> F21RO INTELLIGENT ROBOTICS COURSEWORK <

Word Count: 4016

Implementation of BBR and ER Robot Controllers for T-Maze Problem Solving

Reza Moghtaderi Esfahani, Syed Arif Ali, Eherar Shaikh, Suhana Ayisha

Abstract—In this coursework, we have implemented autonomous robots that can solve a T-maze problem using a robot simulation software called Webots. The robot's goal is to navigate through the maze to find its food/reward (prize). The maze is T-shaped with a left and a right arm. The robot moves forward through the maze, avoiding collision with the walls, and at the junction, chooses to go to the left or right depending on if the spotlight is detected. The prize is placed on the right arm if there is light; otherwise, the prize is placed on the left arm. The coursework is divided into two tasks. The first task involves creating a Behavior-Based Robotics (BBR) based robot that exhibits intelligent behaviors through simple, reactive actions [1]. The second task involves developing an Evolutionary Robotics (ER) based robot that uses evolutionary algorithms where the robot learns and evolves over the iterations, optimizing its decision-making and navigation performance. We analyze both approaches, highlight their strengths and weaknesses, and draw conclusions.

***Index Terms*— Robot simulation software, Behavior-Based Robotics, Evolutionary Robotics, Genetic Algorithms, Fitness function, E-puck Robot,**

I. Introduction

Small mobile robotic agents have become more popular lately, especially for testing various control algorithms in a controlled environment. The accessibility and availability of hardware platforms such as E-puck and mature software simulation environments make small robots a commonly used experimental platform [1]. In our coursework, we have used E-puck robot and Webots as the simulation software. Webots is an open source and multi-platform desktop application used to simulate robots and their behavior in different environments and physical circumstances.

A. E-puck

E-puck is a small, versatile robot that is being used widely across many educational institutes and laboratories for experiments and prototyping. It has a diameter of 70 mm and a structure that ensures the support of the motors, the printed circuit and the battery. It is equipped with eight IR proximity sensors and eight lights placed as illustrated in Fig.1. They are numbered in a clockwise order and placed in such a way that it offers 360 degrees sensing capability. Both proximity sensors and light sensors use the same hardware, but function differently based on the configuration.

The proximity sensors return a high value when it is close to an object and low value when the obstacle is away from it. On the contrary, the light sensors return a low value, almost 0. when they detect light. In the absence of light they return a high value, around 4000. Understanding this is crucial for our coursework to be implemented correctly.

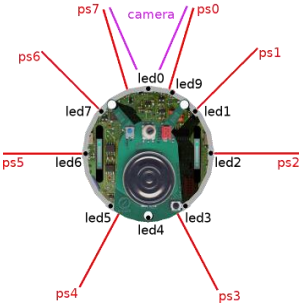

Fig. 1. E-puck robot diagram

B. Behavior-Based Robotics

The concept of BBR was introduced in the mid-1980s, and was championed by Rodney Brooks and others [2].

In behavior-based robotics, robots have a set of simple behaviors or rules that tell them how to react to environmental changes. A behavior will be defined simply as a sequence of actions performed to achieve some goal. Thus, for example, an obstacle avoidance behavior may consist of the actions of stopping, turning, and starting to move again in a different direction [2].

This approach emphasizes the significance of interactions between the robot and its environment, allowing for flexible and adaptive behavior in dynamic situations that are predefined. We get the information we need as input from the robot's sensors, and the robot uses that information to correct its actions gradually according to changes in the immediate environment. These behaviors are layered and organized into a hierarchy. Robots built on this architecture consist of few simple rules which are organized in such a way that it can produce very "intelligent" behavior [3].

In general, a behavior-based robot is first equipped with the most fundamental of all behaviors, namely those that deal with survival [2]. For example, in our coursework, we first try to get the robot moving without hitting an obstacle.

C. Evolutionary Robotics

Evolutionary Robotics is a field that aims to apply evolutionary computation techniques to evolve the overall design or controllers, or both, for real and simulated autonomous robots.
Evolutionary robotics is distinct from other fields of engineering in that it is inherently based on a biological mechanism. It applies the selection, variation, and heredity principles of natural evolution to designing robots with embodied intelligence [4]. It uses evolutionary algorithms to optimize the robot's decision-making and navigation performance. In our coursework, we have used Genetic algorithms (GAs) to achieve this. GAs translates the biological concepts of evolution into algorithmic recipes [5].

