NOVA IMS Information Management School

Project Report for Computational Intelligence for Optimization

Nova Information Management School, Lisbon, summer term 2021

António Pinto[1], Davide Farinati[2], Henrique Vaz[3] & Philipp Metzger[4]

[1]m20200659, m20200659@novaims.unl.pt
[2]m20201080, m20201080@novaims.unl.pt
[3]m20200586, m20200586@novaims.unl.pt
[4]m20201058, m20201058@novaims.unl.pt

Link to the GitHub repository:
https://github.com/ph1001/GA_learning_Snake

Abstract:
This report describes the design, implementation, and testing of a Python program capable of autonomously learning how to play an instance of the video game classic “Snake”. It starts with general explanations about the program, the game, and the existing interactions between them, then goes on with explaining the different Genetic Algorithm (GA) components used. Subsequently the utility of the different GA components for the task at hand are assessed by comparing them with each other over statistically significant numbers of runs of the program. Lastly the findings of the report are summarized.

I. Introduction

This report describes the design, implementation and testing of a program that autonomously learns how to play an instance of the video game classic “Snake”. The program presented in this report uses Genetic Algorithms (GA) and Artificial Neural Networks (ANN) for learning to play and for controlling the game. Descriptions of the general concepts of GA and ANN or of the game “Snake” will not be part of this report. Instead, it will focus on the implementation of the program as a whole and how it interacts with the “Snake” game as well as how it utilizes GA functions and operators. Please note that all figures of this report can be found in the Appendix.

II. Implementation

II.i Start of our program and initialization of its instances

The program at hand is written in Python and largely consists of three main scripts, “snake_evolution.py”, “call_snake.py” and “snake.py”. The purpose of “snake_evolution.py” is to start the whole program. This script creates an instance of the class “Population”, which as a consequence leads to the creation of a defined number of instances of the class “Individual”. The number of created individuals which represents the size of the population is defined by the user by passing a value for the variable “size” to the constructor of the “Population” instance. The population size chosen for this project was 25, which the authors deemed to be a number large enough for allowing adequate evolution capability and small enough for keeping the computational time in a reasonable range. Furthermore, when creating the population and calling its constructor, values for the variables “moves_till_stuck”, “fitness_fuction”, and “optim” are passed which define the number of moves the program will play without eating a piece of food inside the “Snake” game until the game is regarded as “stuck” and terminates, the fitness function that is used for evaluating the fitness of the individuals after each time they finished playing on game of “Snake”, and whether or not the program should be executed with the goal of approximating an optimal solution to the task at hand formulated as a minimization or a maximization problem.

The process of the creation of the population and the individuals as well as their attributes and behaviour are defined in “call_snake.py”. When the population is created, each individual is initialised with a unique value for the instance attribute “ind_number” which represents the individual’s index and a randomly initialized ANN is attached to it by saving it in the instance attribute “model”. The ANN of every individual is made up of layers which are instances of the “Sequential” class from the Deep Learning library Keras[1]. As mentioned before, the weights of these layers are randomly initialised. Furthermore, the authors decided to keep the default initialisation methods which is "glorot_uniform" for the layers' weights and "zeros" for the layers’ biases[2]. These initialisation procedures were kept because they yielded good results and because of that and due to our restricted computational capacities the experimentative focus was aimed at testing different GA-related operators. As the last step of the initialization of each individual it’s class method “play()” is called, resulting in every newly created individual playing one game (or multiple games[3]) of “Snake”. This process is described in the following subchapter.

II.ii. The “Snake” game and its interactions with “call_snake.py”

The class method “play()” calls the function “controlled_run()” which is defined in “snake.py”. The function “controlled_run()” contains the actual “Snake” implementation. The implementation at hand is an adapted version of the game available at [1], which is entirely written in Python.

