Word Count: 2522

HERIOT-WATT UNIVERSITY, DUBAI CAMPUS

Intelligent Robotics Coursework: Robotics Project Report

F20RO: Intelligent Robotics

Date: Nov 30 2023

Ishaq Marashy (H00255437)

Word Count: 2522

Introduction

The goal of this project is to develop controllers for a "Mars rover" in a simulated environment using Webots. Each controller will enable the rover to navigate to a collection/reward zone while following a predefined path. The rover's decision-making process involves selecting one of two routes, Route A or Route B, based on the state of a surface beacon, which is simulated by enabling and disabling the light source.
The project involves simulating a Mars rover in Webots using the e-puck robot, establishing a collection/reward zone, implementing surface beacon status-based route selection, using autonomous path tracking, and obstacle detection and avoidance. The assessment of a robot controller's performance focuses on the time taken to reach the reward zone while adhering to the chosen path, which is determined by the beacon's state, and effectively avoiding obstacles along the way.

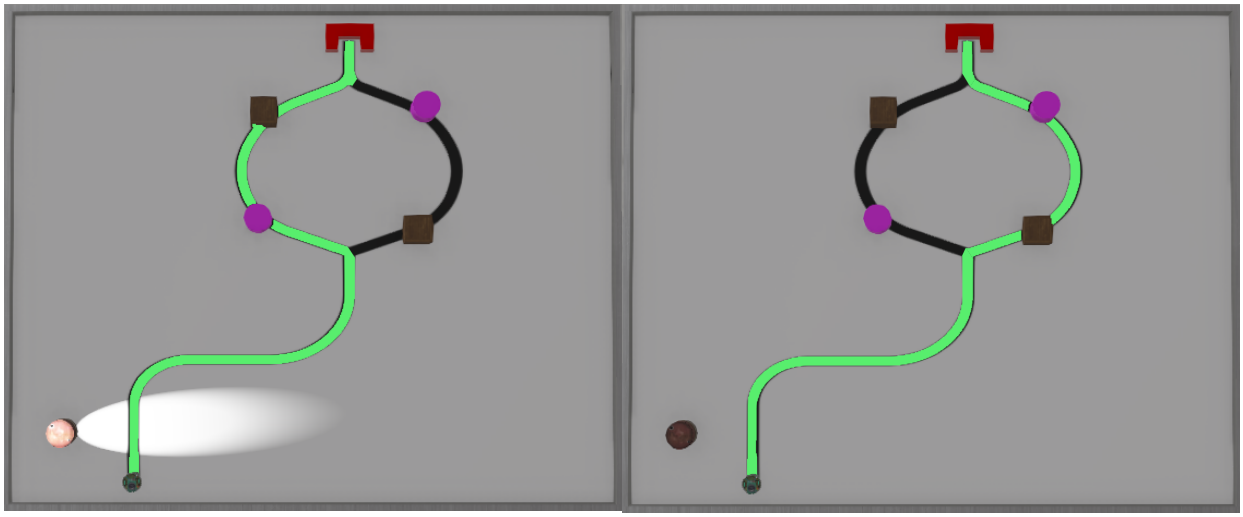

Figure 1 Paths the robot takes. The left side depicts Route A, while the right side shows Route B.

The robot follows the paths represented by the green line displayed in figure 1 when the beacon is signaling the path change which is represented by the light being emitted by the barrel. The figure 2 shows route B shows the other path the robot takes when the beacon is disabled. The reward zone is the red block at the end of the line path.
Two approaches are used:

- Behavior-Based Robotics (BBR): A BBR robot controller is developed to guide the rover to the reward zone, considering the state of the surface beacon. The controller uses predefined behaviors and decision rules.
- Evolutionary Robotics (ER): An ER robot controller is created to guide the rover based on the surface beacon's status. This approach relies on evolutionary algorithms to control the rover's behavior.

Individuals are evaluated in Route A and Route B based on how well it follows the line and if they are able to reach the reward zone.

E-Puck Configuration

The e-puck robot was developed with the specific purpose of serving as an educational tool in the field of engineering. The primary motivation behind its creation was to provide a versatile and accessible platform for teaching various engineering disciplines at the university level at an affordable level (Gonçalves *et al.*, 2009).

Sensor Configuration

1. **Ground Sensors:** Three ground sensors placed at the front of the e-puck are used to detect the black line on the arena.
2. **Proximity Sensors:** Eight proximity sensors around the e-puck, allowing it to detect obstacles.
3. **Light Sensors:** Eight light sensors around the e-puck are used to detect the beacon light signal.

Actuator Configuration

1. **Wheels:** The e-puck is equipped with two wheels, each independently controlled.
2. **Emitter and Receiver:** The e-puck uses communication modules such as emitters and receivers, to exchange information with the supervisor robot mainly for the Evolutionary Robotics (ER) approach. This allows for the evaluation of the robot's fitness and its performance in the simulated environment.

