Intelligent Robotics Coursework

Implementation and Comparative Analysis of Bio-Inspired Algorithms for Mars Rover Navigation

By Ossian Davies, Muhammad Jaffery and Omar Riyaz

June 7, 2024

Number Pages: 26
Word Count: 2532

Declaration of Authorship

We, Ossian Davies, Muhammad Jaffery, and Omar Riyaz, declare that this thesis titled, 'The Mars Rover Coursework,' and the work presented in it are our own. We confirm that this work submitted for assessment is ours and is expressed in our own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Ossian, Muhammad, and Omar

Dated: 29/11/2023

Table of Contents

List of Figures

List of Tables

Abbreviations

AI	**A**rtificial **I**ntelligence
BBR	**B**ehaviour **B**ased **R**obotics
ER	**E**volutionary **R**obotics
MLP	**M**ulti **L**ayer **P**erceptron
GA	**G**enetic **A**lgorithm

Chapter 1

Introduction

1.1 Overview

The search for and exploration of planets like Mars presents many kinds of challenges leading to the need for innovative robotics and AI solutions. The development of intelligent rover controllers that can navigate difficult ground and adjust to its changing environment is an important part of the exploration of mars. In order to address the challenge of Mars rover's navigation, this report centers around employing and analyzing two bio-inspired strategies:

- Behavior Based Robotics: BBR is a type of robotics approach where a robot's behaviors result from the combination of a few basic, predetermined principles. A behaviour-based robot (BBR) will react and correct its actions directly with the information gained from the robot's sensors [Yeon et al., 2015]. It uses modular behaviors to build robotic systems that are sensitive and adaptive, like obstacle avoidance and line following.
- Evolutionary Robotics: ER is the method of using natural selection inspired evolutionary algorithms to develop robot controls. Future generations of controllers improve on this, enabling robots to adjust their behavior to specific scenarios. It is inspired by the Darwinian principle of selective reproduction of the fittest [Nolfi and Floreano, 2000].

The rover's sensing and perception system must be able to sense the surrounding terrain, convert the sensed data into a representation of the local environment, and correlate this local representation with the global map, thereby determining the rover's position [Miller et al., 1989]. Due to obstacles and the need for flexible decision making, the navigation problem for the rover is difficult. While the ER approach uses evolutionary algorithms to evolve controllers over multiple generations, the BBR approach builds a controller based on specific behaviors, such as obstacle avoidance and line following.

Fig. 1.1 shows the two routes and the expected behaviour of the robot. Route A and Route B are the two routes that lead to the reward zone. The state of the surface beacon affects the route that the Mars rover chooses to travel. The Mars rover's only source of direction is the state of this beacon because there could be dust storms nearby. The Mars rover should choose Route A if a satellite activates the beacon, turning on the lights or it shall take Route B if a satellite deactivates the surface beacon, turning off the lights.

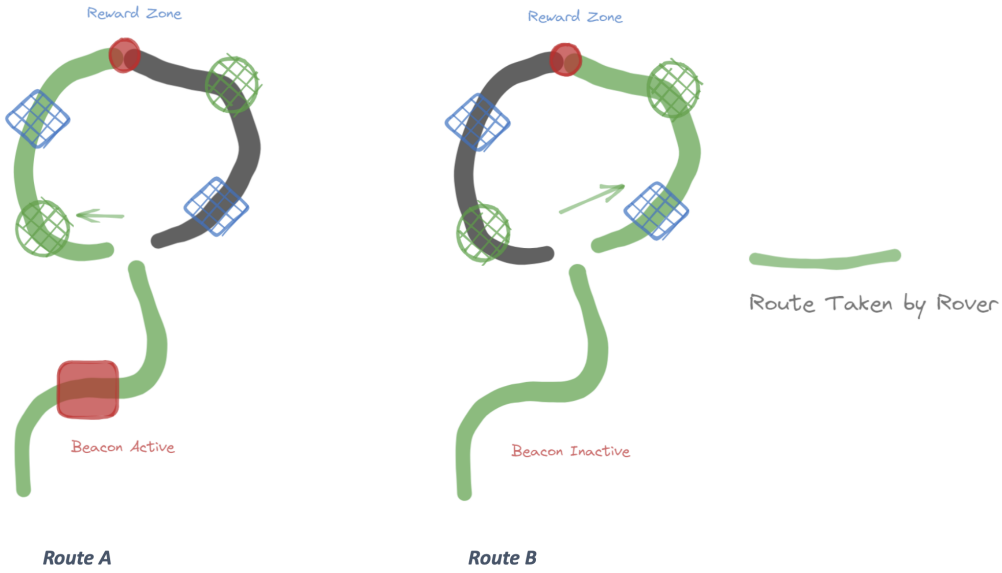

Figure 1.1: Mars Rover possible routes

1.2 Aims and Objectives

The reports main aim is to use Webots to replicate a Mars rover's environment and to then develop and analyze controllers for it. The selected Mars rover, which is based on the e-puck robot, has to go along a predetermined path to arrive at a specific reward zone. The rovers's decision-making process is dependant on the status of a surface beacon, activated by a satellite to indicate the optimal route. The primary objective in this report is to investigate the effectiveness of these bio-inspired algorithms for Mars rover navigation. We aim to assess controller's speed, adaptability, and success rate in reaching the reward zone by putting them into practice using BBR and ER. We will evaluate the findings, make comparisons between the two methods, while gaining understanding of the advantages and disadvantages of each. The complete list of objectives is given in Table 1.1 below.

ID	**Objective**
OBJ-1	Create a BBR controller with distinct behaviors for the Mars rover.
OBJ-2	To develop an ER controller that can change from generation to generation, modify evolutionary algorithms.
OBJ-3	Incorporate the e-puck robot model as the Mars rover, while staying within the given parameters.
OBJ-4	For both BBR and ER controllers, conduct simulations and note the average time it takes to reach the reward zone across three runs.
OBJ-5	Examine the BBR and ER controllers decision making processes, adaptability, and performance.