In the following the process of running the “Snake” game will be described in a summarizing manner. A more detailed description will follow in the next paragraph. The execution of one or multiple games largely consists of running the function “gameLoop()” which is able to recursively call itself should the user have decided that more than one game should be played per individual and generation. If only one game is played, no recursive behaviour is executed. Inside “gameLoop()” a while loop lets the game run until the necessary conditions are met for setting the variable “game_over” to true, thus ending the execution of the while loop and the function “gameLoop”, returning the achieved scores and ages of this respective run or these respective runs in the form of two lists. The values of these two lists for score and age, which represent the number of food items eaten by the snake and the total number of moves done by the snake respectively, are then averaged and returned as two single values to the “play()” function call that initialized the playing of the game in the first place. Inside the play function and with respect to the fitness function that was defined by the user in “snake_evolution.py”, the respective individual’s fitness after this generation is calculated and saved as the instance attribute “fitness”.

[1] Further information on Keras is available at: https://keras.io/

[2] Further information about Keras’ “Dense” class is available at: https://keras.io/api/layers/core_layers/dense/

[3] “or multiple games” is just added here for completeness. The feature of one instance playing multiple games in one generation and then averaging the resulting fitnesses in order to obtain a more stable estimation of this individual’s fitness in the given generation was not used by the authors in the end; again for reasons of keeping the computational time at a reasonable level.

In the following we will describe the events inside the function “gameLoop()” in more detail, especially explaining how it interacts with the script “call_snake.py” in order to obtain move instructions from the ANN of the respective individual. The first move that the snake makes is always in the “Up” direction, since up to this point no useful information has been passed to the ANN that controls the snake’s moves. From the second move onwards, the current state of the game is summarized as a dictionary “game_state” which consists of the fields “snake_Head”, “foodx”, “foody”, “direction”, and “snake_List”. “snake_Head” describes the position of the snake’s head on the game grid, “foodx” and “foody” describe the position of the food item on the game grid which represents the snake’s destination of interest in order to increase its game score, “direction” contains a tuple describing the snake’s last move, and “snake_list” being a list of all positions on the game grid that are currently occupied by the snake. This information is then passed as an argument to the function “control()”, which is defined in “call_snake.py”. The function “control()” has two purposes:

Its first purpose is to process the information of the current state of the game in a manner that makes it usable as input for the ANN that is assigned to the currently playing individual. This is done by translating the information listed above (“snake_Head”, “foodx”, “foody”, “direction”, and “snake_List”) into binary information on whether or not (respective to the last move’s direction) on the next field of the game grid in front, to the right, and to the left of the snake’s head there is an obstacle, which could be either a wall or any part of the snake itself and into binary information on whether or not (respective to the last move’s direction) straight ahead, on the right, and on the left side of the snake’s head there is a food item located. This information is stored in the variables “clear_straight, “clear_right”, “clear_left”, “food_ahead”, “food_right”, and “food_left”. Two examples of game situations and the respective values of the variables listed before are depicted in Figure 1, which is located in the Appendix of this report.

The “control()” function’s second purpose is computing the snake’s next move by passing the information described above to the ANN’s “predict()” function and retrieving the index of the most probable output by using “numpy.argmax()”. The output of this process is an element of {0,1,2}, each of these three values representing one possible action for the snake to perform: If the output is 0, the snake’s next move will be to move one step on the game grid in the same direction as was its move before, meaning that it continues moving in the same direction. If the output is 1, the snake turns right and moves one step on the game grid and if the ANN’s output is 2, the snake turns left and moves one step on the game grid.

II.iii. Utilization of GA components

We are going to start by introducing all the genetic operators used and will then go on by explaining in detail how they behave together in the evolution process.

a) Selection

In order to handle the selection of individuals that would go through operators such as crossover and mutation, different selection algorithms were created and tested. The ones addressed by the authors were 1) Fitness Proportionate Selection, 2) Ranking Selection and 3) Tournament Selection. However, only 1) and 3) were tested because authors feared a loss of information using 2).