Word Count: 2522

Task 1

Design and Implementation

For the BBR task, we used sensors to determine the presence of lines and objects. This requires mapping sensor values to true or false based on thresholds for different sensor types. For the BBR to move predictably as designed we decided three ground sensors, eight proximity sensors and eight light sensors is sufficient to complete this task. The beacon's signal is represented by the light and thus light sensors are used.

```
def run_robot(self):
    while self.robot.step(self.time_step) != -1 and not self.stop:
        # history useful for exit and enter line events
        self.groundp=self.ground
        self.proximityp=self.proximity
        # read and map ground sensors to True/False based on sensor value
        # 310 is a threshold for sensor values when it finds black colored floor
        self.ground = [False if x > 310 else True for x in
                       [self.left_ir.getValue(),
                        self.center_ir.getValue(),
                        self.right_ir.getValue()]]

        # read and map proximity sensor values to True/False
        # 250 is a threshold for sensor values when it finds black colored floor
        self.proximity = [False if x.getValue() < 250 else True for x in self.proximity_sensors]
        self.sense_compute_and_actuate()
        if self.stop:
            print("Time:",self.robot.getTime())
    self.lr(0,0)
```

Figure 2 Robot's execution cycle, displaying the true and false mapping of sensors based on thresholds.

We also developed a state manager to trigger events based on the sensor readings.

```
# state/event manager
# sets enables/disables both avoid and line follow modules
# signals when object is in proximity
# signals enter/exit line event.
def update(self):
    # stops we bot at "end state"
    if sum(self.proximity)>2:
        self.stop=True
    # check if object is in proximity
    if self.proximity[0] and self.proximity[7] :
        self.close=True
    elif not any(self.proximity[1:7]):
        self.close=False
    # line exit event
    if not self.go_around and (all(self.groundp) \
        and not any(self.ground)) and self.close:
        self.go_around=True
    # line enter event
    elif self.go_around and (all(self.ground)):
        self.go_around=False
    # sets line to true when its detecting one
    self.line_end = any(self.ground)
    # enable go around if front sensors detect object
    # disable line follow module till next line enter event
    if any(self.proximity[0:2]) or any(self.proximity[6:8]):
        self.go_around=True
    # check for light event
    if not self.beacon:
            for i in range(8):
                    if self.light_sensors[i].getValue()<2000:
                        self.beacon=True
                        break
```

Figure 3 Robot's event and state manager.

Explanation:

- **If Statement 1:** Signals a stop event when three or more proximity sensors detect an obstacle, indicating that a corner has been reached. This is used for stopping at the reward zone.
- **If Statement 2:** Signals the presence of an object in front of the e-puck and maintains this signal until there are no objects around the e-puck's sides and back. This is achieved by using the eight proximity sensors placed around the e-puck which signal true when an object is nearby.
- **If Statement 3:** Manages the exit line event, where the e-puck leaves the line to navigate around an object. An else if statement handles the line entering event and signals false when entering a line. This is done by comparing the previous state of the ground sensors with its current state.
- **Self.line_end** is the end of line event signal. It checks the all ground sensors for one true signal then signals true if one is detected or false if it is not.

- **If Statement 4:** Sends a go around signal while the e-puck goes around an object. This uses the four proximity centers at the front of the e-puck.
- **If Statement 5:** Maintains a true signal if the e-puck has detected a light source. This is achieved by switching a boolean value in the controller when a light sensor around e-puck detects a light.

```
# helper function for managing motor speed
def lr(self,l,r):
    self.velocity_left = l*self.max_speed
    self.velocity_right = r*self.max_speed
    self.left_motor.setVelocity(self.velocity_left)
    self.right_motor.setVelocity(self.velocity_right)
```

Figure 4 lr is a helper function to set left and right wheel velocity.

```
# follow line module
def follow_line(self):
    # left 0 center 1 right 2
    if not self.go_around:
        if self.ground[0] and self.ground[1] and self.ground[2]:
            self.lr(0.7,0.7)
        elif self.ground[0] and self.ground[1] and not self.ground[2]:
            self.lr(0.2,0.7)
        elif self.ground[0] and not self.ground[1] and not self.ground[2]:
            self.lr(0.2,0.7)
        elif not self.ground[0] and self.ground[1] and self.ground[2]:
            self.lr(0.7,0.2)
        elif not self.ground[0] and not self.ground[1] and self.ground[2]:
            self.lr(0.7,0.2)
        elif self.ground[0] and not self.ground[1] and self.ground[2]:
            self.lr(0.4,0.4)
        elif not self.beacon and not self.ground[0] and not self.ground[1] and not self.ground[2]:
            self.lr(0.5,0.1)
        elif not self.ground[0] and not self.ground[1] and not self.ground[2]:
            self.lr(0.1,0.5)
        elif not self.beacon:
            self.lr(0.1,0.5)
        else:
            self.lr(0.5,0.1)
```