Table 1.1: Objectives of the report

Chapter 2

Behaviour Based Robotics Approach

2.1 Design and Implementation

A class called Controller has methods for initialization, sensing, computing, and actuating the code. This encourages modularity and improves readability of the code. To decide how to manoeuvre the robot, the approach analyses sensor data. The logic makes a distinction between the ground sensors on the left and centre as points of reference. To modify the robot's speed, it considers factors like the direction of light sensors and the information from proximity sensors. Conditional statements, the foundation of the control logic, consider several variables including the orientation of the light sensors, the distance from obstacles, and deviance from the intended route.

2.2 Coding

We start with disabling the robots left ground sensor. This is done initially to accept the case of the light not being active. As you can see in Figure 1 the robot starts with detecting the North-West light sensor, which is shown in the robot view as ls7, as seen in Figure 2.2.

```
#If the NW light sensor detects light and the left
if(self.inputs[-1] < 0.5 and self.inputs[0] == 0):
    self.left_ir.enable(self.time_step)
    self.right_ir.enable(0)
```

Figure 2.1: If Statement to check if the light is on

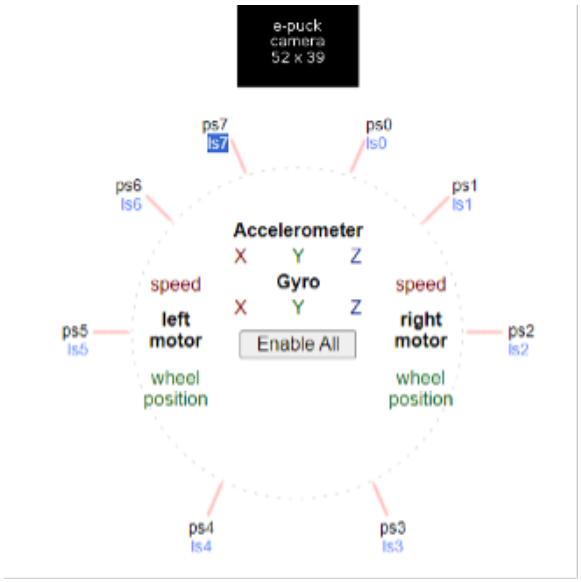

Figure 2.2: The Robot View

If the light is OFF, the left ground sensor stays disabled and uses the centre sensor to navigate forward and left, as shown in Figure 2.3 below.

```
#Use Centre Sensor as left
if(self.inputs[0] == 0):
    print(str(self.inputs[1] > 0.01)+ ":" + str(self.inputs[0] < 0.01) + ":" + str(np.max(self.inputsPrevious[1:5]) < 0.01) + ":" + str(np.max(self.inputs[0:2]) > 0.7))
    print(self.inputs[0]) # 0.028
    print(self.inputs[1]) # 0.04
    #4 0.0285
    if(self.inputs[1] > 0.01 and self.inputs[0] < 0.01 and np.max(self.inputsPrevious[1:5]) < 0.01 and np.max(self.inputs[1:3]) > 0.7):
        for i in range(8):
            if(not(i==2 or i==5)):
                self.proximity_sensors[i].enable(0)
```

Figure 2.3: If statement to use the centre sensor as left

Figure 2.4 shows the if statement for the robot to follow the line if it does not see an object in its North and North-East sensors. The threshold we calculated after multiple experiments for the robot to stay on the line came to 0.76. We found that if the threshold is closer to 1 that means the robot is off the line.

```
if(np.max(self.inputs[3:9]) < 0.03):
    #If off the line
    if(np.min(self.inputs[1:3]) > 0.76):
        self.velocity_left = 0.3
        self.velocity_right = -0.1
    #Left Corner
    elif(round(self.inputs[2],2) < round(self.inputs[1],2) ):
        self.velocity_left = -1*(self.inputs[2]-1) * 1.6
        self.velocity_right = -1*(self.inputs[1]-1) * 0.6
    #Right Corner
    elif(round(self.inputs[1],2)  < round(self.inputs[2],2) ):
        self.velocity_left = -1*(self.inputs[2]-1) * 0.6
        self.velocity_right = -1*(self.inputs[1]-1) * 1.6
    #Centre
    else:
        self.velocity_left = -1*(self.inputs[1]-1)
        self.velocity_right =  -1*(self.inputs[2]-1)
```

Figure 2.4: If statement for following the line

If the robot encounters an obstacle we use np.argmax(self.inputs[3:9]), as shown in Figure 2.5, to get the sensor which has the highest proximity value. From there we use if else statements to move the robot around the opposite direction of highest value proximity sensor.

```
else:
    direction = np.argmax(self.inputs[3:9])
    #Object is north
    if(direction == 0 or direction == 5):
        self.velocity_left = -1
        self.velocity_right = 1
    #Met object north west
    elif(direction == 1):
        if(np.max(self.inputs[1:3]) < 0.305):
            self.velocity_left = -1*(self.inputs[1]-1)
            self.velocity_right =  -1*(self.inputs[2]-1)
        else:
            self.velocity_left = 0.1
            self.velocity_right = 0.5
    # #Object East
    elif(direction == 2):
        if(np.max(self.inputs[1:3]) < 0.3055):
            self.velocity_left = -1
            self.velocity_right = 1
        else:
            self.velocity_left = 1
            self.velocity_right = 0.6
    else:
        self.velocity_left = 0.8
        self.velocity_right = 0.4
```

