Movement control of a robot through an unknown closed circuit

João Santos, 76912[1][0000−0003−0393−6456]

Department of Electronics, Telecommunications and Informatics
University of Aveiro, Portugal

Abstract. In the present assignment, the challenge was to develop a robotic agent capable of going around a unknown track/maze, as fast as possible and without colliding with the walls. The agent must, also, not go backwards as that provides very few points. The best implementation was through the use of a Proportional–Integral–Derivative controller (PID) where a set of parameters were found using a genetic algorithm.

Keywords: robot, navigation, genetic algorithm, simulation, path

1 Introduction

The ability to navigate in a given environment is critical for any mobile device. The robot must first avoid risky scenarios such as collisions and unsafe conditions, but if it has a mission that requires it to visit particular locations in the robot environment, it must do so.

In the proposed assignment, a robotic agent was developed that could, autonomously and as fast as possible, go around a given map. This maps have one and only one closed loop for the robot to travel within. It must avoid collisions with the walls as that deducts points from the overall score while, also, travel in the defined forward direction (going backwards is possible, but points are awarded not every cell traveled but just every completed lap).

2 Implementation

The developed control system resorts on a PID to control the steering of the robot. A PID is a feedback-based control loop mechanism which calculates an error value as the difference between a desired set-point and a measured process variable, on a continuous basis. The PID then outputs a manipulated variable MV relying on proportional, integral, and derivative terms (denoted P, I, and D, respectively), thus the name.

In general terms, the PID control system is given by

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau) d\tau + K_D \frac{de(t)}{dt} \tag{1}$$

where $e(t)$ is the error and K_P, K_I and K_D are, respectively, the gains for the proportional, integral and derivative components.

With these three gains, the PID can control the output variable with the past, present and future values of the error, like so:

– Proportional: Acts upon the current value of the error. The greater the error, the greater the controller output.
– Integral: Takes into account previous values of the error. If there is a residual error after the proportional component, the integral term attempts to eradicate it by introducing a control effect based on the historic cumulative value of the error.
– Proportional: Is the best estimate for the future direction of the error. By exerting a control influence generated by the rate of error change, it is essentially attempting to reduce the effect of the error. The faster the change, the greater the control or damping effect.

This lead us to the necessity of formulating the variable to be measured and what value we want it be have.

Given that the author can freely place the four Infra Red (IR) distance senors at any angle, the selected methodology consists on choosing two values, α and β, between 0 and 180 degrees and then placing the sensors in symmetric pairs, i.e., at $[\alpha, \beta, -\alpha, -\beta]$ degrees. This decision was taken so that an over-fit to the default map would not occur (or at least, not as easily).

With this, and assuming that the desired path for the agent is the one that leads it to be equidistant of the surrounding walls, the process variable can be computes as in equation 2, where w_i and m_i are, respectively, the weight given to and the measure of the IR sensor positioned at i degrees. When this assumption happens to be true, in a scenario without noise in the measurements, the value of each sensor on each pair would cancel each other out, and so this formulation leads to a set-point $SP = 0$.

$$PV = \sum_{i \in [\alpha, \beta, -\alpha, -\beta]} w_i \cdot m_i \tag{2}$$

Notice that, to maintain the desired symmetry, $w_i = -w_{-i}, i \in [\alpha, \beta]$.

2.1 Motion Model

The general speed of the robot can be decomposed in the linear (lin) and rotational (rot) components, which are given by

$$lin = \frac{out_R + out_L}{2} \tag{3}$$

and

$$rot = \frac{out_R - out_L}{D} \tag{4}$$

where out_R and out_L are, respectively, the output speeds for the right and left wheels and D is the diameter of the robot, which is unitary.

Notice that, since we define the output of the PID as the steering control (i.e. $rot = MV$), a positive value in the output means that $out_R > out_L$ (left turn). In the same way, a negative value in the PID output means a right turn.

All of this leads to the linear system of equation 5, that makes it possible to compute the output velocities for both wheels with only the output of the PID and by fixing a base linear speed.

$$\begin{bmatrix} 2lin \\ MV \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} out_R \\ out_L \end{bmatrix} \tag{5}$$

Again, in an ideal scenario with no noise, the output of the PID should be null on straights and different of zero on corners. To try to take advantage of this, and try to slowdown the robot on the corners and speed it up on the straights, a deduction factor (equation 6) was added to the linear system. This deduction factor is based on the last N outputs of the PID (i.e. last N rotations) and also has an associated weight so its influence in the overall velocity can be fine tuned.

$$d = w_d \frac{1}{N} \sum_{i=-1}^{-N} MV_i \tag{6}$$

Adding it all together, equation 7 outputs the desired values for each wheel.

$$\begin{bmatrix} out_R \\ out_L \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} 2lin \\ MV \end{bmatrix} + \begin{bmatrix} d \\ d \end{bmatrix} \tag{7}$$

2.2 Parameter Tuning

The formulation presented up to this point lets us with ten unknown parameters $[lin\ K_P\ K_I\ K_D\ \alpha\ \beta\ w_\alpha\ w_\beta\ w_d\ N]$.