Figure 5 Robot's line following function

The follow_line function adjusts the left and right wheel speeds to keep the e-puck centered on the line. For instance, if the left front sensor detects an absence of the line, we adjust by turning to the right until the sensor detects the line again. The beacon signal determines turning preference when no line is detected, allowing the e-puck to change direction at decision points. Keep in mind that this module is disabled when the go around signal is active.

```python
# object avoid module
def avoid(self):
    # creates a turning preference based on "beacon signal"
    if not self.beacon:
        # front sensors are 0,1(right front) and 6,7 (left front)
        if any(self.proximity[0:2]) or any(self.proximity[6:8]):
            self.lr(0.8,-0.4)
        elif self.proximity[2]:
            self.lr(0.7,0.2)
        elif self.proximity[5]:
            self.lr(0.2,0.7)
        else:
            self.lr(0.1,0.5)
    else:
        if any(self.proximity[0:2]) or any(self.proximity[6:8]):
            self.lr(-0.4, 0.8)
        elif self.proximity[2]:
            self.lr(0.7,0.2)
        elif self.proximity[5]:
            self.lr(0.2,0.7)
        else:
            self.lr(0.5,0.1)
```

Figure 6 Robot's object avoidance function

The avoid function checks front proximity sensors, adjusting speeds to go around objects. For example, if an object is in front and a signal is detected, we rotate counterclockwise while using left and right proximity sensors to navigate around the object with an angled velocity to maintain proximity. When an object is on the right of the e-puck it will move forward with a slight right turn to insure object hugging. Once again turning preference is based on the beacon signal.

```python
def sense_compute_and_actuate(self):
    # run modules
    self.lr(0,0)
    self.update()
    self.avoid()
    self.follow_line()
```

Figure 6 Robot's function execution order.

Finally, the modules are executed in the following order: reset wheel speeds to zero, check for events, execute avoidance modules, and attempt to follow a line.

Word Count: 2522

Sensor Value Plots

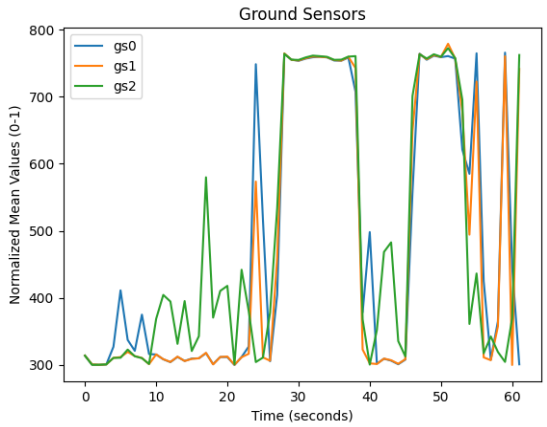

Figure 7 Normalized ground sensor values. Without a beacon signal.

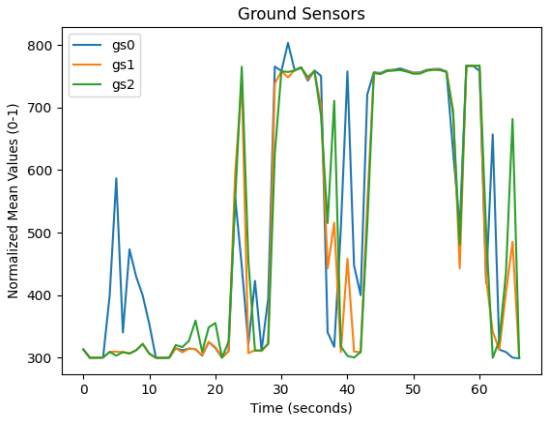

Figure 10 Normalized ground sensor values. With a beacon signal.

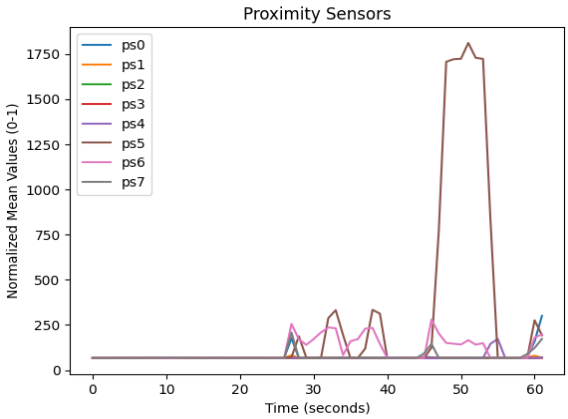

Figure 8 Normalized proximity sensor values. Without a beacon signal.

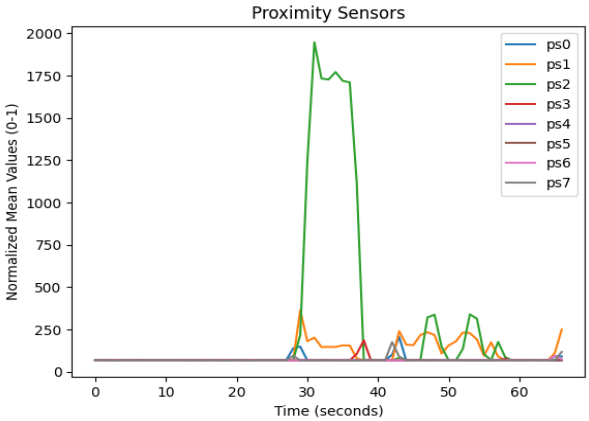

Figure 11 Normalized proximity sensor values.With a beacon signal.