Figure 2.5: If statement for following the line

Figure 2.6 shows that we have the else if statement for the robot to check if it has reached the reward zone or not. The e-puck checks its East and West proximity sensor inside the reward zone and turns the other sensor off. Once the robot sees that there is an obstacle around it the robot stops.

```
else:
    if((self.inputs[5] > 0.03 or self.inputs[6]> 0.03) and np.max(self.inputs[3:5]) == 0 and np.max(self.inputs[7:9]) == 0 ):
        exit(1)
```

Figure 2.6: Statement to check if the reward zone has been reached

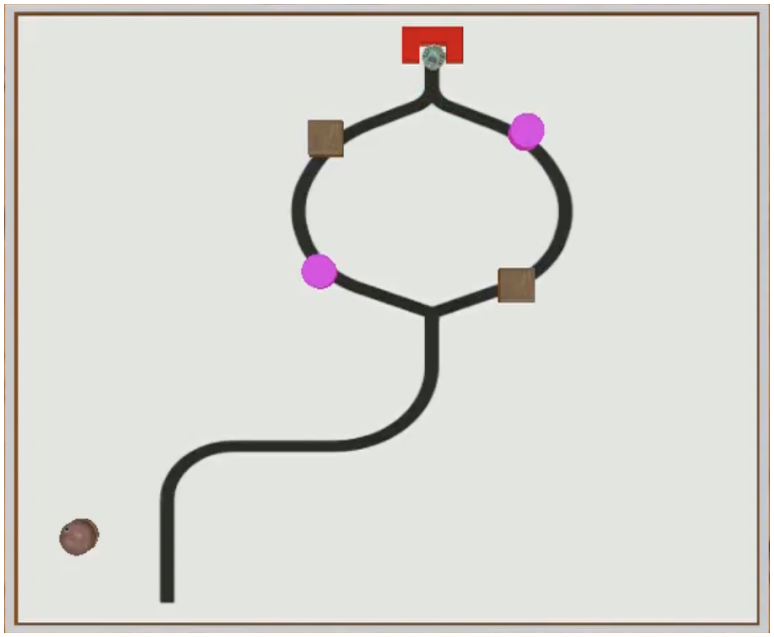

Figure 2.7: E-puck inside the reward zone

If the light is ON, the right ground sensor stays disabled and use the centre senor to navigate forward and right.

```
##Centre Sensor as Right
else:
    print(self.inputs[8])
    print(str(self.inputs[0] > 0.03)+ ":" + str(self.inputs[3] < 0.03) + ":" + str(np.max(self.inputsPrevious[3:8]) < 0.03) + ":" + str(np.max(self.inputs[0:2]) > 0.7))
    if((self.inputs[0] > 0.03  and self.inputs[3] < 0.03) and np.max(self.inputsPrevious[3:8]) < 0.03 and np.max(self.inputs[0:2]) > 0.7):
        for i in range(8):
            if(not(i==2 or i==5)):
                self.proximity_sensors[i].enable(0)
```

Figure 2.8: Else statement to use centre sensor as right.

Below in Figure 2.9, it shows the if statement for the robot to follow the line when it does not see an object in its North and North-West sensors. You see in Figure 2.9, the self.input[] has different values compared to Figure 2.4. This is due to us disabling the appropriate sensor during each task. As shown, the line following function uses the sensors values inverted to provide the robot with an accurate representation of where it is on the line. This allows the robot to stay well within the line at a high speed.

```
#Object not met north or north west
if(np.max(self.inputs[3:9]) < 0.03):
    #If off the line
    if(np.min(self.inputs[0:2]) > 0.76):
        self.velocity_left = -0.1
        self.velocity_right = 0.3
    #Left Corner
    elif(round(self.inputs[1],2) < round(self.inputs[0],2) ):
        self.velocity_left = -1*(self.inputs[1]-1) * 1.4
        self.velocity_right = -1*(self.inputs[0]-1)
    #Right Corner
    elif(round(self.inputs[0],2)  < round(self.inputs[1],2) ):
        self.velocity_left = -1*(self.inputs[1]-1)
        self.velocity_right = -1*(self.inputs[0]-1)* 1.4
    #Centre
    else:
        self.velocity_left = -1*(self.inputs[0]-1)
        self.velocity_right =  -1*(self.inputs[1]-1)
```

Figure 2.9: If statement for following the line.

In Figure 2.10, the implementation is very similar to one in Figure 2.5 and 2.6, except for the sensor input values.

```
else:
    if((self.inputs[5] > 0.03 or self.inputs[6]> 0.03) and np.max(self.inputs[3:5]) == 0 and np.max(self.inputs[7:9]) == 0 ):
        exit(1)
    else:
        direction = np.argmax(self.inputs[3:9])
        #Object is north
        if(direction == 0 or direction == 5):
            self.velocity_left = 1
            self.velocity_right = -1
        #Met object north west
        elif(direction == 4):
            if(np.max(self.inputs[0:2]) < 0.305):
                self.velocity_left = 1
                self.velocity_right = -1
            else:
                self.velocity_left = 0.5
                self.velocity_right = 0.1
        # #Object East
        elif(direction == 3):
            if(np.max(self.inputs[0:2]) < 0.3055):
                self.velocity_left = 1
                self.velocity_right = -1
            else:
                self.velocity_left = 0.6
                self.velocity_right = 1
        else:
            self.velocity_left = 0.4
            self.velocity_right = 0.8
```

Figure 2.10: Else statement to check for reward zone and object.

2.3 Results and analysis of BBR

Our BBR approached achieved the goal of reaching the endzone and determining the correct path based on the beacon as seen in the videos. The robot followed the line well as the graph below displays, where each spike of the centre ground sensor is where the robot was off the line.

2.4 Centre Ground Sensor Route A and B

The robot successful avoided the obstacles and remained on the line after avoiding the obstacle. This is displayed in the graph below where the spike represents where the robot met the obstacle.