The chosen method to obtain the optimal set of parameters was through the implementation of a genetic algorithm.

A genetic algorithm is a method based on natural selection, the mechanism that drives biological evolution, for addressing both constrained and unconstrained optimization problems. A population of individual solutions is repeatedly modified by the genetic algorithm. At each generation, the algorithm selects parents at random from the current population and uses them to produce the children in the next generation. The population evolves toward an optimal solution over the generations.

At each generation, the genetic algorithm has three main steps:

1. Selection: performs the selection of individuals (called parents) based on their fitness score, that will contribute to the next generation.
2. Crossover: performs the combination between two parents to form the children of the next generation.
3. Mutation: applies random mutations to the children.

As already mentioned, each individual must have a fitness score. This score is a value that will be used to compare it to all the others. The higher the fitness, the better.

Typically, an individual with higher fitness has a greater change to be select as a parent than an individual with lower fitness. Notice that this does not prevent an individual with a low fitness to be selected. This is useful to maintain a diversified gene pool that together with the random mutation can, in the long run, generate a better individual.

In the proposed implementation, the fitness is the score (provided by the simulator and following the rules in it) that each individual can obtain in a fixed amount of time.

The PyGAD [1] library was used as a basis for the genetic algorithm. It offers a base form that can be customized to fit any given optimization problem. Table 1 demonstrates the parameters used while table 2 specifies the range and type for each gene.

Table 1. Genetic parameter used in the evolutionary algorithm.

Number generations	500
Solutions per population	100
Number parents mating	25
Number of parents to keep	25
Number of genes	10
Parent selection type	Steady-state selection
Crossover type	Uniform
Mutation type	Random

Table 2. Gene space for any given individual.

	Low	High	Data Type
lin	0	1	float
K_P	$-\infty$	$+\infty$	float
K_I	$-\infty$	$+\infty$	float
K_D	$-\infty$	$+\infty$	float
α	0	180	integer
β	0	180	integer
w_α	$-\infty$	$+\infty$	float
w_β	$-\infty$	$+\infty$	float
w_d	$-\infty$	$+\infty$	float
N	1	3	int

2.3 Alternative Approaches

Additionally to the implementation described earlier, a couple of different ways were tested, yet they provided comparably worse results, maybe due too the inexperience of the author with both.

The first that we will discuss is the use of neural networks. The PyGAD library also provides a module that allows for a generic network to be "trained" using the genetic algorithm. In the attempted implementation, each chromosome was composed by two parts: i) the "optimal" angles for the IR sensors and ii) the weights of each neuron in the network. The network would have four neurons in the input layer (one for each IR sensor measured value) and two neurons on the output layer (the speed of each wheel). Multiple architectures were tested (no hidden layers, a single hidden layer with three or four neurons, two hidden layers with (5,3), (4,2), (3,3) neurons on each). All resulted in worse scores than the PID implementation.

The second alternative attempt consisted on the usage of the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [3]. In essence, the goal of this algorithm is to provide a "genetically evolved" neural network capable of solving a given problem. Yet, NEAT not only optimizes the weight and bias of each neuron but can, also, add/remove neurons and/or connections between them. In particular, the used library, neat-python [2], even allows for the activation function (among other properties) to also evolve along the generations.

2.4 Known Issues

Two main possible issues were detected on the developed agent.

On some corners, specially the hairpins, the robot tends to pass very close of the inner wall. Despite the fact that no collision was seen, and it is a faster path, it certainly comes with some dangers.

Similar to this on some of the ninety degree corners, as overshot may occur that lead the robot towards the outside wall. As previously, the robot is usually able to slowdown and correct the trajectory, but is also a possible collision situation.

3 Results

During the run of the genetic algorithm, each individual in the population was evaluated by letting it try to obtain the highest score possible around the C1 lab provided by the described *ciberRatoTools* environment. The simulation, for each individual, was over when either one thousand steps of the simulation would be completed or a wall was hit. This fitness function provided a pressure for the agents to be as fast as possible without colliding, which was the main goal of the present assignment. To build a more robust agent, the training environment had higher noise levels then the evaluation one. This option was taken after the evolved agents with the standard noise values, in certain circumstances, would hit the wall. This behaviour drastically decreased after the agents were tested in a noisier environment.