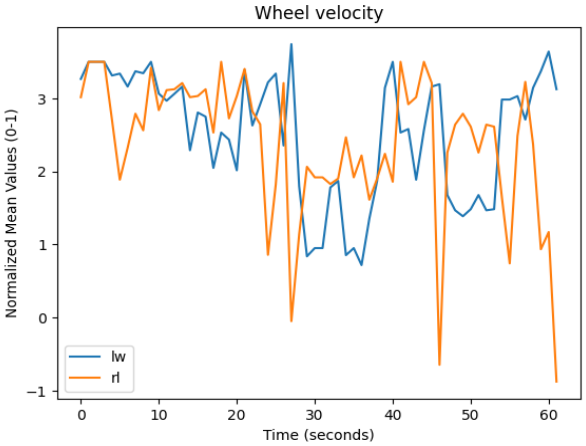

Figure 9 Normalized wheel velocity values. Without a beacon signal.

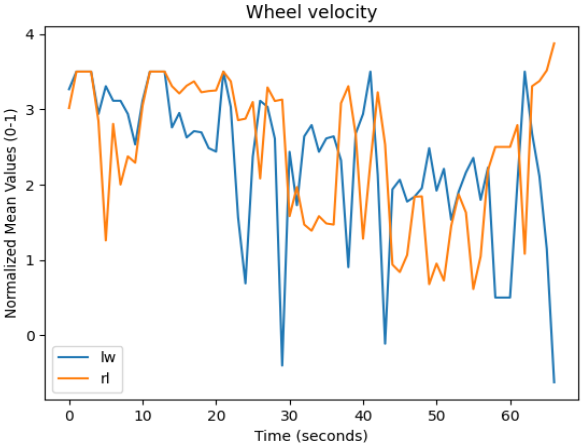

Figure 12 Normalized wheel velocity values. With a beacon signal.

Word Count: 2522

Results and Analysis

The resulting behavior of the robot is that it follows the line while circumventing objects blocking the path of the e-puck. Until it reaches the goal and is disabled by the stop event and displays the simulated time in the console.

Behavior-based robots (BBRs) like the e-pucks controller are not adaptive. They react to their environment based on sensors and predefined rules. If the e-puck is moved from the initial line or faces disruptions such as a change in its environment, it might not be able to reach the reward zone.

By looking at sensor plots we can make assumptions that assist in creating the ER for Task 2. Such as figure 7, 10, 9 and 12. Judging from the values where the graphs show distinct absence of ground sensor readings caused by object avoidance, there is a need to compensate for the epuck leaving the line. We can also observe that in figures 9 and 12, that route A encourages the left wheel to move at higher velocities than the right wheel at certain points of the route and the opposite is true for route B. This can be useful for determining the encouragement for route switching.

Developing a robot with these capabilities requires a careful analysis of possible events during its mission and a good understanding of how to manage transition states. It requires thorough testing and adjustments until a solution is reached. When done correctly, the robot can complete its task fast and efficiently. In our case, the e-puck manages to reach the reward zone while taking route B in 62 seconds when the beacon does not signal, and the e-puck reaches the reward zone while taking route A when the beacon does signal in an average of 71 seconds. We can observe from figure 9 and 12 that on average one wheel will have a higher velocity depending on the route the robot takes.

Word Count: 2522

Task 2

Design and Implementation

For the ER task, we started by determining the detection threshold for each sensor. This task required three ground, eight light, and eight proximity sensors.

E-Puck Controller

Sensor values underwent processing using the bin function, which clips and normalizes values between specified thresholds. This normalization step is important to integrate the inputs into the neural network, as demonstrated by Jakobi (1997). It ensures that sensor values are within a standardized range.

```
def bin(self, min_val, max_val, val):
    if isinstance(val, list):
        return [self.bin(min_val, max_val, v) for v in val]
    else:
        if val > max_val:
            return 1.0
        elif val < min_val:
            return 0.0
        else:
            return (val-min_val)/(max_val-min_val)
```

Figure 13 Normalizing function.

The sensor values are normalized, constraining them between 0-1, before being appended to the input array for the neural network. Two engineered inputs are integrated into the e-puck controller. The first involves a decaying variable for the light sensor, initially set to 1 upon light detection, and then decaying by a factor of 0.999 over the trial duration. A threshold is applied, with the value persisting at 1 if above 0.155 and otherwise 0, this takes approximately 60 seconds. The second input focuses on decaying line input sensing, allowing a 10 second tolerance for the e-puck leaving the line before it becomes 0 again. The variable resets to 1 upon detecting a line again. These engineered inputs are justified by insights gained from the study conducted by Mitri *et al.* (2010), whose study used a similar technique for ground color detection memory.

```
def run_robot(self):
    self.inputs =[]
    while self.robot.step(self.time_step) != -1:
        self.inputs =[]
        self.handle_emitter()
        self.handle_receiver()

        self.inputs+=self.bin(350,700,[self.left_ir.getValue(),self.center_ir.getValue(),self.right_ir.getValue()])
        self.inputs+=self.bin(100,200,[x.getValue() for x in self.proximity_sensors])

        # while above 0.3 input will be 1 and when its below input will be 0
        ls=self.bin(300,3000,min([x.getValue()  for x in self.light_sensors]))
        if ls==0:
            self.ls_prev=ls
        else:
            # 0.155 at 60s
            self.ls_prev=self.ls_prev*0.999

        if 1-np.min(self.inputs[0:3])==1.0:
            self.line_prev=1
        else:
            # 0.04 at 10s
            self.line_prev*=0.99

        self.inputs+=[1.0 if self.ls_prev > 0.155 else 0]
        self.inputs+=[1.0 if self.line_prev > 0.04 else 0]
        self.inputs=np.round(self.inputs,3)
        # print(self.inputs)
        self.check_for_new_genes()
        self.calculate_fitness()
        self.sense_compute_and_actuate()
```

Figure 14 Robot execution cycle. Inputs are normalized and run through the neural network.