The key steps involved in this process include [6], [7]:

- **Initialization**: A set of individuals is generated. This population must be as random as possible to cover the whole solution space.

- **Fitness Evaluation**: The fitness function measures the quality of the solutions the Genetic Algorithm has generated [5]. Each individual is assessed using a fitness function that measures competition between individuals; the fittest has more chances to survive.
  In case of multiple objectives that have to be optimized at the same time, the fitness function values of each single objective can be aggregated, for example by computing the weighted sum. [5]

- **Selection**: A surplus of offspring solutions is generated, and the best are selected to achieve progress towards the optimum [5]. The best ones are selected based on their fitness scores, with fitter individuals having a higher probability of being chosen for reproduction.
  Some selection algorithms are based on randomness along with fitness scores such as Roulette Wheel Selection, Tournament Selection

- **Crossover**: Crossover is an operation that combines the genetic material of two or more individuals [5]. These individuals are selected, and a crossover point is set up randomly. Genes are selected from the parents and crossed over to create new offspring.
  For example, if 0010-110010 is the first parent and 1111-010111 is the second one, one-point crossover would randomly choose a position, let us assume 4, and generate the two offspring candidate solutions 0010- 010111 and 1111-110010 [5].

- **Mutation**: It is an important genetic operator that randomly changes genes in an offspring to maintain genetic diversity and explore new solution spaces. For example, in a binary representation, a byte at a random position is flipped from 0 to 1

D. Comparison of Behaviour-Based Robotics (BBR) and Evolutionary Robotics (ER) in Robotics

Aspect	BBR	ER
Principle	It relies on a simple set of predefined rules to determine how to react to a situation.	It uses evolutionary algorithms to optimize its behavior and evolve over time.
Manual Effort and Knowledge of the Domain	A developer manually programs it, so developers must be well-versed and knowledgeable about it.	It requires less manual effort. The process of finding a solution is automated by algorithms based on fitness functions. Hence, it does not require someone to be knowledgeable in the domain.
Robustness	Not adaptable. Every change in the environment must be predefined. Hence, it is not very robust.	It is highly adaptable to new situations and evolves based on different scenarios. It is more robust than BBR.
Complexity	Well suited for simple to moderately complex tasks where behaviors can be explicitly defined.	Better suited for complex tasks where the details can be too complex to anticipate.
Time Efficiency	Simple tasks are usually faster to do since they are manually programmed. Designing complex programs might take time.	It can take a long time to generate solutions. It is not very time-efficient
Scalability	It can be challenging to scale if the problem increases in complexity.	It is easily scalable if the problem increases in complexity.
Learning Capability	Increasing the number of iterations will not improve the results. It has no learning capability.	Since this is learning-based, the more you train, the better the performance will be until a solution is found.

Both approaches can work well together, with BBR handling simpler, rule-based behaviors and ER handling adaptive challenges [8].

II. Problem

We have a T-Maze with two arms, with food available at the end of each arm, depending on environmental clues (light). The robot should start at the beginning of the bottom side of the T-Maze and move forward while avoiding the maze walls towards the end of either of arms by turning right or left at the junction connecting the arms of the maze. The ‘food/reward/prize’ will be at the end of one of the arms. The robot should turn right at the end of the first arm corridor (junction) if it senses a beam of light from the right side of the middle of the path connecting the bottom of the maze to the arms (T-Maze B). Otherwise, the robot should turn left (T-Maze A).
Our aim is to create the quickest robot that can explore the T-maze in less time while searching for a prize.