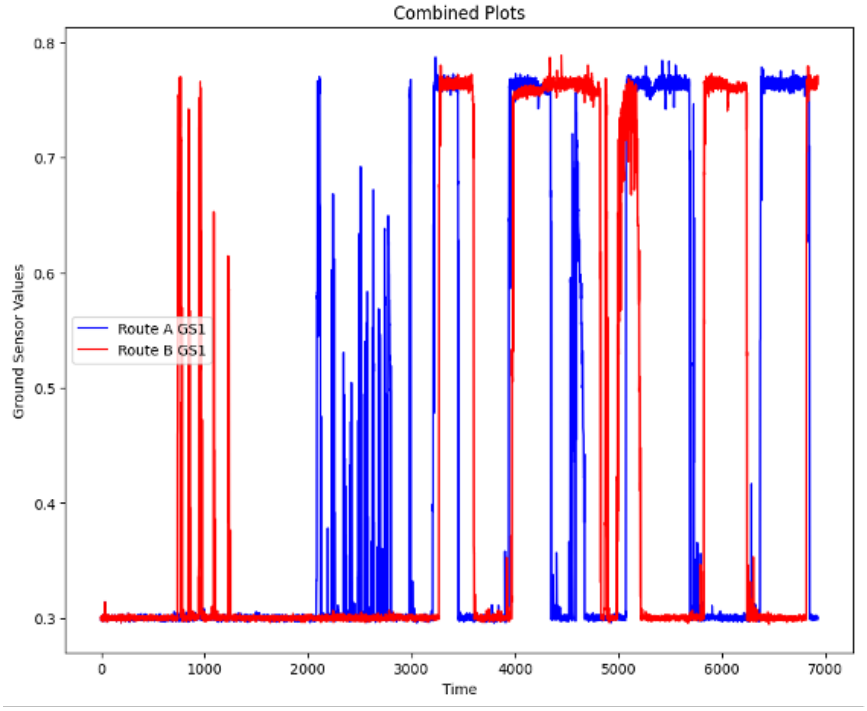

Figure 2.11: Centre Ground Sensor Route A and B:

2.5 Proximity Sensors Average Route A

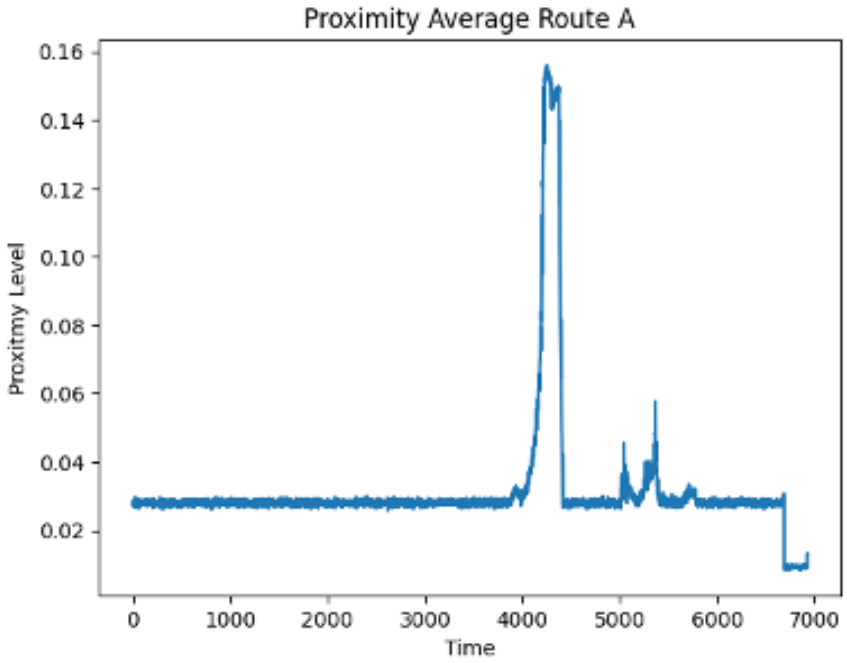

Figure 2.12: Proximity Sensors Average Route A

2.6 Proximity Sensors Average Route B

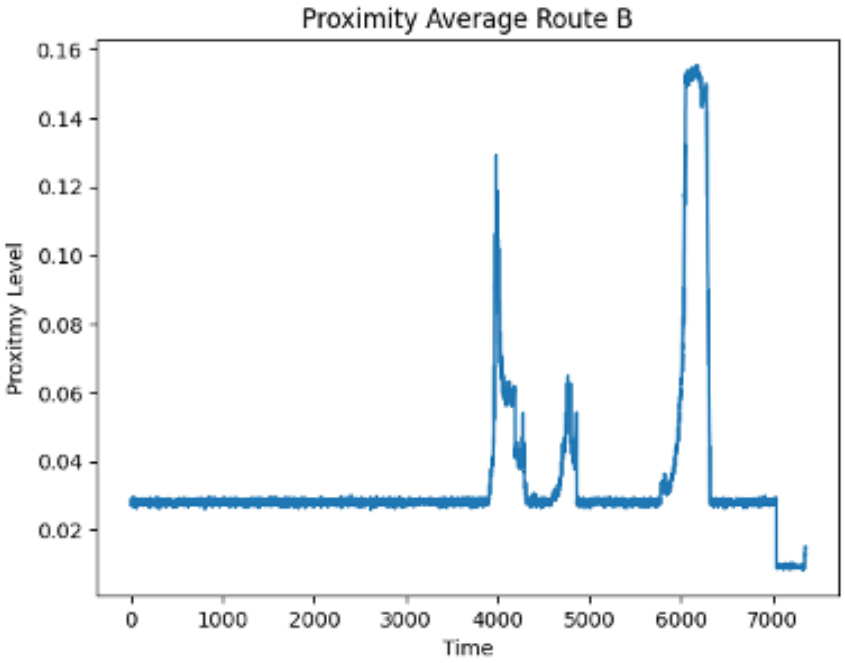

Figure 2.13: Proximity Sensors Average Route B

2.7 Combined Velocity Route A

The robot completed the run in the allotted time. As Graph 2.14 below displays the robot kept and efficient and fast speed even when avoiding the obstacles.