Word Count: 2522

Neural network parameters

The inputs to the neural network are those defined in the run robot function. The neural network should have the same number of neurons in its input layer (Jakobi, 1997). The number of hidden layers and neurons was chosen arbitrarily and tested manually until a working model was found, consisting of 4 hidden layers, each containing 13 neurons. The neurons in the output layer are directly connected to the wheels of the e-puck, controlling the velocity of the wheels (Mitri *et al.*, 2010).

```
self.number_input_layer = 13
self.number_hidden_layer = [13,13,13,13]
self.number_output_layer = 2
```

Figure 15 Neural network structure.

```python
def calculate_fitness(self):
    ### DEFINE the fitness function to increase the speed of the robot and
    ### to encourage the robot to move forward
    forwardFitness =self.bin(0,self.max_speed,(self.velocity_left+self.velocity_right))
    forwardFitness*= 1.0
    ### DEFINE the fitness function equation to line leaving behaviour
    lineFitness = self.inputs[12]
    lineFitness*= 2.0
    ### DEFINE the fitness function equation to avoid collision
    avoidCollisionFitness = 1-np.max(self.inputs[3:11])
    avoidCollisionFitness*= 1
    ### DEFINE the fitness function equation to avoid spining behaviour
    spinningFitness = 1-self.bin(0,self.max_speed,abs(self.velocity_left - self.velocity_right))
    spinningFitness*= 1.0
    if self.inputs[11]==1.0:
        turnFitness = 1.0 if self.velocity_right <= self.velocity_left else 0
    else:
        turnFitness = 1.0 if self.velocity_left <= self.velocity_right else 0
    ### DEFINE the fitness function equation of this iteration which should be a combination of the previous functions
    combinedFitness = (turnFitness+lineFitness+forwardFitness+spinningFitness+avoidCollisionFitness)/6
    self.fitness_values.append(combinedFitness)
    fitm=np.mean(self.fitness_values)
    self.fitness = fitm
```

Figure 16 Fitness function.

The fitness function is defined as follows

- **Forward Fitness:** Encourages forward movement, computed by adding left and right wheel velocities and normalizing between 0 and 1. A value of 1 indicates optimal forward behavior.
- **Line Fitness:** Promotes staying on the line by maximizing ground sensor alignment with the line. The decaying variable for line memory enables the e-puck to navigate around obstacles.
- **Collision Fitness:** Discourages staying near objects and stagnating and encourages obstacle avoidance.
- **Spinning Fitness:** Discourages spinning behavior and is maximized when both wheels sync in a direction.
- **Turn Fitness:** Reward based on the e-puck's turn direction dictated by the decaying light variable. For example, favoring a slower right wheel encourages the left path, and vice versa. This approach was chosen due to the fact we are trying to control the path the e-puck takes to reach the reward zone.

A large reward is later introduced by minimizing distance from the reward zone, preventing stagnation. These forward and spinning fitnesses are heavily inspired by Floreano *et al.* (1994).

Supervisor Controller

The supervisor is responsible for running the genetic algorithm and selecting the best individuals. For this genetic algorithm these were the parameters chosen. Which includes 700 generations with a population of 20 per population and 2 of those are elites. The amount of time every simulation was given is 100 seconds. This means for every generation there will be 20 individuals and of those 20 individuals the best 2 individuals (elites) will always be preserved (Nolfi and Floreano, 1998).

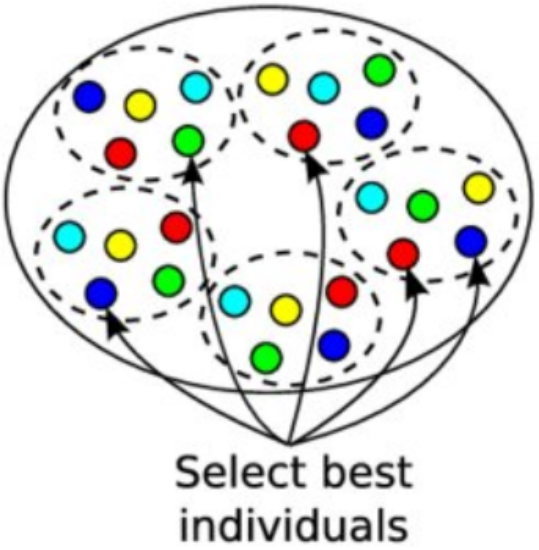

Figure 17 Best individual selection method (Mitri *et al.*, 2010).

