Use of Genetic Algorithms (NEAT) in Racing AI

Made by Group 24:

Diogo Araújo: 60997

Introduction

This project was inspired by the video made by the youtuber “Code Bullet”. This video can be found in the references section of this project. It details the youtuber’s experience with creating an **Artificial Intelligence** (AI) capable of traversing a racetrack by using a reinforcement algorithm, more specifically **QLearning**.

During the video, the youtuber showcases the results obtained by the algorithm as well as the process which he took to create it. While highlighting its execution, it brought me to the question of if there were better alternatives to the algorithm that was being used. Fortunately, during one of the lectures for this class we discussed the use of **Genetic Algorithms** (GA) and their possibilities for video games. This brought me to the idea of utilizing this algorithm in search of a faster, and better performing, algorithm for the applied context. This brings us to the aim of this project, which is to use a genetic algorithm in order complete a map (track) without any issues.

Approach

In this project, I am aiming to create an AI using a **Genetic Algorithm** (GA) which can complete a lap on a racing track without any issues. However, before this implementation is discussed, we must first see the topics / research that lies behind its execution. These include genetic algorithms, and their underlying parts, as well as the actual algorithm used in the project which is the **NeuroEvolution of Augmenting Topologies** (NEAT).

A **Genetic Algorithm** (GA) is an artificial intelligence which is inspired by the process of natural selection. They are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as **mutation** and **crossover**. These are genetic operators which are used to generate a second-generation population of solutions, from those selected, through their combination. This means that for each "child" solution, a pair of "parent" solutions are selected for breeding from the pool selected previously. This causes the "child" to share many of the characteristics of its "parents". This ultimately results in the next generation population being different to the initial generation, which will increase the average fitness since only the best solutions from the first generation are selected for breeding, along with a small proportion of less fit solutions to increase diversity.

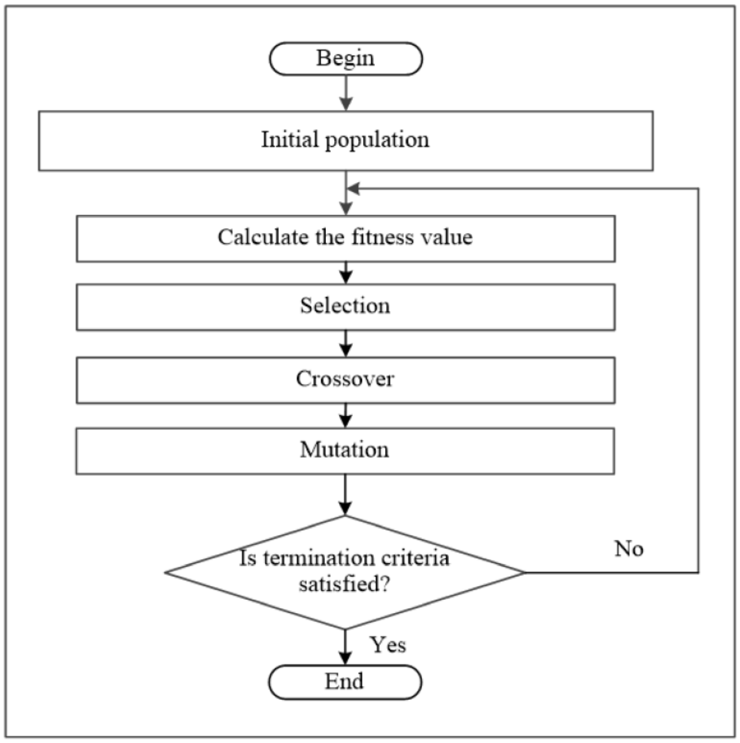

Begin
Initial population
Calculate the fitness value
Selection
Crossover
Mutation
Is termination criteria satisfied?
No
Yes
End

Image 1 – Representation of Genetic Algorithm (GA).

In the previous diagram, a few genetic operators can be seen. These are very important for the continued execution of the algorithm and bring with them the debate of their usage. Its actual usage on the algorithm created shall be seen in the next section, however, for this topic, their usage shall be seen in the most general sense:

- **Selection** is the stage of a genetic algorithm in which individual genomes are chosen from a population for later breeding. They can also be used in choosing the next population, retaining the best individuals in a generation unchanged in the next generation. This is called elitism or elitist selection.
- **Crossover** is a genetic operator used to combine the genetic information of two parents to generate new offspring. It is used to generate new solutions based on an existing population and it typically mutated after.
- **Mutation** is a genetic operator used to maintain diversity in a population. The mutation procedure usually involves defining the probability to flip a bit in a genetic sequence from its original state. Its purpose is to introduce diversity into the population, preventing the population from becoming too similar to each other which could lead to a convergence of the global optimum.