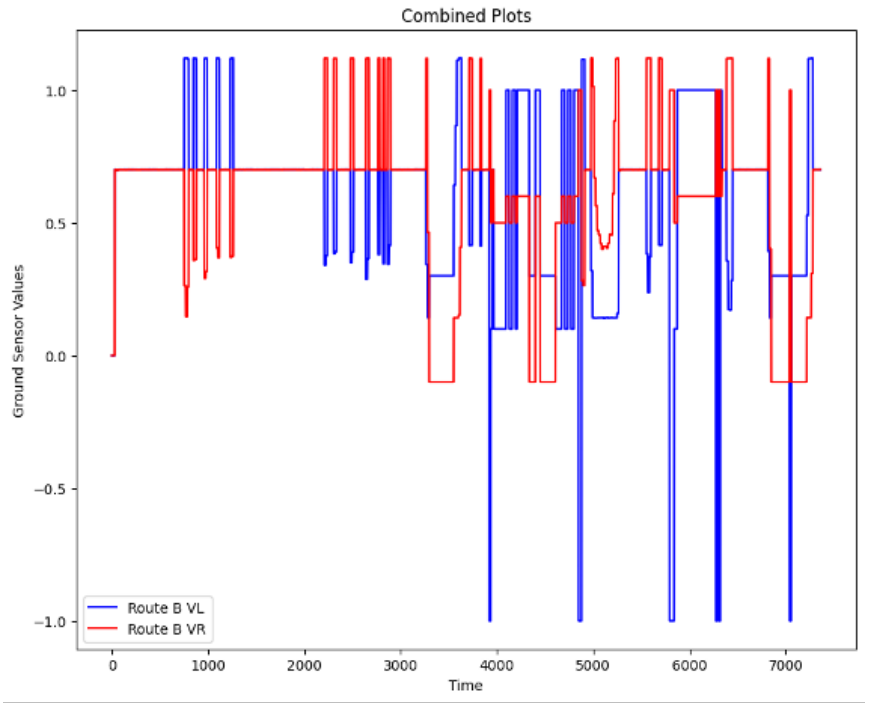

Figure 2.14: Combined Velocity Route B

2.8 Combined Velocity Route B

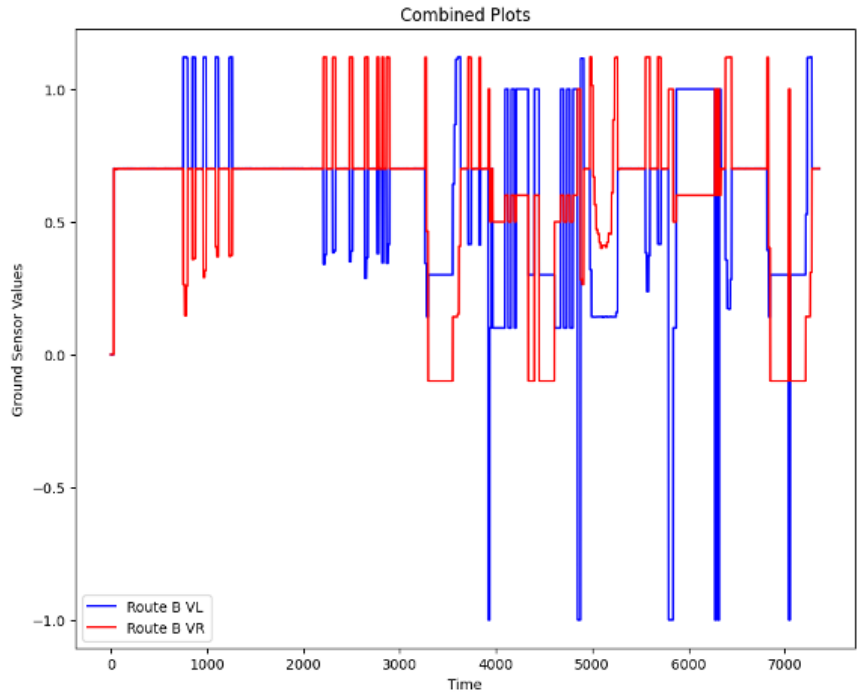

Figure 2.15: Combined Velocity Route B

Chapter 3

Evolutionary Robotics Approach

3.1 Design and Implementation

The implementation design for ER is reusing the code from Lab 4, provided on canvas. We have developed an artificial neural network called a Multi-Layer Perceptron (MLP) that will "learn" how to control the speed of the robot's wheel through evolution. Therefore, to generate accurate values for motor commands, the network's weights must be adjusted. The evolutionary method that was selected for our project was a Genetic Algorithm (GA), which was put into practice.

We start by defining the three GA parameters in "superviosrGA-ER.py" as shown in Figure 3.1.

```
### DEFINE here the 3 GA Parameters:
self.num_generations = 200
self.num_population = 6
self.num_elite = 3
```

Figure 3.1: Defining the 3 GA parameters.

Then we chose the following values for our MLP. This was decided using trial and error until the most optimal solution was reached.