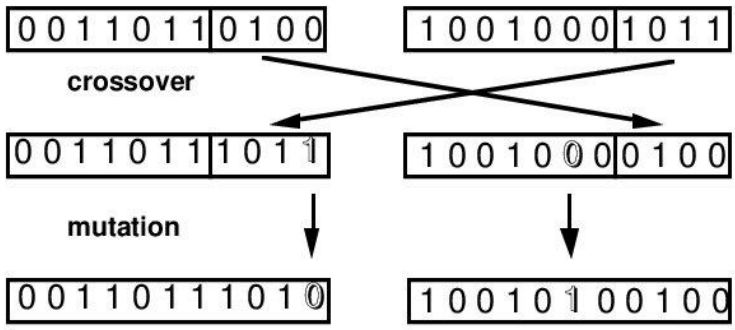

Figure 18 Visualization of Crossover and mutation (Padró, Ozón and Aplicada, 1995).

In genetic algorithms, the crossover point is the position along the chromosomes where the genetic material of two parents is exchanged to create new offspring. It plays a role in finding better genotypes. Similarly the mutation rate is the probability that a particular gene in an individual's chromosome will undergo a random change. Mutation introduces diversity into the population. The mutation rate that was chosen for this task is 10% and the crossover point is 70%.

To evaluate each robot's performance they were run through the environment twice. Once with the signal and once without. The resulting fitness is calculated as the mean of the two robots' performances. This provides a balanced evaluation of each individual's capability in both signaled and no signal scenarios.
Reset env makes sure objects do not move in the environment by colliding with the e-puck.

```
def reward(self):
        FINAL_TRANS=np.array([0.10824,0.931462,0.00173902])
        robot_trans=np.array(self.trans_field.getSFVec3f())
        x=FINAL_TRANS[0]-robot_trans[0]
        y=FINAL_TRANS[1]-robot_trans[1]
        delta_trans = -math.sqrt(math.pow(x,2)+math.pow(y,2))*2
        reward=delta_trans
        print(f'G{delta_trans}')
        return reward
```

Figure 19 Reward function.

```
def evaluate_genotype(self,genotype,generation):
    fitnessPerTrial = []
    left=True
    # TRIAL: TURN RIGHT
    self.emitterData = str(genotype)
    self.reset_env(genotype,left)
    self.run_seconds(self.time_experiment)
    fitness = self.receivedFitness
    fitness+=self.reward(left)
    fitnessPerTrial.append(fitness)
    print("Fitness: {}".format(fitness))

    # TRIAL: TURN LEFT
    self.emitterData = str(genotype)
    self.reset_env(genotype,not left)
    self.run_seconds(self.time_experiment)
    fitness = self.receivedFitness
    fitness+=self.reward(not left)
    fitnessPerTrial.append(fitness)
    print("Fitness: {}".format(fitness))

    fitness = np.mean(fitnessPerTrial)
    current = (generation,genotype,fitness)
    self.genotypes.append(current)

    return fitness
```

Figure 19 Trial function.

The communicated fitness from the e-puck is received and then divided by the distance between the robot and the goal. Subtraction is used because the robot must adhere to the line-following behavior as giving it a high negative reward will encourage the genetic algorithm to minimize it. The distance calculation involves subtracting the robot's x and y position from the x and y position of the goal to obtain the positional difference. The robot's distance is then computed then subtracted to punish the robot for being far from the reward zone.

This process of evaluation and mutation repeats until the specified number of generations is completed. The final genome is determined by selecting the population with the highest fitness score (Floreano *et al.*, 1994).

Word Count: 2522

Sensor Value Plots

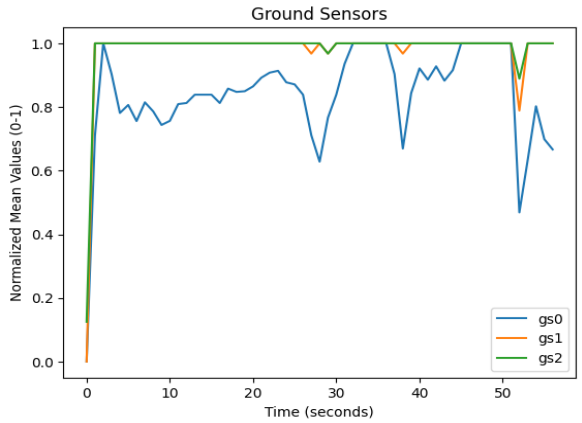

Figure 22 Normalized ground sensor values. Without a beacon signal.

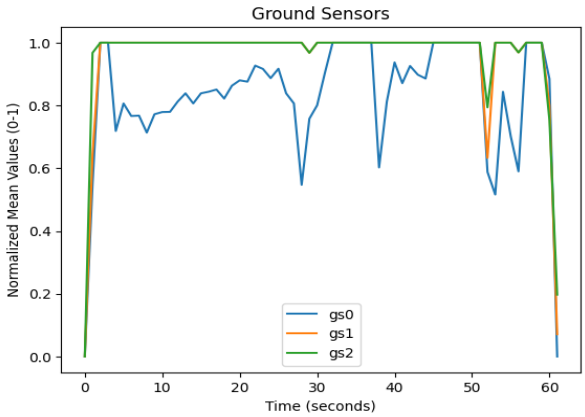

Figure 25 Normalized ground sensor values. With a beacon signal.

