Evolutionary Robotics Simulator

Quintana Pelayo, Guillermo; Herbert, Attila; Kok, Levi
AI Master Students
Data Science and Knowledge Engineering
Maastricht University
g.quintanapelayo, a.herbert, ll.kok
@student.maastrichtuniversity.nl

March 2020

Abstract

Simulators play a critical role in robotics research as one can test out new strategies and algorithms with relative ease. In this paper, we present a solution to the dust collecting robot problem. To train the robot, we used an evolutionary algorithm on a neural network. We arrived at solutions that can navigate without too many collisions and can collect dust. This project is part of the Autonomous Robotic Systems course from the Faculty of Data Science and Knowledge Engineering Master program in Artificial Intelligence of the Maastricht University.

Keywords: *robot, simulator, evolutionary algorithms, agent*

1 Introduction

The main objective of this project is to create an algorithm that can produce an agent that can successfully navigate a map and collect as much dust as possible. To train the agent, we utilize an evolutionary algorithm to select and evolve the weights of its neural network. Agents were tested on a variety of environments with different complexity.
Secondary objectives focus on improving the navigation of the agent, such as minimizing the number of collisions with the walls and increasing the speed of the robot.

2 Implementation

This section contains all the steps followed to build the simulator and the agent. From the very beginning we tried to maintain a structured implementation so that it's easy to add new features in the future and also aimed to make the simulator as light as possible when it comes to CPU resource consumption.

The language used for this project is Python 3.7 with the help of the library pygame [1]. This module was only used for window management, user input and to draw the graphics. [1]

2.1 Robot and Dust

The robot itself is visualised by a white circle and a black line that represents the directions that the robot is facing. It has two independent wheels and 12 sensors (see section 2.2). The Robot class contains the whole model of the robot, including movement and collision handling.

The data on the dust consists of a Boolean matrix that represents whether it has a dust particle or not. Each dust particle encloses a space of 10 by 10 pixels. We compute each dust particle is located within the circular area of the robot, by the following equation

$$(x_{dust} - x_{robot})^2 + (y_{dust} - y_{robot})^2 \leq r_{robot}^2$$

These dust particles are then visualised as grey circles. Each frame, the dust particle's collision with the robot are checked. The first part of our fitness function $f(agent)$ is the total sum of the boolean matrix, which is equal to the amount of dust particles remaining, N_{dust}. The second part is a penalty factor:

[1] Code can be found at `https://github.com/GuilleQP/ARS_assignments`

the number of times the robot collides with the wall, $N_{collisions}$, times 20.

$$f(agent) = \sum^{n} N_{dust} + 20 \sum^{n} N_{collisions}$$

Therefore, for our fitness function holds less is better.

2.2 Sensors

The robot has 12 distance sensors with a 30°angle between them. Their maximum range is 200 pixels. As we will observe later, when a wall is detected by a sensor, it changes its color from blue to orange and updates the ending point to match the intersection with the wall so that the sensor line doesn't overlap the obstacles in the map. Furthermore, every sensor displays a label with the current distance to the closest obstacle within the range.

2.3 Distance Algorithm

The algorithm that determines the distance for each sensor works as follows. Each wall is a line segment consisting of two X,Y coordinates (the edges). We construct a sensor as another line segment. We do this by taking the robots center position as the first coordinate, and the second coordinate on the sensors line, determined by the sensor angle.

$$x_{sensorpoint} = x_{center} + \cos(\theta)$$
$$y_{sensorpoint} = y_{center} + \sin(\theta)$$

From these two points we can construct a line segment for the sensor. We now have a line segment for the wall and the sensor. We can compute the intersection of two line segments using Cramer's rule [2]: If we have two line segments

$$a_1x + b_1y = c_1, \ a_2x + b_2y = c_2$$

and these lines are not parallel

$$a_1b_2 - a_2b_1 \neq 0$$

Then the intersection points are computed as

$$x_s = \frac{c_1b_2 - c_2b_1}{a_1b_2 - a_2b_1}, \quad y_s = \frac{a_1c_2 - a_2c_1}{a_1b_2 - a_2b_1}.$$

We now have the point at which the sensor segment intersects with the wall. We now compute the euclidean distance between the robot and the intersection to find the sensor distance value. We have to take two things into account: there can be an intersection with multiple walls, and the sensor's line segment has no direction, it can thus also have an intersection in the opposite direction. We solve the first issue by looping over all walls, and returning the smallest distance value. We solve the second issue by checking if the found point of intersection is in the same direction as the point of the sensor from the viewpoint of the robot's center.