Next, we create our fitness function in epuck_python – ER". We break down our fitness function into 4 performance metrices:

1. Forward and Line fitness function,
2. Endzone fitness function.
3. Collision fitness function.

4. Spinning fitness function.

In the method below in Figure 3.2 the left, centre, and right sensor input values are entered into the method. Based on the relationship between the left and right wheel velocities and the sensor readings, there are three instances within this scenario.

```
def calculate_forward_and_line_fitness(self):
    left_sensor=self.inputs[0]
    center_sensor=self.inputs[1]
    right_sensor=self.inputs[2]
    if(self.velocity_right > 0  and self.velocity_left > 0 ):
        if(self.velocity_right>self.velocity_left and (np.max(self.inputs[0:2]) < 0.5 and self.inputs[2] >= 0.5)):
            return (1-self.inputs[1])*1-(self.velocity_right - self.velocity_left)
        elif(self.velocity_right==self.velocity_left and (np.max(self.inputs[0:3]) < 0.5)):
            return (1-self.inputs[1])*self.velocity_right
        elif(self.velocity_right<self.velocity_left and (np.max(self.inputs[1:3]) < 0.5 and self.inputs[0] >= 0.5)):
            return (1-self.inputs[1])*1-(self.velocity_left - self.velocity_right)
        else:
            return 0
    else:
        return 0
```

Figure 3.2: Forward and Line fitness function 3.2.

The first step of the method below in Figure 3.3 is to retrieve the robot's prior translation values from the attribute. Next, it uses a list comprehension to round each translation value to the closest integer. Using the equality operator, the procedure determines whether the rounded translation values and the values given in ENDZONETRANS match: if ENDZONETRANS == rounded_translation.

```
def calculate_endzone_fitness(self):
    translation_values = self.previousPosition
    rounded_translation = [round(value) for value in translation_values]
    ENDZONETRANS = [0.1,0.9,0]
    if(ENDZONETRANS==rounded_translation):
        return 1
    else:
        return 0
```

Figure 3.3: Endzone fitness function.

The maximum value between two distinct proximity sensor readings—self.inputs[3] and self.inputs[10]—is first determined by the method shown in Figure 3.4. The method returns a fitness value of 0, meaning that the robot is penalised for being too close to an obstacle, if the maximum proximity value (max_prox) is greater than 0.1 (indicating that an obstacle is too close). The method uses the formula -1 * ((max_prox * 10) - 1) to determine a fitness value (coll_fit) if the maximum proximity value is less than or equal to 0.1. The robot is rewarded for keeping a safe distance from obstacles because this formula assigns higher fitness values for lower proximity values.

```
def calculate_collisonFitness(self):
    max_prox = np.max([self.inputs[3], self.inputs[10]])
    if(max_prox > 0.1):
        return 0
    else:
        coll_fit = -1* ((max_prox*10)-1)
        return coll_fit
```

Figure 3.4: Collision fitness function.

The method first verifies that the robot's left and right velocities are both positive. The robot is guaranteed to be moving forward by this condition. It determines whether the difference between them (self.velocity_left - self.velocity_right) is greater than 0.3 if self.velocity_left is greater than self.velocity_right. If so, the method returns 0 and the robot is penalised for spinning too much. If not, it yields 1. It determines whether the difference (self.velocity_right - self.velocity_left) is greater than 0.3 if self.velocity_left is less than self.velocity_right. If so, the method returns 0 once more, signifying that overspinning has a cost. If not, it yields 1. It returns 1 if self.velocity_left and self.velocity_right are equal. This suggests that the robot receives a positive fitness if its left and right velocities are equal, and it is not deemed to be spinning excessively. The method returns 0 if the initial condition (self.velocity_left is greater than 0 and self.velocity_right is greater than 0) is not met, which indicates that the robot is not moving forward.

```
def calculate_spinningFitness(self):
    if(self.velocity_right > 0  and self.velocity_left > 0 ):
        if(self.velocity_left > self.velocity_right):
            if(self.velocity_left-self.velocity_right > 0.3):
                return 0
            else:
                return 1
        elif(self.velocity_left < self.velocity_right):
            if(self.velocity_right-self.velocity_left > 0.3):
                return 0
            else:
                return 1
        else:
            return 1

    else:
        return 0
```

Figure 3.5: Spinning fitness function.

Establishing weights for each fitness component is the first step in the process. The relative significance of every element in the overall fitness computation is established by these weights. A weighted value is obtained by multiplying each fitness component by its corresponding weight. The weighted sum divided by the total weight yields the combined fitness. This normalisation process makes sure that, regardless of the weights selected, the total fitness at the end is within a constant range.

```python
def calculate_combinedFitness(self,endzoneFitness,lineFitness,avoidCollisionFitness,spinningFitness):
    weights = {
        'line': 0.9,
        'avoidCollision': 0.2,
        'spinning': 0.6,
        'endzone': 0.1
    }
    # Multiply each fitness component by its respective weight
    weighted_endzone = endzoneFitness * weights['endzone']
    weighted_line = lineFitness * weights['line']
    weighted_avoidCollision = avoidCollisionFitness * weights['avoidCollision']
    weighted_spinning = spinningFitness * weights['spinning']

    # Calculate the weighted sum and divide by the sum of weights
    weighted_sum = (
        weighted_line + weighted_avoidCollision + weighted_spinning
    )
    total_weight = sum(weights.values())
    combined_fitness = weighted_sum / total_weight

    return combined_fitness
```

Figure 3.6: Combined fitness function.

To run the optimisation of our GA, we changed the 'run_optimization()' function from lab 4 [Vargas, 2021] to rest the robots physics according to our world. We have also edited how the best genomes are saved. The lab 4 approach collects the last fitness after each trial, this value is then saved appended and then after the population has ran the mean fitness it taken. In our method, we take the mean value of the fitness over the run, then the genome with the highest mean fitness is saved.

3.2 Results and Analysis of ER

Both our runs did not achieve the goal. The first run only reached the pink cylinder and the second looped around the map. For the first run the graphs below demonstrate the relationship between the combined fitness and the ground sensors.