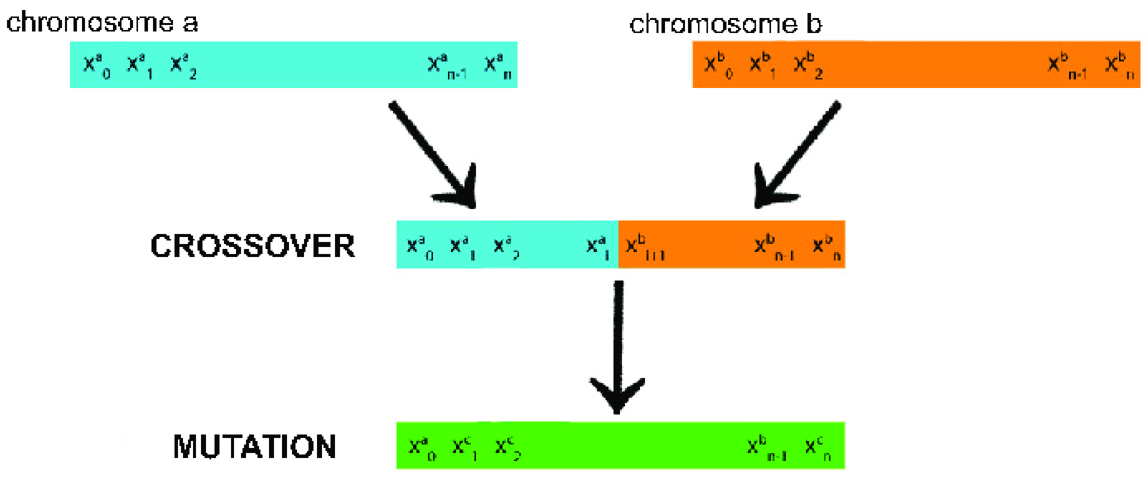
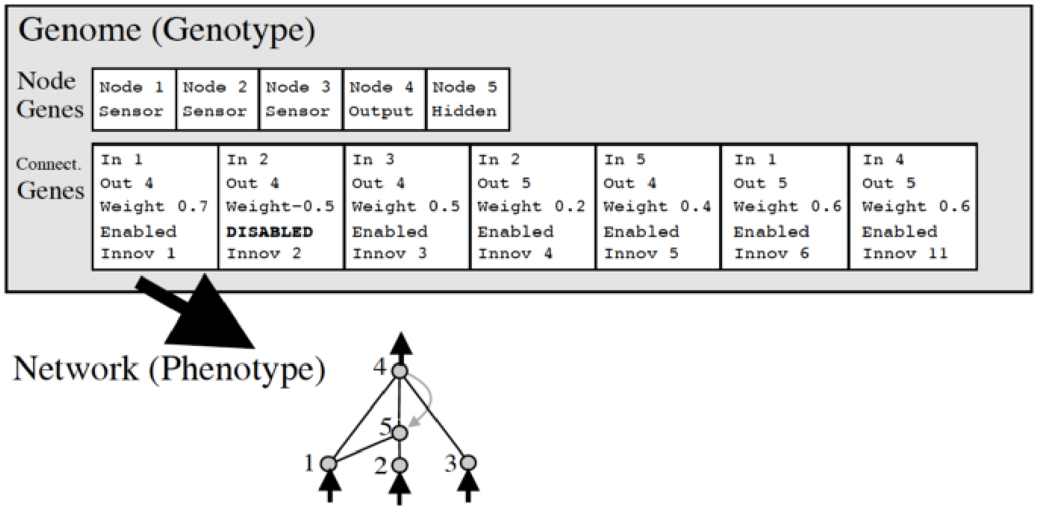


Image 2 – Representation of both Crossover and Mutation.

The actual algorithm implemented for the AI of the racing game was the **NeuroEvolution of Augmenting Topologies** (NEAT). It is a genetic algorithm used in the generation of evolving artificial neural networks (neuroevolution technique). It alters both the weighting parameters and structures of networks, attempting to find a balance between the fitness of evolved solutions and their diversity. It is based on applying three key techniques to the algorithm: tracking genes with history markers to allow crossover among topologies, applying speciation (evolution) to preserve innovations and developing topologies incrementally from simple initial structures (complexifying).

The question of encoding comes from the question of how we wish to represent individuals genetically in our algorithm. The way in which we encode our individuals lays out the path for how our algorithm will handle the key evolutionary processes: selection, mutation, and crossover. Any encoding will fall into one of two categories, direct or indirect. A **direct encoding** will explicitly specify everything about an individual. If it represents a neural network this means that each gene will directly be linked to some node, connection, or property of the network. An **indirect encoding** is the exact opposite. Instead of directly specifying what a structure may look like, indirect encodings tend to specify rules or parameters of processes for creating an individual.