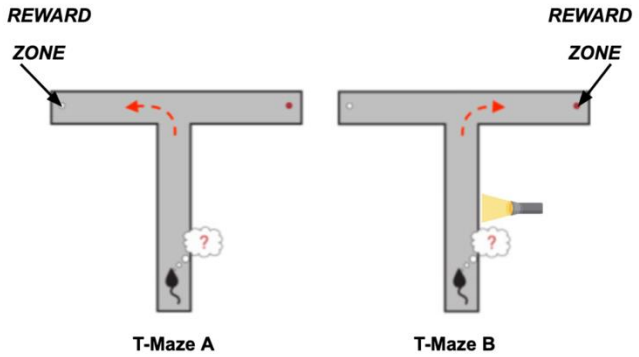

Fig. 2. T-Maze A and T-Maze B

A. Conditions of the experiment

1. Use the e-puck robot as our robot.
2. GPS node cannot be added to the e-puck robot
3. Use T-Maze arenas just as they are given. The walls cannot be moved. The dimensions of the arena, areas, and obstacles should not be changed. The black marking in the middle of the path must be removed.
4. The maximum time to perform any of the tasks is 5 minutes for each part of the coursework. After that, the simulation should be stopped, and 5 min should be added to that specific group as the time spent, irrespective of whether the robot(s) have completed the task.

III. Methods and Implementation Rationale

A. Task 1- BBR:

Robot Controller:

We have added a controller for the robot, which navigates through the maze while reacting to a light beam (placed on the right side), avoiding the walls, and centering itself between them, turning at junction, and detects the end of the maze.

1. Main Program (__main__)

- Creates a Robot object and instantiates the controller with it.
- Starts the robot's control loop using run_robot.

2. Initialization (__init__ Method)
The controller initializes the following:

- Robot parameters: This sets the time step for simulation updates, maximum speed, adjustment factor for steering, maximum motor velocity, and end count to track the end of all iterations (six iterations in total, three for each type of maze).
- Reset state: Calls the reset() method to configure motors, sensors, and internal state variable by resetting them at each iteration.

These values remain static throughout the experiment.

3. Resetting the Robot (reset Method):
This method configures and initializes the robot to the state it must be at the beginning of an iteration:

- Motors: Retrieves motor devices (left wheel motor and right wheel motor) and sets their position to infinity (allows continuous rotation).
- Sets initial velocity to 0.0 (stops the robot).
- Emitter: This enables a communication emitter for signaling states like detecting light (b'L') or reaching the maze's end (b'E'). “b” is a byte data type. The signal is sent to the supervisor.
- Proximity Sensors: Retrieves and enables eight proximity sensors for detecting nearby obstacles.
- Light Sensors: Retrieves and enables eight light sensors to detect light sources.
- Triggers: Initializes the robot's state variables: light_trigger, end_trigger, and turn_trigger. These are reactive variables which can accomplish their functionality wherever there is a junction (turn), light (emitted from right side), or end detection (reaching a dead-end).

4. Main Loop (run_robot Method)
The robot continuously steps through the time steps, updating its state and behaviors:

1. Checks if the robot has reached the maze's end: Stops and resets after reaching the end for up to 6 iterations.
2. Calls sense_and_actuate to perceive and act on the environment.
3. Stops the loop when all tasks are complete.

5. Sensor Reading and Processing (sense_and_actuate):

Reads all proximity and light sensors to determine the robot's surroundings. Updates thresholds and tolerances for wall detection. Calls methods like light_is_found and center_between_walls to respond to the environment.

6. Detecting Light (light_is_found)
Checks the value of the light sensor.

- If no light has been previously detected and light is found from right side of the robot, the light trigger is set to True, and a signal is emitted to the supervisor to indicate that light has been detected.

7. Centering Between Walls (center_between_walls):
It is the main method responsible for the logic of the motion behind the robot’s movements. Calculates the average proximity to the right and left walls and adjusts motor velocities based on the difference between the sides to center the robot. It turns left if the left wall is farther and right if the right wall is farther. If the walls are equally distant, it moves straight (due to sensor fluctuations in the robot simulation environments such as webots, it is rare for the robot to move straight for more than few timesteps, so the robot is always adjusting its position). Centering between walls method also handles the turning at junctions (based on the spotlight

detection), so it knows when to start the turning phase, and when to end it.

8. Movement Control Methods

- forward(): Sets the left and right motors to the maximum speed for forward movement (for forward movement, this value must be a positive and equal value for both motors).
- stop(): Stops the robot by setting motor velocities to 0.
- motor_flush(): Stops the robot, preparing it for resetting motor speeds. This method is technically same as stop() method, but with flushing considerations.