1) Fitness Proportionate Selection

The purpose of this algorithm is to decide which individuals are going to be selected as parents for crossover and mutation based on their fitness values. In a case of maximization, which is the scenario of this project, the individual with highest fitness value has the largest probability of being selected and the other follow by descendent order of fitness value. The formula follows bellow:

$P(Select\ Individual\ i) = \frac{f_i}{\sum_{j=1}^{n} f}$ (For maximization)

$n = size\ of\ the\ population,$

$f_i = fitness\ of\ individual\ i$

$f_j = fitness\ of\ individual\ j$

2) Tournament Selection

This selection algorithm holds some advantages when compared to the others. The first one is that with this technique the program doesn’t need to compute the fitness values of all individuals. Second advantage is related to the possibility of modifying the selection pressure by adjusting the tournament size.

The way the algorithm works is by randomly selecting k (k being the tournament size) individuals from the actual population, and the one that gets chosen is the individual holding the best fitness value (depending on the type of optimization).

b) Crossover

Once the individuals are selected, it is possible to proceed to the next step and apply the genetic operators. One of the operators implemented is the crossover, which is used to combine information of the parents, to generate the new offspring. In our project we implemented arithmetic crossover, which is the crossover operator that is most fit for dealing with genomes made up of continuous values. Arithmetic crossover generates new sets of weights for the neural network at hand, generated from the previously selected ones (parents). Arithmetic crossover linearly combines 2 sets of weights to produce the new offspring based on the following equations:

$$offspring\ 1 \ = \ \alpha * parent\ 1(1-\alpha) * parent\ 2$$
$$offspring\ 2 \ = \ \alpha * parent\ 2(1-\alpha) * parent\ 1$$

where α is a random weighting factor.
The probability "co_p", which represents the probability of applying this operator to a given pair of parents, was chosen to be 50%.

c) Mutation

As for crossover, since the individuals at hands are arrays of weights associated with the different neural networks' neurons, the type of mutation chosen has to be one that can handle solutions made up of continuous values.
Considering this, the type of mutation adopted for the project was Geometric Mutation. The logic behind this kind of mutation is a simple one, in which the user only needs to specify a constant value, known as the mutation step, m_s. Once m_s is decided on, the range $[-m_s, m_s]$ is created and from that range, with a given probability (probability of mutation), a value is picked from a uniform distribution ranging from $-m_s$ to m_s and added to a selected position in the genome of the individual.
In addition to this, since the optimization is an iterative process and starts converging overtime, m_s starts as a high value and decreases after each generation. That way, the authors ensure that at a given time when the program is closer to an optimal solution it undergoes only smaller alterations.
Additionally, a function for Normal Distribution Mutation was created by the authors, however the latter didn't get to be tested.

d) Fitness Sharing

One of the main drawbacks of the standard Genetic Algorithm is the premature convergence caused by the loss of diversity. To avoid this issue in our code we implemented two useful features, one to track various types of diversity through the evolution process and one to apply fitness sharing. By passing 'record_diversity = True' in the "evolve()" function of the class "Population" we were able to record phenotypic and genotypic variance and phenotypic and genotypic entropy over all the generations.
The variance was calculated with the formula $V(P) \ = \ \frac{1}{n-1} \times \sum_{i=1}^{n}(x_i - \overline{x})^2$ where for the phenotypic variance the variable were set as:
$n \ = \ size\ of\ the\ population,$
$x_i \ = \ fitness\ of\ the\ ith\ individual$and
$\overline{x} \ = \ average\ fitness\ of\ the\ population,$
and for the genotypic variance as
$n \ = \ size\ of\ the\ population,$
$x_i \ = \ distance\ of\ the\ ith\ individual\ to\ the\ origin$ and
$\overline{x} \ = \ average\ distance\ of\ all\ the\ individuals\ in\ the\ population\ from\ the\ origin.$

The entropy, instead, was calculated with the formula $H(P) \ = \sum_{j=1}^{N} F_j \log{(F_j)}$ where for the phenotypic entropy the variable were set as:
$N \ = total\ number\ of\ fitnesses\ in\ the\ population$ and
$F_j \ = \ fraction\ of\ individuals\ in\ the\ population\ having\ a\ certain\ fitness\ value\ (\frac{n_j}{N}),$
and for the genotypic entropy as
$N \ = total\ number\ of\ distance\ values\ in\ the\ population$ and
$F_j = \ fraction\ of\ individuals\ in\ the\ population\ having\ a\ certain\ distance\ value\ from\ the\ origin.$