3.3 Run 1

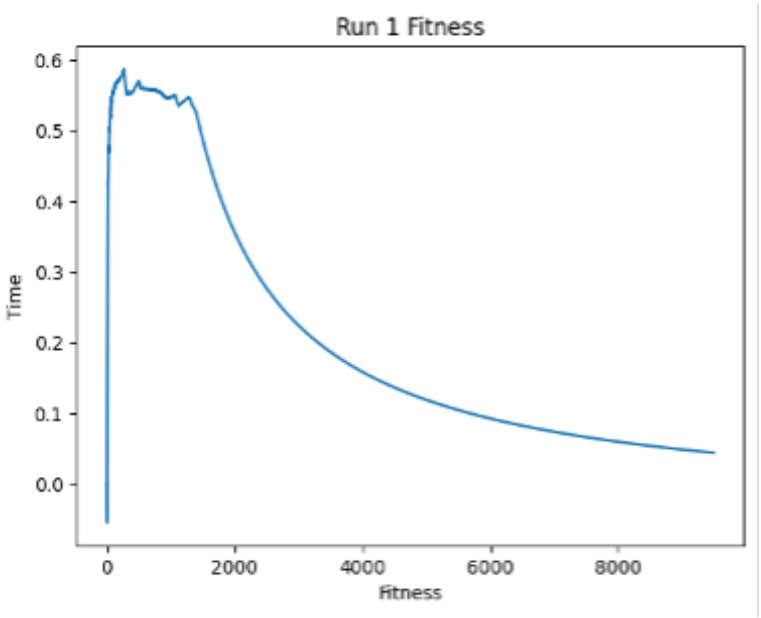

Figure 3.7: Run 1 Fitness

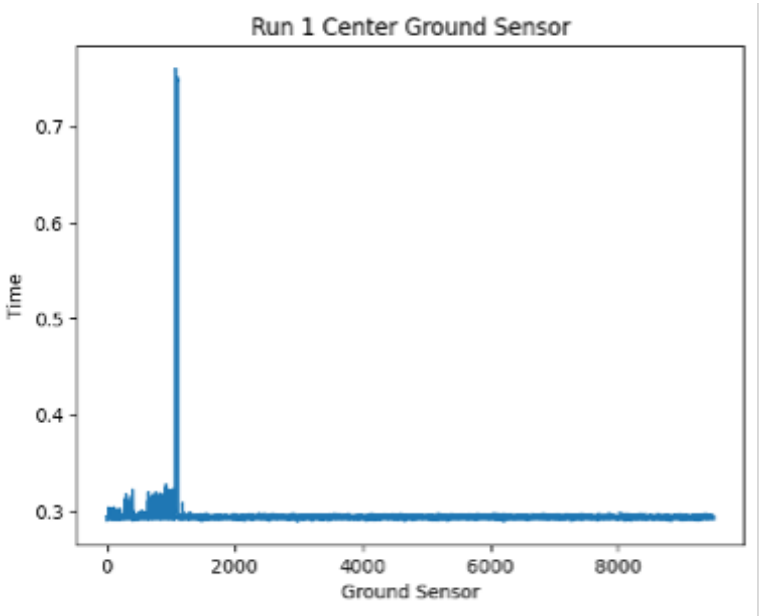

Figure 3.8: Run 1 Center Ground Sensor

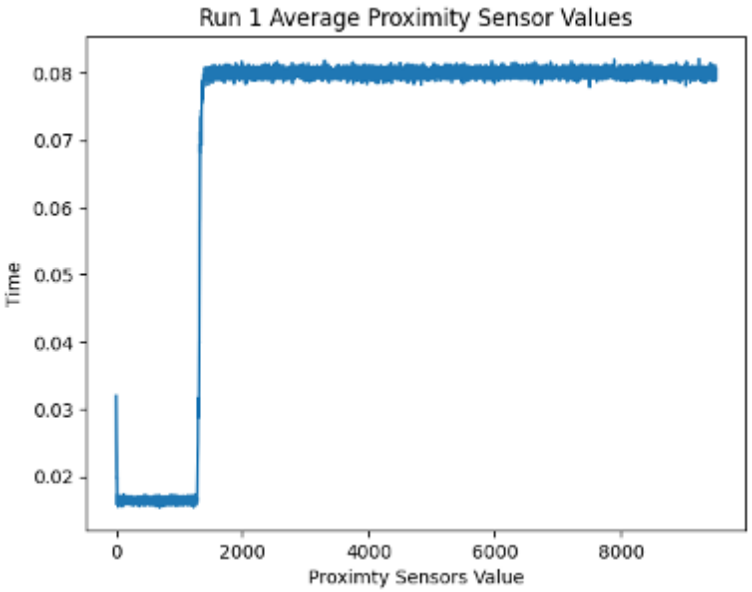

Figure 3.9: Run 1 Average Proximity Sensor Values

3.4 Run 2

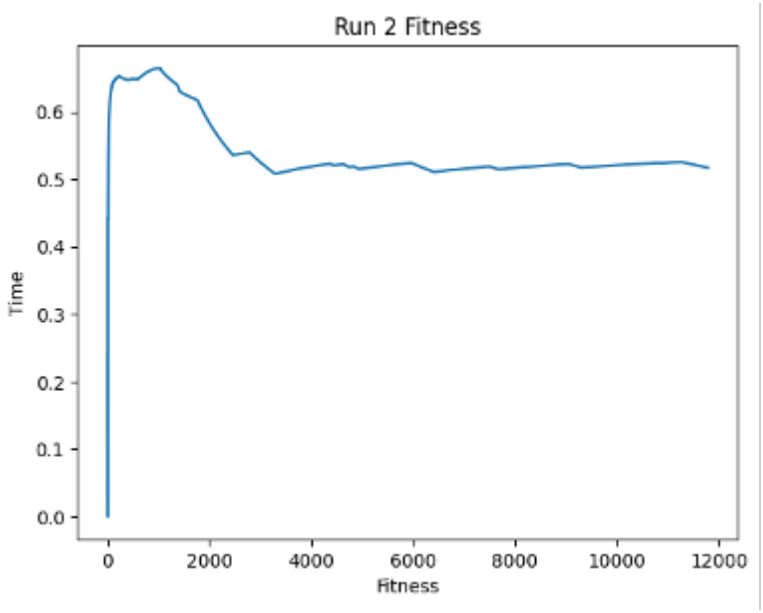

Figure 3.10: Run 2 Fitness

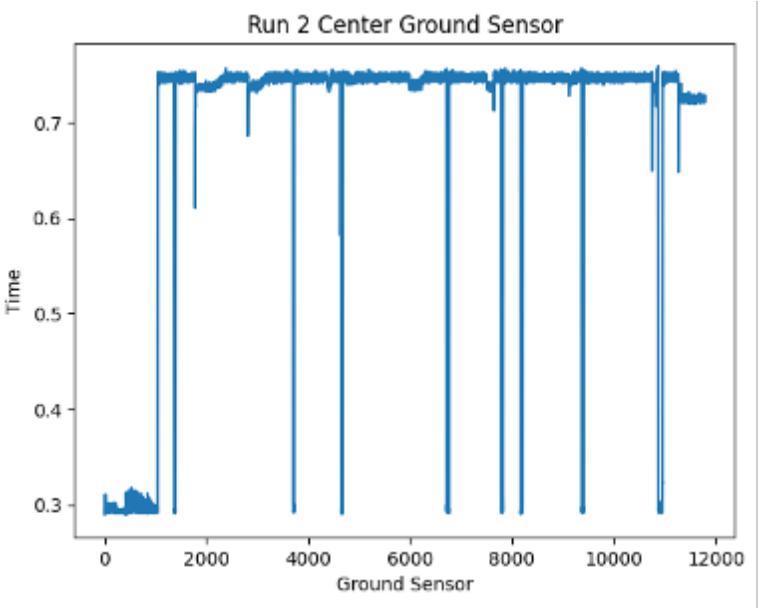

Figure 3.11: Run 2 Center Ground Sensor

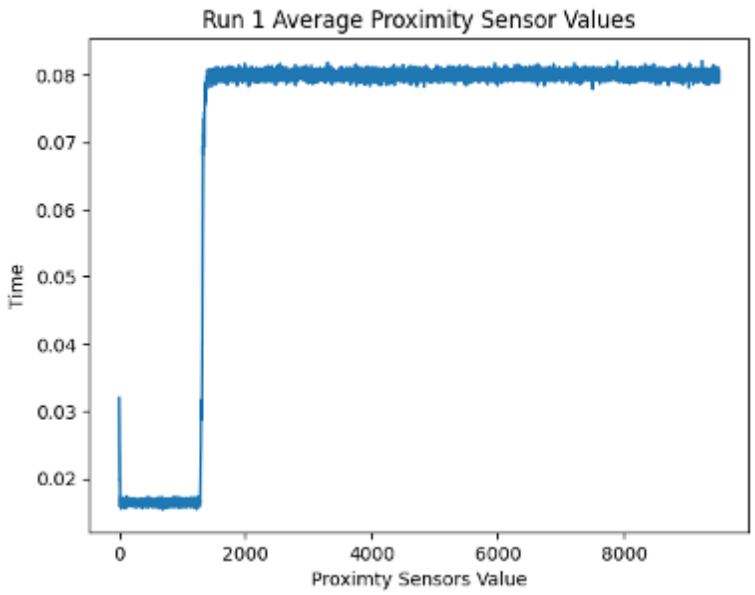

Figure 3.12: Run 2 Average Proximity Sensor Values

The robot doesn't account for light which doesn't influence its path making decisions. The endzone fitness function should encourage the robot to go to the endzone which may explain its looping behaviour in run 2.

Population metrics such as crossover, mutation rate and generation size were changed constantly to test all available options [Ahmadzadeh and Masehian, 2015].

3.5 Discussions and conclusions

Our BBR robot did succeed in both routes. However, the code could have been improved so that the robot moved around the objects in more smooth manner, especially when moving around the box. According to [Nolfi and Floreano, 2000] our approach appears to fall under the completive behaviours approach, since our if statements are nested so some behaviours require the conditions of others to be true.

