F20RO Robotics Coursework

(Raymond) Pyra Firkins

Final Word Count: [insert] (between 2500 and 3000)

# Introduction

This is a report for the F20RO coursework comparing and discussing implementations of BBR and ER controllers on an ePuck robot in a T-maze. The final results were somewhat inconclusive due to incorrectly-working ER fitness rewards, but general indications showed that [ER was faster than BBR by a factor of], after the ER controller was run for 50 generations.

# Task 1 Methods & Implementation Rationale

The controller for Task 1 heavily relies on the proximity sensors to dictate when the ePuck turns in a specified direction. The direction is specified with the ground sensor detecting a value of less than 500 (half its maximum), which only occurs when driving over the black hint square. The duration it spins for is around 1 in-World second, so not dependant on time, but rather the speed it spins at was manually determined with trial and error by just using basic use of “self.velocity\_left/right = “ statements. This implementation meant that the controller could operate without needing a Supervisor to send/receive information such as location/rotation direction.

A picture containing graphical user interface

Description automatically generated

# Task 1 Results & Analysis

The results of the Task1Controller are consistent and predictable, as it is essentially a hard-wired controller to work with this exact scenario. This does mean it only works for this maze and that any other layout will cause the ePuck to not work, even just swapping which side the black square hints at would cause the controller to give the exact reverse-from-desired results.

The largest inefficiency with this implementation is the time delay when turning and when halting the controller, both from the same cause. There seems to be a constant ~2.3 second delay between hitting a wall and stopping, to realising its hit said wall and actually turning or halting operation. The reason for this is unclear, as in other controllers there is no such delay for this long a duration. Depending on the application of this robot, it would either be an insignificant issue or a large one; if time was a key factor in its performance metric, then these two 2.3 second delays combine to around 4.6 seconds, which is almost 37% of the total average runtime of the robot. However, if only consistent results matter, and the time taken is irrelevant, then this is a minor issue that could be generally ignored.

The orientation of the ePuck’s initial rotation also further shows the controller’s reliance on the exact same circumstances to work properly, as it only reaches the goal when the robot is facing completely forwards. Having the robot face any direction in which it won’t reach the top of the maze without hitting something first and it won’t reach either end goal.

# Task 2 Methods & Implementation Rationale

The fitness functions in the controller specify fitness changes based around moving forward, spinning, moving backwards, and avoiding obstacles. Moving forward with any speed for both motors gives a +2 fitness, to discourage spinning on the spot of slamming backwards into walls, which itself is punished with backwardsFitness. Spinning on the spot in either direction, detected by both wheels moving in opposite directions at any speeds, nets a -2 to the fitness.

An avoidCollisionFitness started was provided in the template, but it was found that at times the black square would be confused with a wall or some other collision obstacle, so an avoidCollision fitness function was not implemented in the final build.

The supervisor is used to provide reward incentives based on the ePuck’s final location, both with a general gradient value for any x/z-value and for specific reward values for being in certain “zones”. The ePuck remaining in the starting “zone” gives a flat -3 fitness, to try to encourage exploration and further discourage spinning or running into a corner, and +3 for being in the top 1/3 of the maze. For the sideways paths, if a hint is detected it’ll reward for being on the right, and vice versa for when no hint detected.

Text

Description automatically generated

To tell the supervisor that a hint’s been detected, the emitter was altered to send a hintDetected variable instead of a “fitness:” string. Placing a “t” or “n” to indicate true/false was tested but the supervisor receiver wouldn’t work properly with the extended initial string, so the message had to be sent using the same length of characters before the fitness value itself. These alterations allow a reward based on the square hint to be created inside the supervisor. Otherwise, the ePuck would need to be aware of its own location, using either the GPS module which would modify the template World, or by the supervisor sending the location to the controller, which seemed to be a more difficult task.

Text

Description automatically generated

Text

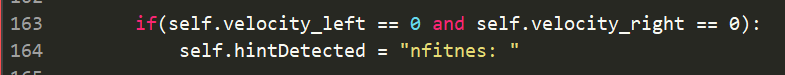
Description automatically generated

Text

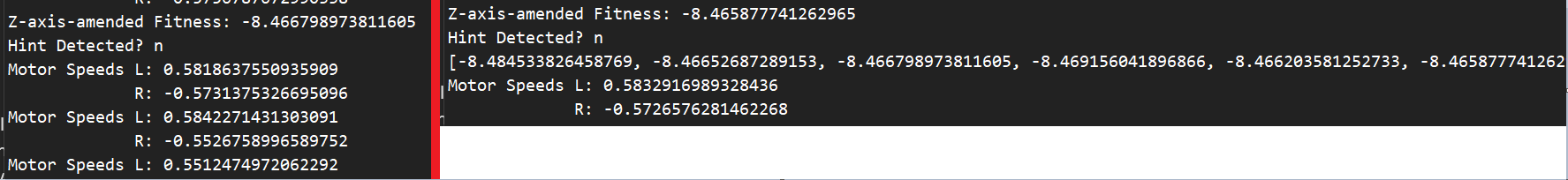
Description automatically generated

The next issue was resetting the hintDetected value back to false after a given run. Without this function the ePuck will consistently reach an end goal, but which side is not dependant on the hint. Some tests evolved to randomly prefer left or right in a way that coincided with being correct, but this occurred randomly and entirely be chance.