2.4 Collision detection

Collisions are handled in a relatively straightforward manner. If the robot is close to a wall, collision management is started. If on the next timestep, the robot would collide with a wall, we modify the timestep so it can approach the wall and hit exactly the edge. After a wall is hit, we connect the wall to the robot so it can move along the wall. In case the robot is connected to two walls, it means that it is at a corner. Corners are handled a bit differently, since the robot is either inside or outside of the corner. In the inside case, the robot's velocity is set to 0 if it isn't moving away from either wall. If it is moving away, that wall is removed. As mentioned before, we count the number of collisions the agent has, because this is a penalty factor to our fitness function. In the outside case, the robot's velocity is not modified until we are sure which way the robot can get away from the corner. In case a direction is found, the robot moves away.

We chose this kind of collision handling because of time restrictions. Although it doesn't utilize any physical rules, it works well for the simulation purposes.

2.5 Maps

All our maps have been designed with a width of 700 px and a height of 500 px. Four different maps have been designed and implemented for this simulator. The framework is built in a way that it is fairly easy to add new maps into it. Each map is saved in a text file that includes the edges of every wall. Figure 1 provides an overview of these maps. The maps are designed to have a variety in complexity, open space, corners and objects. This provides the agents with a variety of challenges. For example, an open room can be easily explored without collisions but a complex maze requires turns in every direction, going straight and being able to explore.

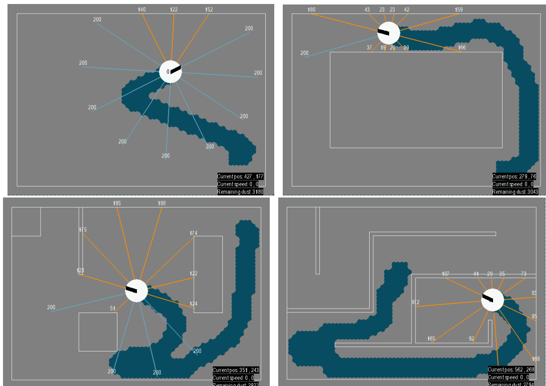

Figure 1: Four maps implemented into the simulator (clockwise, starting left top): *Rectangle*, *Double Rectangle*, *Room* and *Maze*. Gray and dark blue pixels correspond to floor with and without dust respectively. This robot was manually driven.

2.6 Neural Network

We implemented a neural network consisting of the values of the 12 distance sensors, the speed of left and right wheel of the robot, and the delayed previous activation values of the hidden layer that act as the memory. We decided upon one hidden layer with 6 nodes. We used the activation function Tanh(x), as it's output ranges within (-1,1), meaning that the robot can have the wheels go backwards and steer quickly.

The weights were randomly initialised by an uniform distribution between a range between (-0.7, 0.7). The neural network algorithm was coded by ourselves using *Numpy*.

2.7 Evolutionary Algorithms

Our evolutionary algorithm is quite simple. We use mutation with uniform distribution between (-0.5, 0.5), scaled with a mutation factor. In the experiments, we used different mutation factors ranging from 0.01 to 0.3.

We tried using crossover mutations between the weights of the neural networks of the different agents. This type of mutation performed worse. See section 3.1.

One extra feature we implemented was an inflated starting population. We noticed that robot's performance can be highly variable, and that the initial starting agent is very influential to the end result. We therefore start with 10 times the normal population, select the best starting point and start the evolution from there.

2.8 Memory

As previously discussed, we implement a memory by feeding back the activations of the hidden layer into the input layer with a delay Δt. We notice that the usage of this memory has little impact on the results. We tested the Δt for 0, 10, 25, 50, 100 and 200 timesteps. The longer the delay, the worse the results. We ended up settling on a delay of 50 timesteps, which is equal to 0.8 seconds.

2.9 Extra features

During the implementation of this simulator, we tried to introduce two improvements to our codebase. The first was multi-threading computation, this ended up being as slow as the sequential running that we were already using. Secondly we tried randomization of the starting position for every map for every agent. However this introduced too much complexity to our training. After these tests we decided to stay with the

sequential training and one fixed starting position for every map.

3 Experiments and results

Figure 2 shows an example fitness function of an agent over time. Ideally, we want to make this curve as straight down as possible, which means that the robot collects as much dust as efficiently as possible.

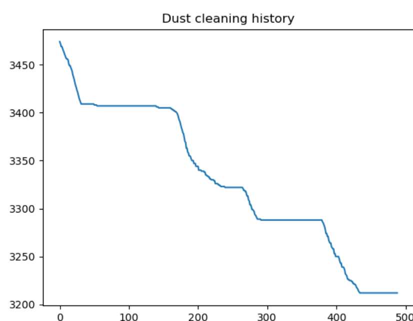

Figure 2: Dust cleaning history. The y axis represents the amount of dust remaining on the floor and the y axis are the time steps (frames).

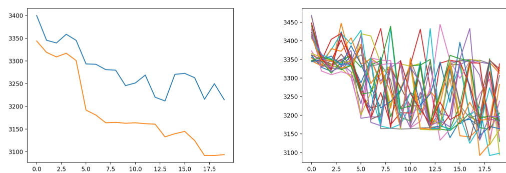

Figure 3: Development of average and best score on the left, and development of all the agent population on the right. These agents were trained on the maze map.