For both the genotypic variance and genotypic entropy the origin was a random individual in the Population.
In order to avoid this loss of diversity we also made it possible to use fitness sharing, by just passing 'fitness_sharing = True' to the "Population" class' function "evolve()". Fitness sharing works by calculating the pairwise distance between all the weights of the neural networks (the genotype of our individuals) of all the individuals in the population, then it normalizes these distances in the range [0,1] and applies the function $s(i,j) \ = \ 1 \ - \ d(i,j)$ to all the distances. The for each individual it calculates the sharing coefficient as
$SHARING(i) \ = \ \sum_{j\epsilon p, j\neq i} s(i,j)$
and then adjusts the fitness of each individual as
$f_s(i) \ = \ \frac{f(i)}{SHARING(I)} \ for\ maximization\ problems$ and as
$f_s(i) \ = \ f(i) \ \times \ SHARING(i) \ \ for\ minimization\ problems.$
Applying fitness sharing allows the Genetic Algorithm to explore different peaks of the fitness space and not concentrate around only one.

e) Elitism

As the name suggests, elitism is the process of allowing the elite (best individuals), in our case the set of weights that achieved the best fitness, to carry over to the next generation without being transformed. This process ensures that the solution quality, in terms of fitness obtained by the algorithm, never decreases over the generations, achieving, in the end, the best individual that has been found throughout the process.
In the case of this project, an elite individual is a set of weights that is kept and attributed to a new neural network.

f) Multi-Objective Optimization
Previously we discussed the loss of the convergence of the fitness function as one of the drawbacks of standard Genetic Algorithm, another drawback of this evolutionary algorithm is the unicity of the fitness function. In our problem we were trying to optimize both the age and the score of the snake at the same time, and we did so by combining them into a mathematical function. Although by passing the argument “multi_objective = True” to the function “evolve()” of the class “Population” and setting “fitness_function = lambda x,y: (x,y)” we implemented a multi-objective version of the Genetic Algorithm. Multi-Objective Optimization works by copying all the individuals into a set and then removing all the non-dominated ones and assigning them a flag equal to 1. For an individual to be non-dominated it means that there doesn't exist another solution better on all criteria (in our case age and score). Then this process is iterated by increasing the flag at each step until the initial set is empty. Then the probability of selection is set to be inversely proportional to the flag assigned to the individual, specifically as $P(i) = 1 - \frac{flag(i)}{\sum_{j=0}^{n} flag(j)}$. When the evolution process is over the best individual is selected as the individual that has the smallest distance to the best optimal solution, that is constructed by selecting the best possible age and the best possible score.

These operators combine in the Genetic Algorithm evolution process in this way: firstly, a random Population with prefixed size is generated. Then two parents are chosen thanks to a selection operator, then the crossover operator is applied to the two parents with a predefined probability (“co_p” in the function “evolve()” of the class “Population”). Then to the two offsprings is applied the mutation operator with another predefined probability (“mu_p” in the function “evolve()” of the class “Population”) and the resulting offsprings are added to a new empty Population. This whole process is applied until the new Population has the size of the starting Population, and it is done for as many times as defined in the “gens” parameter in the “evolve()” function of the class “Population”.

III. Results and Discussion

In order to achieve statistical significance in the results shown, the authors had the goal to always perform 30 runs for each parameter to be tested. The values to be presented in the graphics and different analysis in this section are a computed average of the values over 30 runs (except for the assessments of the selection function and elitism, for which the authors, due to time constraints, could only execute 21 runs each).
Additionally, the population size used for testing was one of 25 snakes (sets of neural network weights) and the number of generations for each run was 100. These values were chosen considering a trade-off between having enough runs and snakes for achieving good results and keeping the computations times at a reasonable level.