Figure 1 shows the evolution of the score along the generations. It very clearly shows that, by pure chance, very early a big leap in fitness score was achieved that took the best solution too obtain a score of almost six hundred points (i.e. the robot traveled almost sixty cells within the stipulated time). Over time, the fitness floated, but tended towards a stagnation around the 680 value.

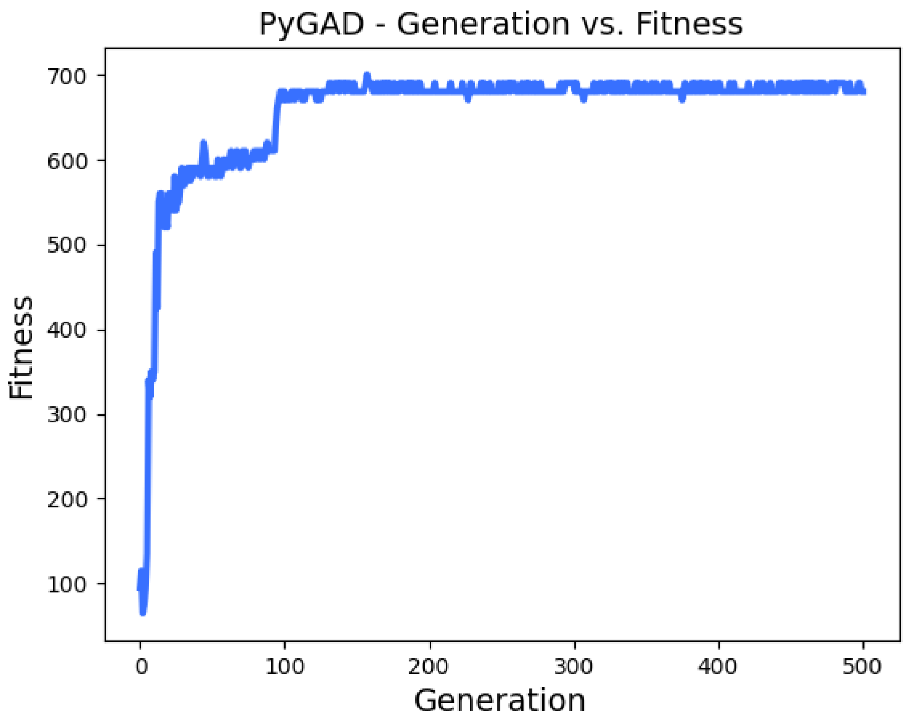

Fig. 1. Evolution of the fitness value of the best individual in each generation.

After all the generations, the best solutions was

$$\begin{bmatrix} lin \\ K_P \\ K_I \\ K_D \\ \alpha \\ \beta \\ w_\alpha \\ w_\beta \\ w_d \\ N \end{bmatrix} = \begin{bmatrix} 0.189068 \\ 7.740097 \\ 0.681015 \\ -0.007488 \\ 36 \\ 42 \\ 0.851787 \\ 0.758472 \\ 0.480972 \\ 1 \end{bmatrix}$$

and the ability to reach a score of 3540 when allowed to run for the full five thousand simulator steps.

Two particularities caught the attention of the author:

1. although the initial idea was for w_α and w_β to provide a weighted average, the fact is that, on the best solution, $w_\alpha + w_\beta > 1$;
2. the best solution "opted" to use as little knowledge about the past as possible ($N = 1$), but it was still useful since it gave it a non-zero weight w_d.

Finally, a couple of videos are provided of the developed agent running on the C1 lab and on another one developed by the author:

– https://youtu.be/jJx0Y6EDXHQ
– https://youtu.be/Wgdqme6gSD8

4 Conclusion

The developed agent is, has intended, able to move around the map without colliding with the walls and so, the major goal of this assignment has been accomplished. The author acknowledges that faster ways could exist, but given the tries he made, the PID ended up being the most reliable one.

References

1. Gad, A.F.: Pygad: An intuitive genetic algorithm python library (2021)
2. McIntyre, A., Kallada, M., Miguel, C.G., da Silva, C.F.: neat-python. https://github.com/CodeReclaimers/neat-python
3. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation **10**(2), 99–127 (2002), http://nn.cs.utexas.edu/?stanley:ec02