Setting the rules for an indirect encoding can result in a heavy bias within the search space, therefore, it is much harder to create an indirect encoding without substantial knowledge about how the encoding will be used. The NEAT algorithm chooses a **direct encoding methodology** because of this. Their representation is a little more complex than a simple graph or binary encoding, however, it is still straightforward to understand. It simply has two lists of genes, a series of nodes and a series of connections. To see what this looks like visually, I have a picture from the original paper here:



Node 1 Sensor	Node 2 Sensor	Node 3 Sensor	Node 4 Output	Node 5 Hidden

In 1	In 2	In 3	In 2	In 5	In 1	In 4
Out 4	Out 4	Out 4	Out 5	Out 4	Out 5	Out 5
Weight 0.7	Weight-0.5	Weight 0.5	Weight 0.2	Weight 0.4	Weight 0.6	Weight 0.6
Enabled	**DISABLED**	Enabled	Enabled	Enabled	Enabled	Enabled
Innov 1	Innov 2	Innov 3	Innov 4	Innov 5	Innov 6	Innov 11

Image 3 – Representation of NEAT Algorithm (example).

Mutation, in NEAT, can either mutate existing connections or add new structure to a network. If a new connection is added between a start and end node, it is randomly assigned a weight. If a new node is added, it is placed between two nodes that are already connected. The previous connection is disabled. The previous start node is linked to the new node with the weight of the old connection and the new node is linked to the previous end node with a weight of 1.

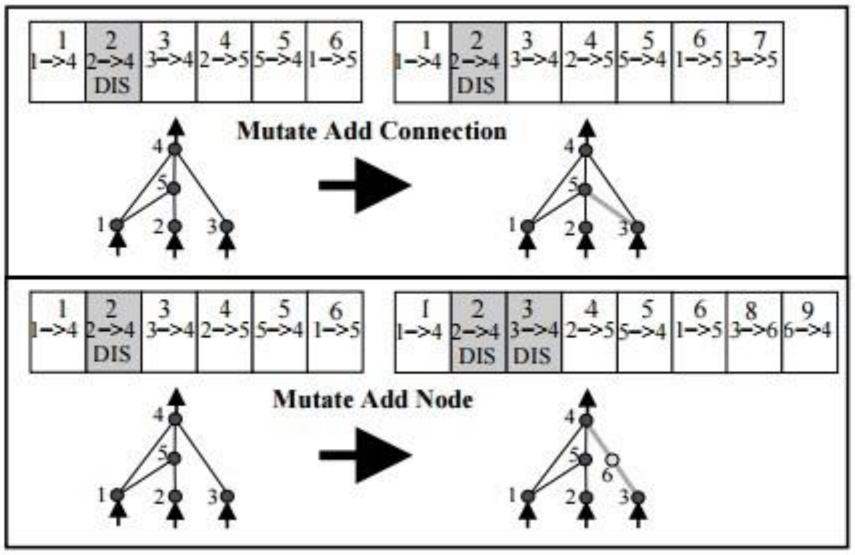


Image 4 – Mutation in NEAT Algorithm (example).

Implementation

As was stated previously, the aim of this project is to create an AI capable of traversing any given map without issues. This project is split into two different parts, which are the foundation of the algorithm, which is a game that can be played by the user, and the algorithm itself, which is an implementation of the NEAT algorithm in the game.

Implementation of the Game (Foundation)

The project started off with the creation of an initial foundation for the implementation of the genetic algorithm. This was a game, created in pygame, in which a single car would drive around a man-made track until he reached the finish line where he would then win the game.

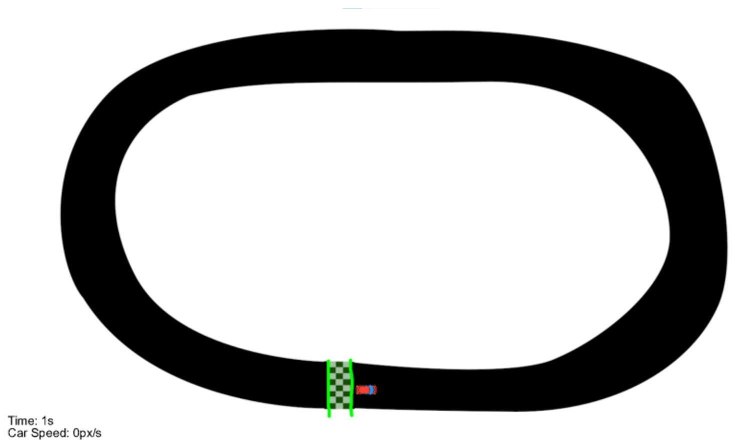


Image 5 – Execution of game on a simple map.

For this, I started by allowing the game to run on any map that was made by the user. This was one of the initial parameters for the execution of the genetic algorithm and, as such, had to be implemented in the game as well.

In order to utilize the track creation, the user created track must abide by a few basic constraints. In case the user does not wish to create a map for the program, there are a total of 5 initial maps that have been already created and that can be used. These vary in complexity and can be used in the game as well as the genetic algorithm. The first one involves only utilizing three colors for its creation. These ones are white, which will represent the track's borders and the area which the car cannot go through, black, which represents the track itself and the drivable area, and green, which will represent the finish line.