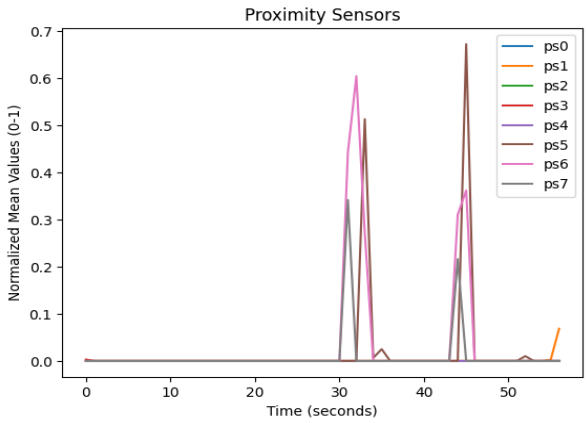

Figure 23 Normalized proximity sensor values. Without a beacon signal.

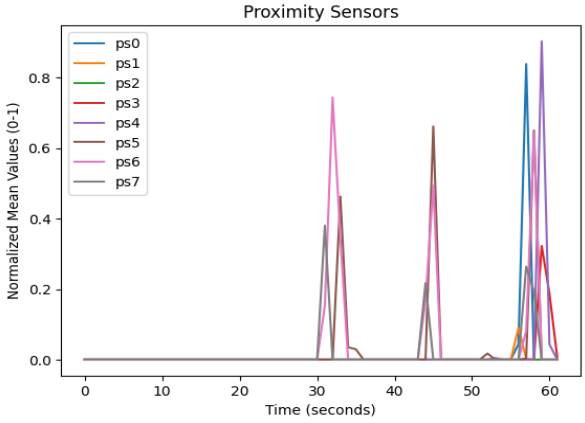

Figure 26 Normalized proximity sensor values.With a beacon signal.

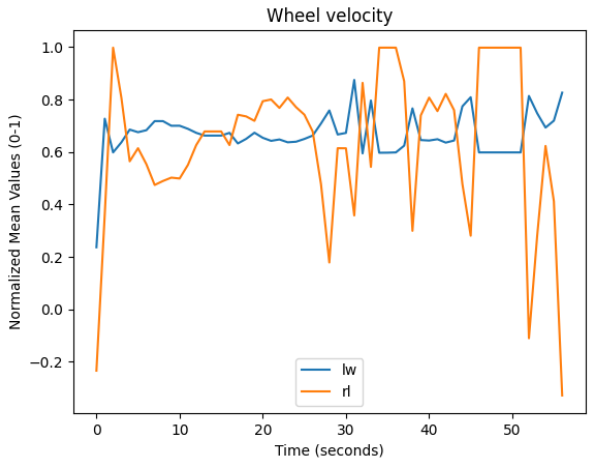

Figure 24 Normalized wheel velocity values. Without a beacon signal.

Figure 27 Normalized wheel velocity values. With a beacon signal.