9. Velocity Clamping
This method ensures the motor speed does not exceed the allowed maximum by clamping and limiting it within a specified range.

Supervisor Controller:

The supervisor manages the environment (T-maze) and the whole simulation through which the robot navigates toward the food under either light beam conditions. The supervisor handles light status, prize position, robot position, time keeping, CSV generation, ...

1. Main Program (__main__)

Instantiates the PrizeSupervisor Class and starts the simulation by calling the run() method

2. Runs the main loop (run())

This contains the main loop, which steps the simulation according to the time step provided while also continuously checking for proximity to the prize and handling communication with the controller by receiver.

3. Initializes the Supervisor and Receiver (_init__())

- Initializes the supervisor and receiver to listen for messages from the robot controller.
- Initializes lists to keep track of iteration durations and a timer for the overall task completion.
- Keeps track of iteration count
- Calls reset() to set up the environment to its initial state after each iteration.

4. Resetting the Supervisor (reset ())

- Resets the environment to its intital state.
- Initializes objects: the prize, e-puck robot, and light source using their DEF name from Webots.
- Sets the light's state and prize's initial position by calling set_initial_light_state().
- Records the start time of the current iteration.
- Calls simulationResetPhysics() method to reset the physics of the simulation at each iteration. This is done to make sure there is no residual or trace physical properties remaining from previous iteration affecting the current iteration and robot functionality.

5. Sets the Light State (set_initial_light_state())

- Decides that the light in the T-maze will be "off" for the first 3 iterations and "on" for the remaining 3 iterations.
- Places the prize based on the light state. If the light is on, the prize spawns on the right side of the T-maze; otherwise, it spawns on the left.

6. Reset E-puck position (set_epuck_initial_position()

Resets the e-puck robot's position and rotation to its starting point in the maze to maintain consistency for each iteration.

7. Checks Proximity (check_proximity ())

- Calculates the distance between the robot and the prize.
- If the robot is close to the prize, the prize disappears.
- It checks for incoming messages from the robot by its receiver. If the message is "E", it indicates the end of the iteration, triggering handle_iteration_end().
-

8. Iteration End (handle_iteration_end ())

- Calculates the duration of the current iteration and stores it with the corresponding T-maze name.
- Checks if all iterations are complete:
    - If not, resets the simulation for the next iteration.
    - If complete, writes iteration data to a CSV file.

9. Write to CSV file (write_to_csv ())

- Writes iteration durations, type of maze at each iteration, and total simulation time to a CSV file.

10. Remove Prize (disappear_prize())

- The disappearance logic is by relocating the prize to a distant location in the world (out-of-view) to stimulate the disappearance when the robot reaches (touches) it.

B. Task 2 - ER

Robot Controller:

We have added a controller that contains implementation of a multilayer perceptron (MLP). It is a fully connected neural network that takes robots sensory data as input and produces an output to control the two motors of the robot. MLP architecture comprises of 3 parts as below:

1. **Input Layer**
   - Proximity Sensors: 8 sensors for obstacle detection.
   - Light Sensors: 4 sensors to detect light intensity.

2. **Hidden Layers**
   - Hidden layers allow to learn complex patterns and relationship in input data with purposes of feature extraction, dimensionality reduction etc.
   - Size of hidden layers must be balanced to improve efficiency and performance, thereby avoiding overfitting or underfitting of the model.

3. **Output Layer**
   - The output layer comprises of 2 nodes accounting for the 2 motors of the robot.

1. Main Program (__main__)
   - Creates a Robot object and instantiates the controller with it.
   - Starts the robot's control loop using run_robot.

2. Initialization (__init__ Method)
   - Robot parameters: Sets the time step for simulation updates, maximum speed.
   - Neural Network setup: The neural network input, hidden and output layers parameter.
   - Motor and sensor initialization: Sets the value to velocity to zero and enables the proximity and light sensors.
   - Supervisor communication: Enables communication between supervisor and robot controller.
   - Fitness parameters: Initializes the fitness parameters that will be used to store fitness values.