The second one is that the map must be 1920x1080 in size. This is due to the game, and therefore the algorithm, only considering maps of that size for their utilization. The usage of other sizes will make the program unusable. The third, and last constraint, is the usage of the same initial starting position for each map. This means that, no matter what map is made, the cars must start in the same position as they do on the already created maps. For this, the usage of the already created finish line (used in other maps) should be mandatory and used in the exact same position.

After having followed through all these constraints, the user may utilize his created map in the program. For this, you simply need to add the map to the "Maps" folder, present in the "Images" folder, and then change the image loaded directly. This is done via the "game_map" variable which can be found near the end of the code. This mapping feature can be utilized in both the genetic algorithm and the game, following the same constraints.

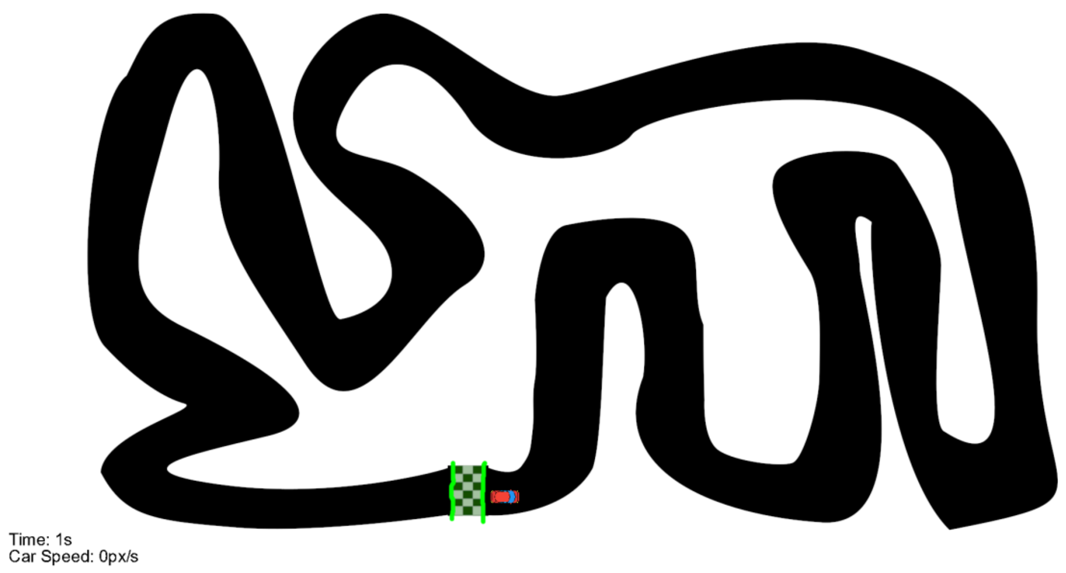

Time: 1s
Car Speed: 0px/s

Image 6 – Execution of game on a more complex map.

The game itself is very simple, following a few basic concepts in order to represent a more complete system. First, the car can move forwards and backwards, which is done via the W, A, S, D keys that will also control its rotation. It can rotate to either side, in which case the angle of the car and the direction of its movement will change. The speed of its movement is adjusted based on its acceleration as well as its max speed. In case the car is moving, and the user stops pressing forwards, the car will deaccelerate until it stops.

Second, the car is tracked individually on the track. In case it encounters either the wall or the finish line then it will behave appropriately. If it hits the wall then the car will crash, coming to a full stop. If it reaches the finish line, then the game will end. This is represented by a text, showing that the car has reached the end, and the reset of the map and the car after a few seconds. Third, and finally, the game will display some information about the game in the bottom left corner of the screen. This can be seen in the previous images and shows the current speed of the car as well as the time it is taking to complete the level.

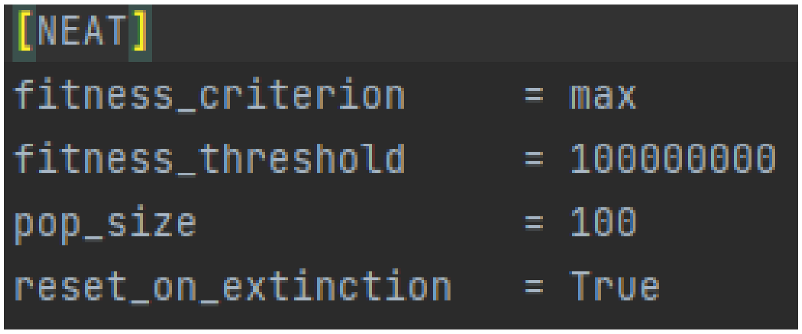
All these concepts together form the game which will serve as the foundation for the genetic algorithm. Not all of these remained for the algorithm as they only served a purpose for the execution of the game. In case the user wants to run the game (without any changes), all that has to be done is to run the “game.py” file.

