Contents

[Analysis 3](#_Toc123027320)

[Problem identification 3](#_Toc123027321)

[***Table 1 - Identification of stakeholders*** 3](#_Toc123027322)

[***Table 2 - Computational Methods*** 4](#_Toc123027323)

[Research 6](#_Toc123027324)

[Existing similar solution 6](#_Toc123027325)

[Features of the proposed solution 9](#_Toc123027326)

[A brief explanation of the LEGO MINDSTORMS EV3 unit 9](#_Toc123027327)

[Requirements 11](#_Toc123027328)

[Software requirements 11](#_Toc123027329)

[Hardware Requirements 11](#_Toc123027330)

[Stakeholder requirements 11](#_Toc123027331)

[Success Criteria 12](#_Toc123027332)

[Design 13](#_Toc123027333)

[Overview of system 13](#_Toc123027334)

[**System Layer Diagram** 13](#_Toc123027335)

[**System Navigation Chart** 14](#_Toc123027336)

[Algorithms and further breakdown 15](#_Toc123027337)

[Car State Diagram 15](#_Toc123027338)

[Dijkstra’s Flow Chart 16](#_Toc123027339)

[Search Algorithm Flow Chart 17](#_Toc123027340)

[User Interface 18](#_Toc123027341)

[**Desktop UI** 18](#_Toc123027342)

[Pseudo Code 19](#_Toc123027343)

[Menus 19](#_Toc123027344)

[Search Algorithm 19](#_Toc123027345)

[Dijkstra’s 19](#_Toc123027346)

[Log File Layout 19](#_Toc123027347)

[Data Dictionary 19](#_Toc123027348)

[Improvements/actions needed 19](#_Toc123027349)

# Analysis

## 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 - 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 - 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 |

## Research

### Existing similar solution

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

Goal of solution:

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

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.

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

***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 |

**A sample of code from the solution:**

Graphical user interface, application, table, Excel

Description automatically generated

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

Goal of solutions:

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

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.

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 |

Diagram

Description automatically generatedImages of the solution:

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);   
}

### Features of the proposed solution

#### 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 |

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).

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 panels to develop the algorithm and work out any flaws.

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.

### 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 |

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.

#### 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.

## Requirements

### 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

### 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

### Stakeholder requirements

#### 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 |

#### 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 |

#### 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. |

## 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. |

# Design

## 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.

### **System Layer Diagram**

**Chart, diagram

Description automatically generated**The above diagram is a visualisation of the flow of data, with each layer of the diagram representing different sub-systems within the whole system. 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.

### **Diagram Description automatically generatedSystem Navigation Chart**

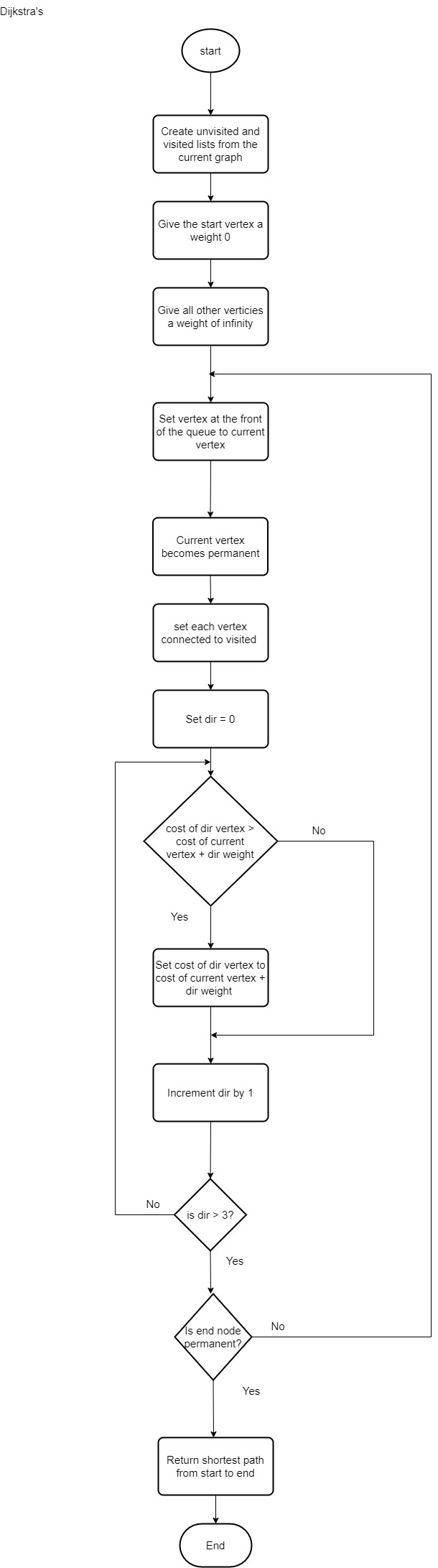
Key: Menu, Subroutine, Misc.