3. Check for new genes (check_for_new_genes Method)
   - This method dynamically updates the weights of the neural network by new weights provided by the supervisor
   - This method extracts the weights and reshapes them for input layer plus bias, hidden layers and output layer.
   - It resets fitness allowing fresh evaluation of the genome.

4. Compute and Actuate (sense_compute_and_actuate Method)
   - This method processes e-puck robot sensor input data through a neural network using forward propagation.
   - This network uses a Tanh activation function.
   - The network output is set as velocity of the left and right motors.
   - To increase speed, the velocity values are scaled by a factor of 2.
   - This allows real-time decision-making for navigation and movement.

5. Detect Light Beam (detect_light_beam Method)
   - This method checks if a light beam is detected by the robot's right-side light sensors (Sensors 0,1,2,3).
   - If at least 2 sensors detect light, it sets the light_beam_detected flag to true, else it is set to false.

6. Forward Moving behavior (forward_behaviour Method)
   - This method evaluates the forward moving behavior of the robot in a straight line and rewards it once it achieves the desired behavior, ensuring efficient navigation.
   - It calculates an average forward speed and assigns a movement score to encourage forward motion.
   - Using sensor data and predefined deviation thresholds, it measures path deviation from the center line.
   - When the robot moves in straight line, it is rewarded with a higher score else it is penalized for varying behavior.

7. Avoid Collision Zones (avoid_collision_zones Method)
   - This method helps to avoid collisions for the robot using proximity sensor readings.
   - Using predefined threshold values, it defines the exploration space as safe, warning, and danger.
   - All sensor values are checked and higher fitness scores assigned in safe zones and penalties are applied in warning or danger zones.
   - This fitness score allows robot to maintain safe distances from arena boundary and reduce collisions while navigating.

8. Avoid Spinning Behavior (avoid_spinning_behaviour Method)
   - This method calculates a spinning fitness score to prevent excessive spinning.
   - It calculates a score based on how close the two velocities
   - The goal is to minimize significant speed differences between the 2 motors that will lead to spinning behavior.
   - The resulting spinning fitness minimizes the velocity difference between the two motors and ensures smooth movement through the arena.

9. Junction Detection (junction_decision Method)
   - This method determines the turn robot will take after reaching the junction.

- If a light beam is detected and robot has reached the junction, the robot decides to turn right
- In all other cases, the robot has the default behavior to turn left after reaching the junction.

10. Calculate Fitness (calculate_fitness Method)
- This method computes robot fitness by combining fitness values of forward behavior, collision zones, spinning and junction detection functions.
- Specific weights are given to the above fitness sources to encourage the desired behavior.
- As the function heavily punishes the undesirable behavior, we have noticed highly negative values for fitness which get closer to zero (will never be equal to zero) for desired behaviors.
- With increased number of generations, the fitness values will be greater than 0 and move to positive values.
- The combined fitness is then averaged over multiple iterations to provide an overall fitness value for the robot.

11. Run Robot (run_robot Method)
- This method is the main loop of the robot controller.
- It reads the sensor values, normalizes them and applies the MLP to obtain the motor velocity outputs.
- It calculates the fitness of the robot for the current iteration.
- It also applies the weight updates sent across by the supervisor.

Supervisor Controller:

Genetic Algorithm (GA) is a population-based algorithm that mimics real world population reproduction to evolve solutions. It selects the fittest individuals, applies crossover and mutation, and evaluates their fitness to generate better solutions over generations. The GA iteratively refines the population by combining and modifying individuals to approach optimal solutions. Supervisor class runs a genetic algorithm to optimize the robot performance. It manages the robot populations, applies genetic operations (crossover, mutation) and tracks fitness over multiple generations to arrive at the most optimal solution. We have used the same template available as part of Lab 4 in class lab exercises.

1. Main Program (__main__)
- Creates a supervisor object and manages the simulation.
- It provides a character text user interface, allowing user to input “S” to initiate the simulation and “R” to run the best individual solution.

2. Initialization (__init__ Method)
- GA Hyperparameters: Sets the number of generations, population size and elite size.
- Controller communication: Enables communication between supervisor and controller.