We tried developing a well performing agent by first training it on a simple map (double rectangle), taking the best performing agent and retraining it on a more difficult map (maze). This is similar to how a human would learn, starting easy and then increasing the difficulty. While the agent did perform well on the double rectangle, it did not manage to convert on the maze. It could make right turns very well, but never came across left turns on the double rectangle. It never managed to learn left turns well on the maze. The evaluation function over time are in the figure below.

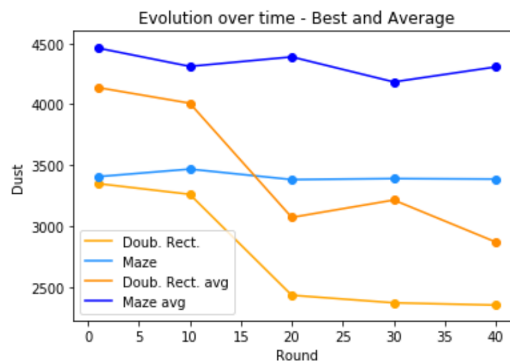

Figure 4: Development of average and best score for an agent first trained on the double rectangle map, and then on the maze. Although the agent did perform well on the double rectangle, it did not convert on the maze. Neither a high nor low mutation rate solved this problem.

3.1 Crossover

We experimented with creating mutations through the crossover techniques *One point, Uniform* and *Arithmetic*. However, the resulting agents were under-performing compared to the evolutionary algorithm using the simpler random weight mutations. The agents did not converge well. It appeared the gene pool became too similar too quickly. We did not manage to improve these results and therefore did not use crossover techniques. We believe it is hard to meaningfully represent neural network' weights as genes, which means crossovers on neural networks have low likelihood of producing better results than normal mutation. The code from this experiment can be found on the github branch *newevolution*.

3.2 Convergence

A good example of the interesting solutions that evolutionary algorithms can find can be seen in figure 5. This agent closely hugs the wall without touching it, thus avoiding collision. After 3000 frames this agent is capable of cleaning more than 72% of the available dust (see figure 6), it fails at cleaning the dust at the four corners of the map.

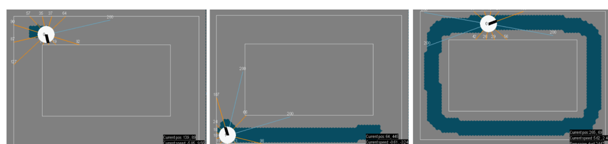

Figure 5: Agents performance for iteration 1, 10 and 20. Pool of 20 agents. Mutation rate 0.3.

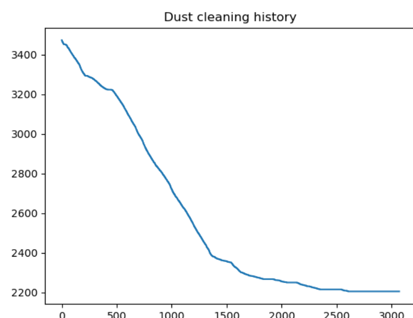

Figure 6: This agent performs well on the double rectangle map. It learns to hug the walls, but does not touch them. We tried taking this agent and retraining it on the maze. It would then need to learn to take left turns as well.

This next agent (figure 7) stops in a corner and doesn't know how to continue from there. This particular room was hard for the agents to navigate even after 100 evolution iterations.

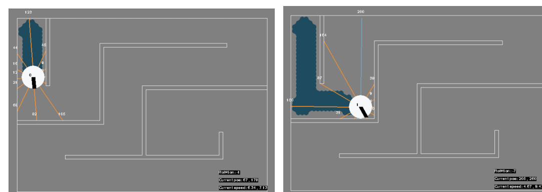

Figure 7: Agents performance for iteration 1 and 20. Pool of 40 agents. Mutation rate 0.4.

4 Conclusion

To conclude, we implemented a simulator capable of simulating a dust-collecting robot. We designed several testing rooms and implemented two different evolutionary algorithms to develop agents. Our results and observations lead us to believe that an evolutionary algorithm is not necessarily the most efficient way to develop agents, however, it's diversity is remarkable. Most other techniques require specific loss or fitness functions for every situation, but with our evolutionary algorithm, we could train agents on any of the four rooms without any changes to the algorithm or the fitness function.
One drawback of evolutionary algorithms is their resource-dependency. We couldn't afford to run a thousand agents parallel, so our agents easily got stuck at some points and their performance plateaued. This problem is however present in all optimization algorithms. Genetic algorithms require huge populations to avoid this problem, which in turn require copious resources.

References

[1] Pygame library. `https://www.pygame.org/wiki/about`, 2020.

[2] V. Dvortsov. Cramer's rule. *Atomization and Sprays - ATOMIZATION SPRAYS*, c, 01 2006.