert = Vertex(chr(self.label\_index))          Vert.update\_adjacency([left,right,rear,front])          self.network.append(Vert)          self.label\_index += 1 # increments the label by 1 e.g A -> B      def Dijkstras(self):          Queue = self.network          Permanent = []            Queue[0].totalweight = 0 # sets the initial vertex weight to 0          end\_vertex = self.find\_vertex(Queue, self.end)          while end\_vertex in Queue: # the main loop which checks all the vertices              self.bubble(Queue) # sorts the queue                current\_vertex = Queue.pop(0) # similar to the dequeue in static languages              current\_vertex.permanent = True              Permanent.append(current\_vertex)              for i in range(0,4): # iterates from 0 to 3, stops when at 4                  dirVertex = self.find\_vertex(Queue,current\_vertex.adjacencyLabels[i]) # finds the vertex in the direction                  if dirVertex.totalweight > current\_vertex.totalweight + current\_vertex.adjacencyWeights[i]:                          dirVertex.totalweight = current\_vertex.totalweight + current\_vertex.adjacencyWeights[i]                          dirVertex.previousVertex = current\_vertex.label                          dirVertex.previousDir = i          # Output the final path to follow          print(self.findPath(Permanent, self.end))      def findPath(self, final\_list, curlabel, output = ""):          dirList = ["Left", "Right", "Rear", "Front"]          curVertex = self.find\_vertex(final\_list, curlabel)          output += "\n\nlabel: {},\nweight: {},\ndir: {}".format(curVertex.label, curVertex.totalweight, dirList[curVertex.previousDir])          if curVertex.label == "A":              return output          else:              return self.findPath(final\_list,curVertex.previousVertex, output)          def find\_vertex(self, Queue, label):          if len(Queue)>0:              for item in Queue:                  if item.label == label:                      return item # returns the vertex          return self.nullVertex        def bubble(self, queue): # sorts the vertices by weight from start              swaps = 1              while swaps > 0:                  swaps = 0                  for i in range(0, len(queue)-1):                      if queue[i].totalweight > queue[i+1].totalweight:                          queue[i], queue[i+1] = queue[i+1], queue[i] # swaps the two values round                          swaps += 1              return queue # returns the sorted version of queue  myNetwork = Graph("E")  myNetwork.add\_vertex(right=["B", 4])  myNetwork.add\_vertex(left=["A", 4], front=["C", 4])  myNetwork.add\_vertex(right=["E", 4], rear=["B", 4], front=["D", 4])  myNetwork.add\_vertex(rear=["D", 4])  myNetwork.add\_vertex(left=["C", 4])  myNetwork.Dijkstras()  print("reached the end") |

## 3.3 Iteration 3

# Appendix

## Easy Test Mat

The below test mat will be scales up. This means each edge will be roughly 50 in length.



## Medium Test Mat

The Below test mat will be scaled up. The Length AB should be 50 cm.



## Difficult Test Mat

## Motor Control Test Mat

The motor control mat will need to be scaled up so that the length indicated is exactly one meter.



## Test Sheet Print Out

This sheet will be printed out on A4 to record the results of the tests. Table

Description automatically generated