## Algorithms and further breakdown

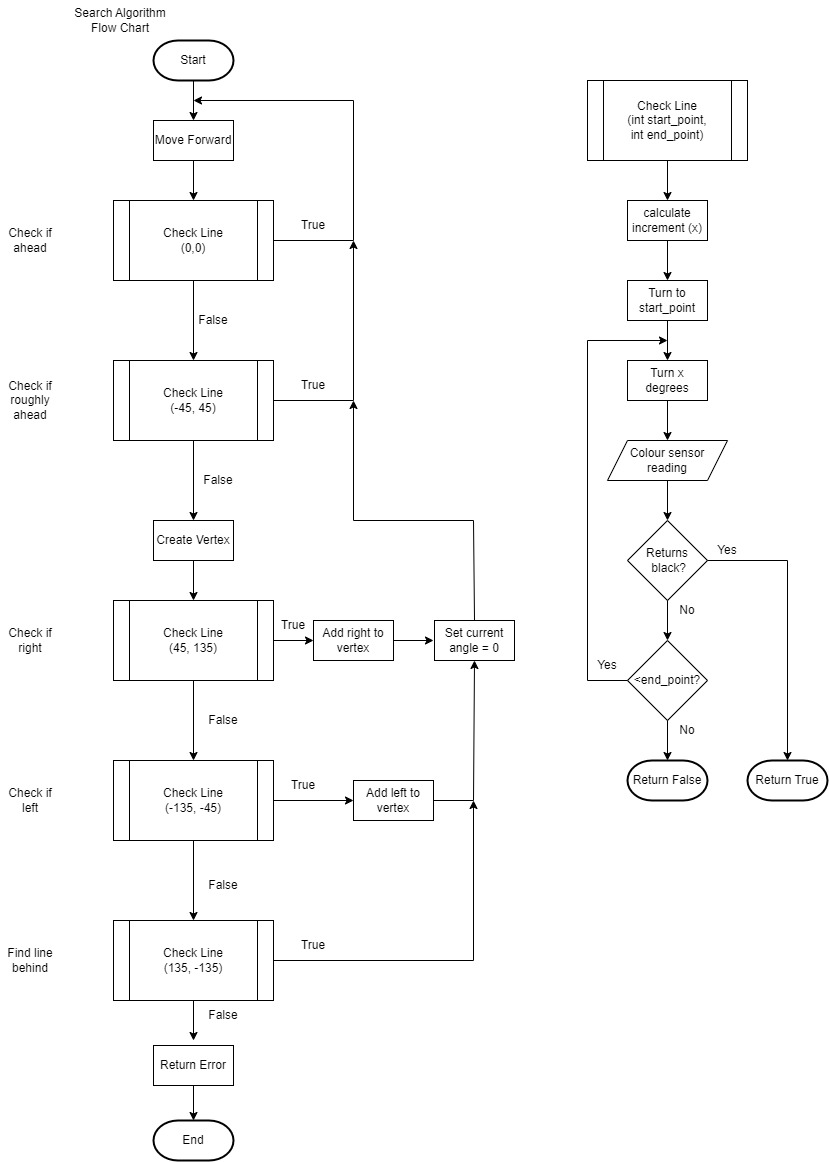
### Car State Diagram



### Dijkstra’s Flow Chart



### Search Algorithm Flow Chart



## User Interface

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

### **Car’s UI**

**Diagram

Description automatically generated**

### **Desktop UI**

Table

Description automatically generated with low confidence

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 above diagram is useful as it means I can visualise the desktop experience.

## Pseudo Code

### 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.

|  |
| --- |
| #!/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) |

### 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 |

### Dijkstra’s

Here 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 be made.

|  |
| --- |
| 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 |

## 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. They may look something like this, although future testing may require a slight change to how they read:

|  |
| --- |
| [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 |

# Diagram captions

System Navigation chart:

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 aspects 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.

Car state Diagram:

This diagram 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.

Dijkstra’s flow chart:

This flow chart is an abstraction of the widely used Dijkstra’s algorithm used for network traversal. The structure of the flow chart is done in such a way that allows for computational methods to be effective. 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.

Search algorithm flow chart:

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.

Car UI:

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.

Desktop UI:

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.

# Improvements/actions needed

Unit testing (testing a screen, a module, a class), different test strategies, integration testing, identify test data, give each test a reference, white box testing (unit testing, algorithms, developer, state data input used for testing), black box testing (post development, extreme values that are on the boundaries of the range, values that are outside the range, values that are not of the correct data type)