3. Evaluate Genotype (evaluate_genotype Method)
- The robot performs 2 tasks, Task A – to navigate to left and Task B – to navigate to right.
- In each task, robot fitness is calculated based on its proximity to the designated reward zone.
- The method runs 5 iterations, calculates the average fitness across trials and stores the result for current generation.
- These results are then utilized to calculate overall fitness.

IV. Results And Analysis

A. Task 1- BBR:

TABLE I
BEST RESULTS FOR BBR TASKS

Maze/Runs	1	2	3	Average Time
T-Maze A	14.72	14.56	14.59	14.62
T-Maze B	14.49	14.47	14.55	14.50

We did a lot of experimenting with different values and finally found the above table as our best result with the attributes max_speed = 5.5 and adjustment_factor = 0.05.
All our experiments for BBR can be found in the Appendix (A. Various experiments done on BBR).
Even though some other combinations of max speed and adjustment factor gave us slightly shorter duration, increasing speed more than this would end up colliding with the walls and altering the adjustment factor would make the behavior of the robot less smooth. We also see in **Fig. 3** that initially when we increase the speed there is a steep decline in duration but slowly it comes to a plateau.

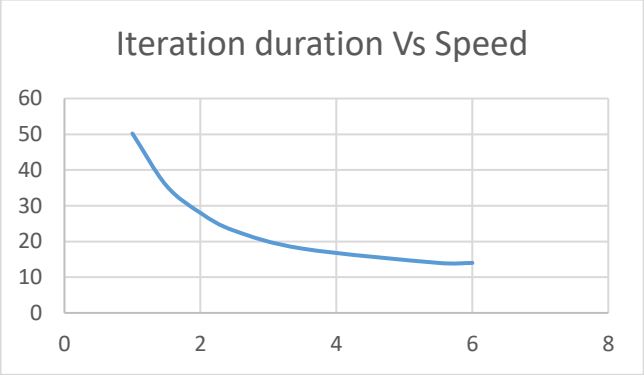

Fig. 3. Average Iteration duration Vs Speed

As we can see, the duration remains almost the same throughout all iterations for each task. This is because there is no learning as such. The robot behaves according to the rules provided to

it. Even for the dynamic changes in the environment such as light on or off, the robot is prepared and already knows how to react to these changes.

B. Task 2- ER:

TABLE II
BEST RESULTS FOR ER TASKS

Routes/Runs	1	2	3	Average Time
T-Maze A	66	62	66	64
T-Maze B	68	66	70	68

We experimented extensively with the ER implementation by modifying the hyper parameter values of the GA algorithm including number of generations, population size and elite size. We finally were able to achieve acceptable results for a neural network structure of [12,4,2] with good performance observed from 6th generation onwards and noticing constant fitness results generation 13 onwards. These experiments were quite exhaustive and time consuming as we had to re-run all the generations once any parameter values were updated.

Fig. 4. Average Iteration duration for ER

V. DISCUSSIONS AND CONCLUSION

While working on the coursework problem statement, we explored Behavior-Based Robotics (BBR) and Evolutionary Robotics (ER) approaches and understood their characteristics, advantages, and limitations in solving the T-maze problem. Eventually the e-puck robot did navigate the maze using both approaches, but their performance varied significantly in terms of adaptability, efficiency, and robustness.

We observed that Behavior-based systems consist of multiple interacting behaviors that respond to the environment. Since these systems involve numerous behavioral layers and varying probabilities of execution, it is very difficult to determine the set of actions that led to a particular behavior, as several behaviors operate together and switch rapidly between them.

Evolutionary Robotics (ER), using genetic algorithms and neural networks, outperforms Behavior-Based Robotics (BBR) in adaptability and efficiency and does not require hardcoding of desired behavior. Success of Evolutionary Robotics (ER) relies on iterative learning and a well-designed fitness function. This fitness function can be modularized and weighted to promote self-organization.

To conclude, while BBR excels in design simplicity, ER proves superior in adaptability and scalability for the T-maze problem. BBR is best suited for static environments requiring pre-deterministic behavior, whereas ER's learning-based approach is ideal for dynamic environments with changing objectives [9].