Word Count: 2522

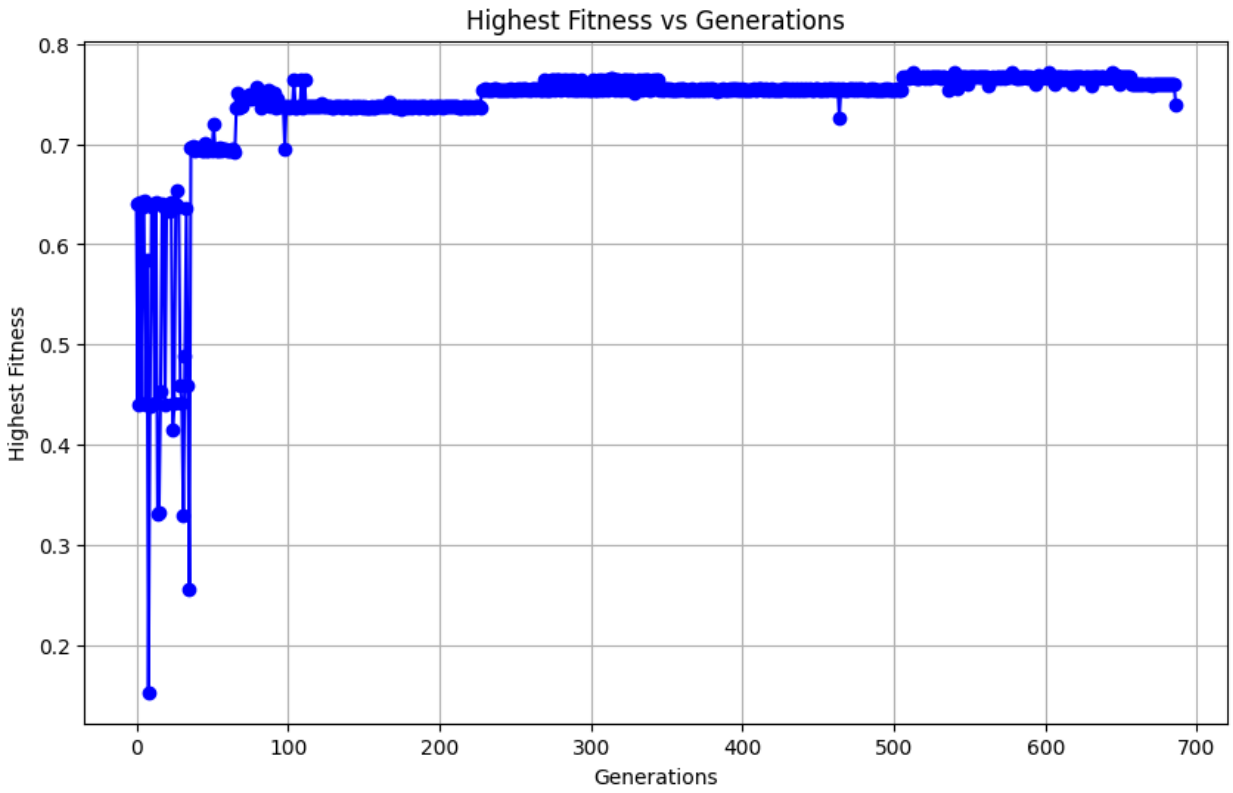

Figure 21 Highest fitness individuals Vs. generations.

Results and Analysis

The resulting behavior of the robot is that it follows the line and avoids objects blocking its path until it reaches the goal. Unlike BBRs, ERs do not use predefined rules to determine the robot's behavior. Instead, ERs use a genomic sequence that acts as weights for the multilayer perceptron. In genetic algorithms, various genomes are tested until the semi-optimal genome is found. This process generally provides a more robust model for handling the real world as it evolves with its environment.

Developing an ER requires careful analysis of the behaviors needed to achieve its mission. The testing and adjustments are done autonomously through genetic algorithms using Lamarckian evolution until a practical solution is achieved. When done correctly, the robot can complete its task but may occasionally display unusual behavior, such as skipping lines to the reward zone. In some cases, learning could be hindered by extreme rewards, especially since elitism preserves the highest fitness individuals.

Word Count: 2522

Conclusion

The BBR approach relies on predefined rules and behaviors, making it effective in fixed scenarios but less resilient to unforeseen conditions. ER showed a higher degree of adaptability, as evolving controllers adjust to changes in the environment. However, the ER approach demands careful parameter tuning and longer computational times due to the genetic algorithm's iterative nature.

The performance metrics indicated that the BBR-based e-puck completed Route A in 71 seconds and Route B in 62 seconds. The ER-based e-puck had better results in a single path, adapting its behavior based on an evolved neural network. Each experiment was given 80 seconds to reach the goal. However most runs took under a minute for the e-puck to reach the goal.

Due to the nature of BBRs the sensor inputs need to be defined to handle the problem, this presents an issue when the robot is implemented in the real world. Sensors in the real world often have noise and this may cause BBRs to fail completely. ER can be trained to handle noise in the simulated world so that it handles the real world better. This was demonstrated by (Jakobi, 1997) and then was revisited by (du Plessis, Phillips and Pretorius, 2022), where adding noise to the sensor inputs caused the ERs to better cross the reality gap.

	Time in minutes and seconds			
	First Run	Second Run	Third Run	Average Time
Task 1 A	01:07	01:08	01:18	1:11
Task 1 B	01:02	01:01	01:02	1:02
Task 2 A	5:00	5:00	5:00	5:00
Task 2 B	00:58	01:00	00:56	00:58

Table 1 Robot route completion time and their averages.

In our case the robot did not switch sides to complete Task 2 route A. However it does teach the goal at an average time of 57s while taking route B.

Word Count: 2522

References

Floreano, D. *et al.* (1994) ‘Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural Network Driven Robot’.

Gonçalves, P. *et al.* (2009) ‘The e-puck, a Robot Designed for Education in Engineering’, *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, 1.

Jakobi, N. (1997) ‘Evolutionary Robotics and the Radical Envelope-of-Noise Hypothesis’, *Adaptive Behavior*, 6(2), pp. 325–368. Available at: https://doi.org/10.1177/105971239700600205.

Mitri, S. *et al.* (2010) ‘Evolutionary Conditions for the Emergence of Communication’, *Evolution of Communication and Language in Embodied Agents* [Preprint]. Available at: https://doi.org/10.1007/978-3-642-01250-1_8.

Nolfi, S. and Floreano, D. (1998) ‘Coevolving predator and prey robots: do “arms races” arise in artificial evolution?’, *Artificial Life*, 4(4), pp. 311–335. Available at: https://doi.org/10.1162/106454698568620.

Padró, F., Ozón, J. and Aplicada, D. (1995) ‘Graph Coloring Algorithms for Assignment Problems in Radio Networks’.

du Plessis, M.C., Phillips, A.P. and Pretorius, C.J. (2022) ‘Revisiting the Use of Noise in Evolutionary Robotics’, in E. Jembere et al. (eds) *Artificial Intelligence Research*. Cham: Springer International Publishing (Communications in Computer and Information Science), pp. 211–226. Available at: https://doi.org/10.1007/978-3-030-95070-5_14.