Implementation of the Algorithm (NEAT)

This algorithm aims to create a simulation of the evolution of AI cars (self-driving) in man-made environments (racetracks). In order to create this program, we used python and the “neat-python” library (or just neat) as a method of utilizing the NEAT algorithm. This code was based on the video made by “Cheesy AI” but was optimized and changed to suit the needs of the game.

Before starting the implementation of the program, we must first set up the “neat-python” file. This is the “neat-config” file which will contain all sorts of parameters which will influence things like reproduction, mutation, and the amount of species. These parameters were initially taken from the video and then altered to suit the needs of the game, more specifically of the cars.

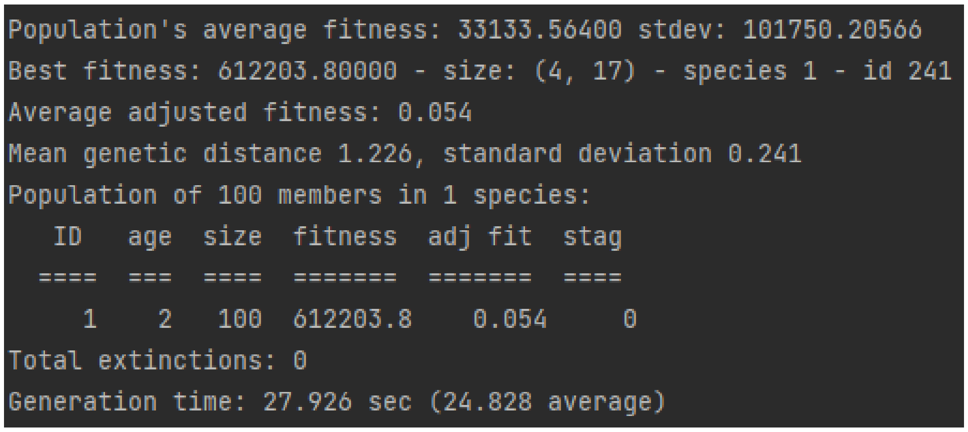
The most important parameters are the network parameters. These will represent the input and the output neurons of the neural network which had to be altered. There are 5 input neurons, due to the five sensors used by the cars to track the borders of the racetrack (what AI can see), and 4 output neurons which represent all the actions that the car can take (what AI can do). These consist of steering to the left and right (rotating) and increasing and decreasing the speed (move forward). These are the same actions that can be taken by the player, in the previously built game, excluding the action of moving backwards which was replaced by slowing down. As the AI aims to complete the track in optimal time, the addition of moving backwards would only be a detriment. We can also add a hidden layer, as it increases the complexity of the model, but it also increases the chance of overfitting and the training time. Other important parameters include the size of the population, which is the number of cars per generation, and the fitness threshold which is the fitness we aim to achieve.



```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 100000000
pop_size              = 100
reset_on_extinction   = True
```

Image 7 – Config file parameters for NEAT.

After having set the parameters for the configuration file of NEAT, we then had to access them via the code. This was done using various functions of the "neat-python" library which are equipped to deal with the various hyperparameters given. The "run_simulation" function is one created by us in order to initialize the game window and set the population of cars on it. Info is also displayed in the center of the screen that shows information about the current generation such as the number of cars still alive as well as the current generation number. In addition to this, there is a statistics reporter that displays in the console a few statistics about each generation after their completion. This information is the one that will be used to test / check the results.



```
Population's average fitness: 33133.56400 stdev: 101750.20566
Best fitness: 612203.80000 - size: (4, 17) - species 1 - id 241
Average adjusted fitness: 0.054
Mean genetic distance 1.226, standard deviation 0.241
Population of 100 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    2   100  612203.8    0.054     0
Total extinctions: 0
Generation time: 27.926 sec (24.828 average)
```

Image 8 – Information displayed by Statistics Reporter.

Having discussed all the settings required to run the algorithm, and the method of seeing their results, all that is left is to see how the algorithm functions. This will be seen by the perspective of the cars as they are the main influence of the neural network which will define the algorithm. All the code necessary for this interaction with the environment can be found in the "Car" class of the code.

As was seen previously, each car has five sensors which it will use to see the track and its borders. It will then take an action based on these sensors, which consist of increasing or decreasing the speed and rotating. These represent the input and output neurons of our neural network. The neural network connects all these neurons, in which each connection has a certain weight. Depending on their weights, the model will react in a certain way based on the input given (sensors). Initially, all of these reactions will be random as there will be no intelligence behind what they are doing or their motives for doing so. However, for each action they take they will either receive a reward or a penalty. This will influence them to only take the actions that give them rewards and stay away from actions that give them penalties. This was implemented using the fitness metric, in which the fitness of a car increases depending on the distance covered without crashing (reward). In case a car crashes, they are eliminated from the simulation (penalty).