VI. APPENDIX

A. Various experiments done on BBR

1.max_speed=1.0,self_adjustment=0.01

Maze/Runs	1	2	3
T-Maze A	53.8106947	53.1474633	52.3547177
T-Maze B	54.1196628	51.8190284	53.0537069

2. max_speed = 1.5, self_adjustment = 0.01

Maze/Runs	1	2	3
T-Maze A	35.874436	36.076322	35.488528
T-Maze B	35.337594	35.712757	35.616911

3. max_speed = 2.0, self_adjustment = 0.01

Maze/Runs	1	2	3
T-Maze A	28.116334	28.572119	27.981376
T-Maze B	29.190457	28.63497	28.377854

4. max_speed = 2.5 self_adjustment = 0.01

Maze/Runs	1	2	3
T-Maze A	23.91424	24.05114	24.142028
T-Maze B	24.263215	23.643928	23.696074

5. max_speed = 3.5 self_adjustment = 0.01

Maze/Runs	1	2	3
T-Maze A	18.984703	19.233768	18.892715
T-Maze B	18.955543	18.455786	18.381334

6. max_speed = 3.5 self_adjustment = 0.01

Maze/Runs	1	2	3
T-Maze A	19.001433	19.110776	18.865551
T-Maze B	18.804736	18.63873	18.715331

7. max_speed = 5.5 self_adjustment = 0.01

Comment: Hits the wall

Maze/Runs	1	2	3
T-Maze A	14.387729	14.383391	13.863024
T-Maze B	14.211129	14.240166	13.706486

8. max_speed = 5.5 self_adjustment = 0.05

Comment: Does not hit the wall

Maze/Runs	1	2	3
T-Maze A	14.637635	14.637635	13.863024
T-Maze B	14.566646	14.628354	14.988173

9. max_speed = 5.5 self_adjustment = 0.08

Comment: Nothing changed

Maze/Runs	1	2	3
T-Maze A	14.357086	14.27125	14.003718
T-Maze B	14.315139	14.262018	14.139198

10. max_speed = 5.5 self_adjustment = 0.09

Comment: Robot gets stuck at the junction

11. max_speed = 5.5 self_adjustment = 0.03

Comment: Robot is wobbly

12. max_speed = 6.0 self_adjustment = 0.05

Comment: Does not hit the wall

Maze/Runs	1	2	3
T-Maze A	14.268292	14.171305	14.276327
T-Maze B	14.238343	14.085819	13.978302

B. GIT Repository Details

https://github.com/Arif-Ali-H00484788/F21RO-Coursework-PG-Group-7

VII. References

[1] Slušný, S., Neruda, R., & Vidnerová, P. (2010). Comparison of behavior-based and planning techniques on the small robot maze exploration problem. Neural Networks, 23(4), 560-567. https://doi.org/10.1016/j.neunet.2010.02.001

[2] M. Wahde, "Behavior-based robotics," in INTRODUCTION TO AUTONOMOUS ROBOTS. CHALMERS UNIVERSITY OF TECHNOLOGY, 2007.

[3] "Behavior-Based 'Bottom-Up' Robots," Ilstu.edu, 2024. https://mind.ilstu.edu/curriculum/medical_robotics/behavior1.html (accessed Nov. 28, 2024).

[4] Doncieux, S., Bredeche, N., Mouret, J., & Eiben, A. E. (2015). Evolutionary Robotics: What, Why, and Where to. Frontiers in Robotics and AI, 2, 126753. https://doi.org/10.3389/frobt.2015.00004

[5] O. Kramer, *Genetic Algorithm Essentials*. Cham Springer International Publishing, 2017.

[6] H. Batatia, Class Lecture, Topic: "Genetic Algorithm and Neuroevolution." F21BC, School of Mathematical and Computer Sciences, Heriot-Watt University Dubai Campus, Dubai, Oct., 10, 2023.

[7] M. Kumar, D. M. Husain, N. Upreti, and D. Gupta, "Genetic algorithm: Review and application," Available at SSRN 3529843, 2010.

[8] J. Pettersson, Generation and organization of behaviors for autonomous robots. Chalmers University of Technology, 2006.

[9] M. Wahde, "Evolutionary robotics," The use of Artificial Evolution in Robotics, Tutorial presented at IROS, 2004.