Our ER robot failed in both runs. However, the first run demonstrates its ability to follow the line while the second displays the avoidance of the obstacle. According to [Doncieux et al., 2015], with more training and variation of the population metrics and longer experiment times, the robot may have achieved the goal.

Bibliography

[Ahmadzadeh and Masehian, 2015] Ahmadzadeh, H. and Masehian, E. (2015). Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization. *Artificial Intelligence*, 223:27–64. 24

[Doncieux et al., 2015] Doncieux, S., Bredeche, N., Mouret, J.-B., and Eiben, A. E. (2015). Evolutionary robotics: what, why, and where to. *Frontiers in Robotics and AI*, 2:4. 25

[Miller et al., 1989] Miller, D., Atkinson, D., Wilcox, B., and Mishkin, A. (1989). Autonomous navigation and control of a mars rover. *IFAC Proceedings Volumes*, 22(7):111–114. IFAC Symposium on Automatic Control in Aerospace, Tsukuba, Japan, 17-21 July 1989. 1

[Nolfi and Floreano, 2000] Nolfi, S. and Floreano, D. (2000). *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. MIT press. 1, 25

[Vargas, 2021] Vargas, P. (2021). Intelligent robotics lectures. 18

[Yeon et al., 2015] Yeon, A., Visvanathan, R., Mamduh, S., Kamarudin, K., Kamarudin, L., and Zakaria, A. (2015). Implementation of behaviour based robot with sense of smell and sight. *Procedia Computer Science*, 76:119–125. 2015 IEEE International Symposium on Robotics and Intelligent Sensors (IEEE IRIS2015). 1