Distance was used as a reward in order to stop cars from staying in the same place and not moving (if the time spent alive was a reward) or from simply turning around and reaching the finish line (if reaching the finish line was a reward). After each generation, we evolve our cars with the cars that have the highest fitness value as these will most likely survive and reproduce. On the other hand, cars with a low fitness value will eventually go extinct. When a car reproduces, it will not just duplicate itself onto a child car. This child car will be quite like its parent but not the same. Any car that is quite similar forms a species and if a species does not see any improvements after a few generations then it goes extinct as they are most likely dead ends. Using these principles, we create an algorithm in which the best cars survive and reproduce whereas the worst cars go extinct. Due to these extinctions, the model is forced to experiment with variations in order to find an improving species. Summarizing, the cars that go the most distance and are therefore rewarded persist and are replicated, although not fully prevent the population from becoming too similar to each other.

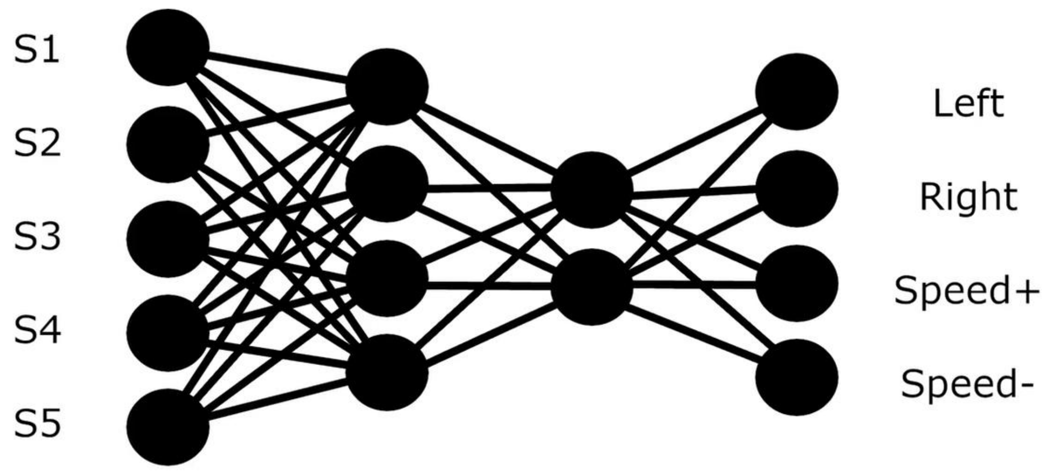


Image 9 – Architecture of Neural Network representing the Car.

Results

In this section, we will see the results obtained from the execution of the algorithm using different maps. These will alter in complexity which will be executed in order to see the algorithm's adaptation to more complex environments. Most parameters will be constant to avoid diversifying the results too much and to keep them comparable to each other in terms of testing. The most important ones consist of the generations, which will be left at 15 total generations, and the population, which will be left at 100 cars. As the simulation is ran in real time, as can be seen from the image below, all the results obtained in this section will be displayed by their respective Statistics Reporter.

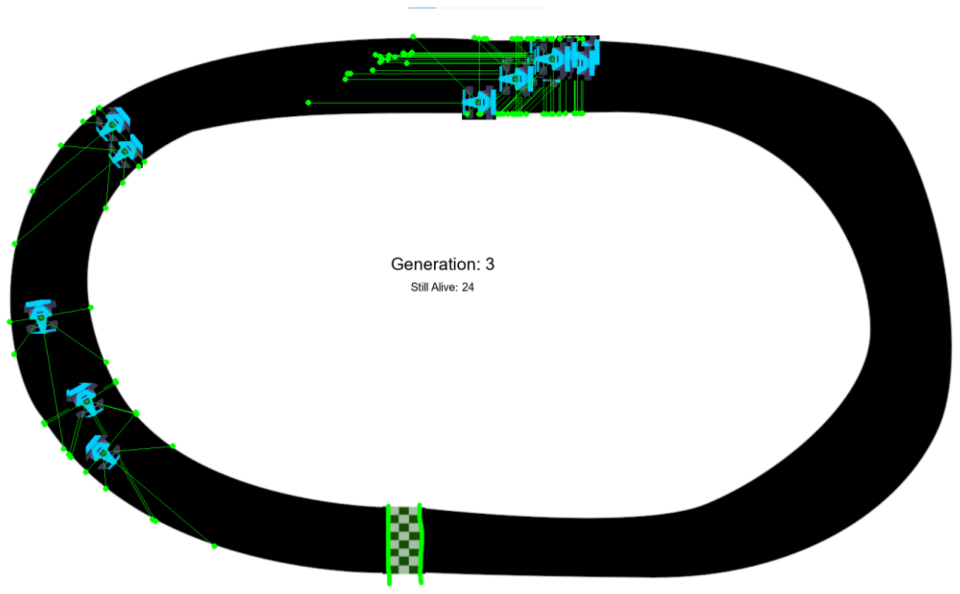
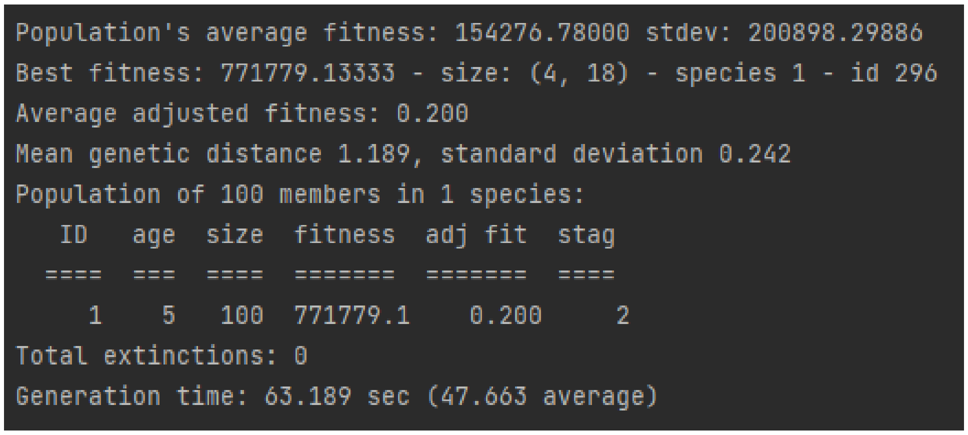