[assuming I cant fix hintDetectedReset] Ultimately the issue of resetting hintDetected correctly wasn’t fixed. Several different attempts were made by placing a hintDetected = “nfitnes: “statement in different locations in the controller but the correct location/prerequisite to run wasn’t found.



The currently implemented-yet-failing attempt involves detecting when the ePuck is no longer moving, namely in between iterations, and then resetting hintDetected’s value. Based on the Supervisor.resetPhysics documentation [2], which states node inertia and velocities are all set to zero, it was thought when this is run before a new trial would begin that the ePuck velocity would be zero for at least a fraction of a second, but this was never found to be the case. The image below shows some example initial velocities after resetPhysics is run for a new run of the maze.



The crossover rate and mutation percentages were set to 75 and 90% respectively, which are both rather high but were thought necessary. This was done to allow for fast evolution and wide experimentation on the GA’s part without being entirely random, also since early versions of code using lower values would take an unfeasibly long time to make any progress.

# An odd issue that was encountered was the black square seemingly changing how the robot acted, despite the code being affected by the square was not functional. The square would occasionally cause the robot to turn abruptly into a wall and halt or slow down considerably. The best guess as to why that would occur is that the square is ever so slightly raised above the floor of the map to be visible by the ePuck’s ground sensors. If the ePuck is driving very slowly, as does tend to happen, the raised square could be causing the robot to get stuck temporarily, and since it’d be arriving at an angle the trajectory of the robot is changed compared to if there was no hint square.

# Task 2 Results & Analysis

Task 2 results are less conclusive than Task 1’s, due to the hintDetected code not functioning properly. To combat this and still have a resulting robot prefer reaching either end-point, the x-axis rewards were run regardless of if hintDetected was set to the correct value.

The current controller was ran for 50 generations, with populations of 4 and an elite number of 6. This was repeated 3 times, and the average fitness of each generation was recorded and plotted below. A note with the data is that due to each generation’s fitness being averaged, there can be discrepancies in a given generation, for example in the first run, generation 42 has an average fitness of 6.622, but each individual fitness were as follows: 14.520621308901049, 14.480829371237183, 8.960601062317586, 8.096377996107266, -12.947647859499101. So while some runs had amazing fitnesses, and the pattern of that generation was a very good fitness, the final -12.948 dragged the average down. However plotting every single data point seemed excessive and too time-consuming/messy, and the mean was still thought to be a good indication of how the GA was performing overall.

Chart, line chart

Description automatically generated

An interesting note is a rapid increase in fitness around generations 18 to 22 for all 3 runs. The reason for this is unknown, but does mean that one can expect a positive fitness to be the norm by generation 25 on a consistent basis.

# Discussion & Conclusion

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Time | | | |
|  | First run | Second run | Third run | Average time |
| Task 1  T-Maze A | 12.48775 seconds | 12.512437 seconds | 12.4641 seconds | 12.488096 seconds |
| Task 2  T-Maze A | 12.458922 seconds | 12.53051 seconds | 12.473162 seconds | 12.487531 seconds |
| Task 1  T-Maze B |  |  |  |  |
| Task 2  T-Maze B |  |  |  |  |

[discussion of results]

Behaviour-Based Robotics (BBR) was found to lead to a robot that was predictable and consistent, yet restricted to only work on this specific set of circumstances, while Evolutionary Robotics (ER) lead to a robot that would work in varying situations, but could either find a bad result to focus on, or take an entirely unknown length of time to reach the desired result. This is also dependant on how well the fitness functions are written, if like in this case the fitness was not tuned to work 100% correctly, it is much more likely an alternate-and-incorrect result would be found as the conclusion in its view.

Both have their pros and cons depending on the scenario: for example if BBR code had to be written for multiple maze types or layouts then that method would have become more inefficient, either relying on multiple separate controllers for each maze, or a single controller with an ever-growing number of “if/else if/else” statements, making code hard to read and understand.

# References

[insert]

<https://cyberbotics.com/doc/reference/supervisor#wb_supervisor_node_get_field> [Accessed 27/11/2022]

[2] <https://cyberbotics.com/doc/reference/supervisor?tab-language=python#wb_supervisor_node_reset_physics>

Text, letter

Description automatically generated