Image 10 – Execution of simulation on a Simple Map.

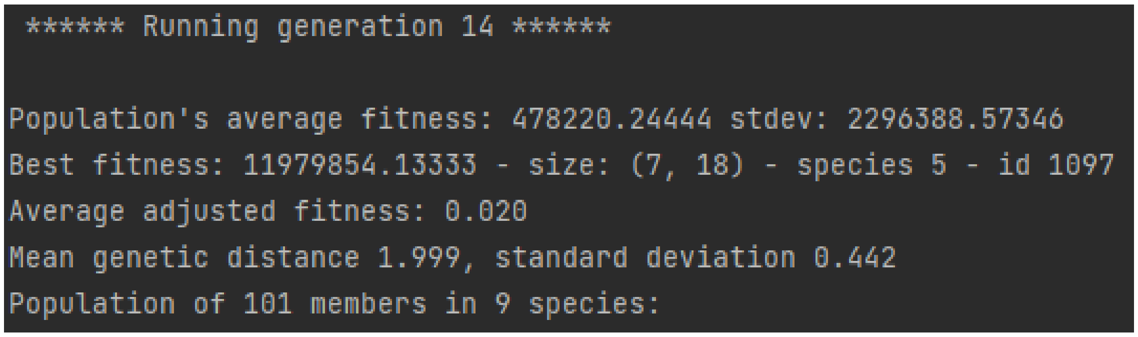
The first execution of the program / simulation was run on an amazingly simple map. This map can be seen above, and it is only an oval shaped racetrack which does not require the cars to think as much. As such, the cars were already able to finish the track by the second generation and, by the fifth generation they already had a high number of cars finishing it and continuously increasing their distance by not crashing. Because of this, I expected the next iterations to follow the same pattern and canceled the simulation early (on the fifth generation). The results can be seen below, in which the best fitness nearly reaches the fitness threshold highlighting its reliable results. Therefore, this simple map is more than completable by the AI.



```
Population's average fitness: 154276.78000 stdev: 200898.29886
Best fitness: 771779.13333 - size: (4, 18) - species 1 - id 296
Average adjusted fitness: 0.200
Mean genetic distance 1.189, standard deviation 0.242
Population of 100 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    5   100  771779.1    0.200     2
Total extinctions: 0
Generation time: 63.189 sec (47.663 average)
```

Image 11 – Results of Gen. 5 on Simple Map.

The next execution of the algorithm was run on a more complex map. This is the second map, which can be found in the "Maps" folder, and it contains several light and sharp curves to constantly test the cars. This is a much more complex map which causes the algorithm to take much longer to complete it. Given the 15-generation limit imposed by these tests, the algorithm was not able to complete the map but was extremely close to finishing it. Some cars also got stuck in a loop at the start of the map which would eventually be extinguished by the algorithm. This, however, did not happen during its execution and therefore can only be concluded based on its theory. The results seen below display the complexity of the map, given the high number of species, as well as the difference between the average fitness and the best fitness showcasing the diversity in results. Given more generations, the algorithm should be able to reliably complete the racetrack.

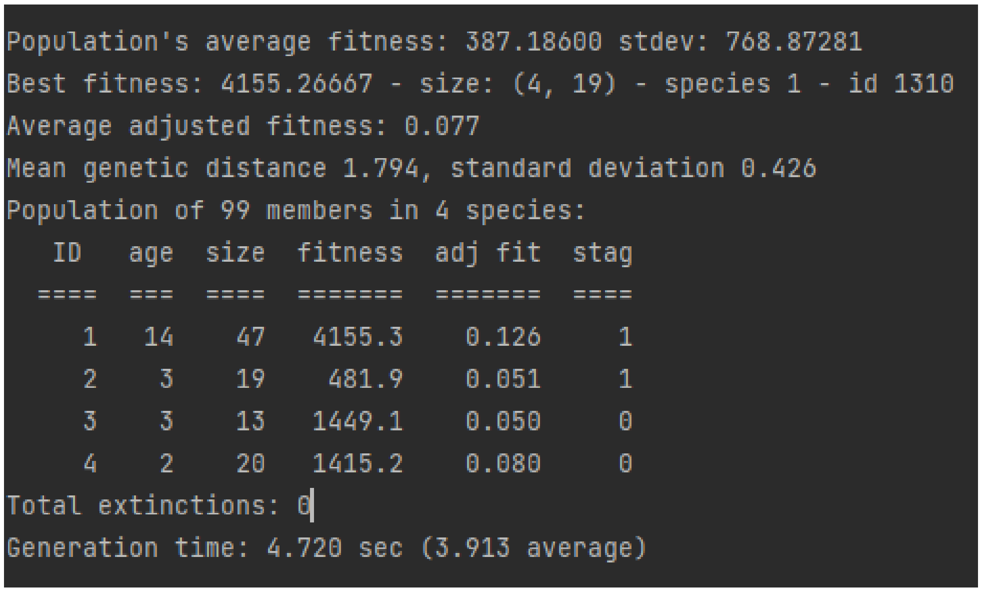


```
 ****** Running generation 14 ******

Population's average fitness: 478220.24444 stdev: 2296388.57346
Best fitness: 11979854.13333 - size: (7, 18) - species 5 - id 1097
Average adjusted fitness: 0.020
Mean genetic distance 1.999, standard deviation 0.442
Population of 101 members in 9 species:
```

Image 12 – Results of Gen. 14 on More Complex Map.

The last execution of the algorithm was done on the most complex map created. This is the fourth map, which can be found in the “Maps” folder, and it consists of extremely tight tunnels (slightly larger than the dimensions of the car) as well as sharp corners. This map was impossible for the algorithm to do in the small number of generations given. Some cars were able to adjust and complete the first sharp curve but could not get past that. The results display a few species, differentiating the ones that die instantly by their low fitness, as well as the amount of time that it took for that generation, which was nearly zero due to most dying right off the bat. Some cars can be seen as having done a better job, which is represented by the first species (id equal to 1) having a much higher fitness than the other species. Based on these results, the algorithm would have an extremely challenging time completing this track and would take hundreds of generations to even have a chance. Since the map is more complex, this could be adjusted by using hidden layers. However, these are currently not operational in the algorithm and therefore not usable.



```
Population's average fitness: 387.18600 stdev: 768.87281
Best fitness: 4155.26667 - size: (4, 19) - species 1 - id 1310
Average adjusted fitness: 0.077
Mean genetic distance 1.794, standard deviation 0.426
Population of 99 members in 4 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1   14    47   4155.3    0.126     1
     2    3    19    481.9    0.051     1
     3    3    13   1449.1    0.050     0
     4    2    20   1415.2    0.080     0
Total extinctions: 0
Generation time: 4.720 sec (3.913 average)
```

Image 13 – Results of Gen. 14 on Extremely Complex Map.

Comments

Having concluded this project, I see that in terms of usefulness it shows no major deviation to the one created via the reinforcement algorithm. They complete the same objective, although using different methods, and with no discernable differences in terms of execution. However, the work put into using the NEAT algorithm in this context displays its usefulness in terms of AI, especially due to its quick performance in showing results only requiring less than 5 generations for simpler maps.

In terms of the work shown, all the objectives defined in the proposal were achieved. The algorithm can run on any map given that falls under certain constraints and produces an AI capable of traversing a track without issues. However, this ended up not being implemented directly in the game due to conflicts with connecting the two implementations. This would be the next logical step for the algorithm although not its main goal, as that had been its implementation.

Due to not having multiprocessing, it is also terribly slow when it simulates a high number of cars which causes it to slow down drastically. This is a noticeable limitation as the algorithm takes longer due to this (increases time running). It also requires the maps given to adhere to its rules, which although not a massive limitation could be altered to suit the user / environment better. The most severe limitation is the inability of the algorithm to function with hidden layers. This should be a basic part of the model but due to unknown reasons, it causes several errors when done with hidden layers. If these were to be implemented, the algorithm could be used for more complex maps (such as the last one seen in the results section).

References

Video used as inspiration (Code Bullet):

- https://www.youtube.com/watch?v=r428O_CMcpI&list=WL&index=2&t=4s

Information about Genetic Algorithms:

- https://en.wikipedia.org/wiki/Genetic_algorithm

Information about NEAT:

- https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f

Initial foundation for code (Cheesy Ai):

- https://www.youtube.com/watch?v=2o-jMhXmmxA