III.i Fitness function

The selection of an appropriate fitness function is one of the most important steps to take into account in an optimization problem. Thus, it is important to have multiple options and make decisions based on the results achieved. Having this into consideration, the authors decided to test four different fitness functions:

1. *“davide”*: $f(age, score) = age * score$
2. *“antonio”*: $f(age, score) = age * e^{score}$
3. *“henrique”*: $f(age, score) = age * e^{score} + 101 * score$
4. *“philipp”*: $f(age, score) = e^{score}$

For the problem in hand, the authors had to find which of these outputs of a game (age and score) was to be rewarded more. In order to tackle this, the functions above were designed, and as can be seen, each one rewards age and score differently.
In Figure 2 the reader can infer that there are two fitness functions constantly reaching higher scores, those are: *davide* and *henrique*. Since no significant conclusion could be taken directly from the score (because both had similar development) another selection criteria had to be brought. Along with the score, the authors also kept track of the age and used it as a complement for the decision making. In Figure 3 the reader can see how long, on average, a snake lasted per generation. This was an indicator of how prown a snake was to hit either the walls or itself. In regard to the age, fitness function *henrique* did slightly better over the course of 100 generations, ending up as the chosen fitness function.

III.ii Selection function

As stated in II.iii a), three different selection algorithms were implemented. However only two of those ended up being used. The functions tested by the author regarding the selection process of the GA were tournament selection and fitness proportionate selection (FPS). The first one mentioned was considered the default to use. The graphics in the appendix, namely in Figure 4 and

Figure 5 are representative of these two trials. One line/box considers tournament selection and the other regards FPS. From the lines, the reader can conclude that despite the fact that the FPS line reached the highest spike close to generation 80, the one using tournament selection had a better overall performance among all generations. Looking at the box plot aids in verifying what was mentioned. The median logarithm of the fitness value is slightly above 20, on the other hand, the model with FPS didn't even reach that value.
Additionally, considering the savings in computational resources associated with using tournament selection led the author to keep this selection algorithm as the preferred one.

III.iii Elitism

The question whether or not to use elitism was also assessed in this project. It is important to note that the authors opted for using it from the beginning, leaving it to test later if it was beneficial to remove it or not. Another important note is that the elite individual in the problem at hand is not certain to perform the same way in further generations (because the positions of the food are randomly selected).
With fitness function and selection algorithm defined, the author resorted to fitness values in order to find out if the use of elitism brought improvement. Thus, a model with elitism was plotted against a model without elitism. The results are shown in Figure 6. As the reader can see, the orange line (without elitism) yielded worse performance throughout almost the whole process. The model with elitism helped the author to get better performance overall, so the decision was made to keep elitism on for the final version.

III.iv Fitness sharing

After having the fitness function decided, the authors were set to test if the use of fitness sharing would improve the performance of the snakes. In order to decide on this the users did another experiment, running the model 30 times with fitness sharing and another 30 without it. Since at this point the fitness function was already chosen, the authors could start using it as a primary metric of the model's performance evaluation. That being said, the graphics in the appendix, Figure 4 and Figure 5 are representations of the evolution of both configurations (with and without fitness sharing). Resorting to Figure 4 it can be stated that the models have reasonably comparable performances - both converge to log of fitness value in the range of 50-80, however, the one without fitness sharing reached higher fitness values more often. Any subsisting doubt disappeared after looking at Figure 5, where the reader can see two box plots, one for each configuration. From this chart, it can be concluded that despite the fact that minimum and maximum (both with outliers removed) are quite the same, the median value of the log of the fitness value regarding the model without fitness sharing, around 40, is much more encouraging than its counterpart, which is approximately 20.
Taking the above analysis into consideration, the authors decided to keep fitness sharing off for further runs of the model.

III.v Multi-objective optimization

As stated in II.iii f), this project also contemplates the implementations of Multi-Objective Optimization (MO). Since this technique can even be seen as an alternative to classic fitness functions, it becomes obvious that the comparison between results, using and not using MO, can't be done through fitness values. Thus, for this step we will look at the scores of the snakes.
Considering Figure 6 it is evident that the model with MO is outperformed by its counterpart. Snakes using the technique considered in this section don't reach an average score of 10 during the course of 30 runs, while the one without MO quickly converges to scores of 20 or higher. Only by looking at the lines chart with the scores, the decision whether to keep or not MO for further runs was trivial.

IV. Conclusion

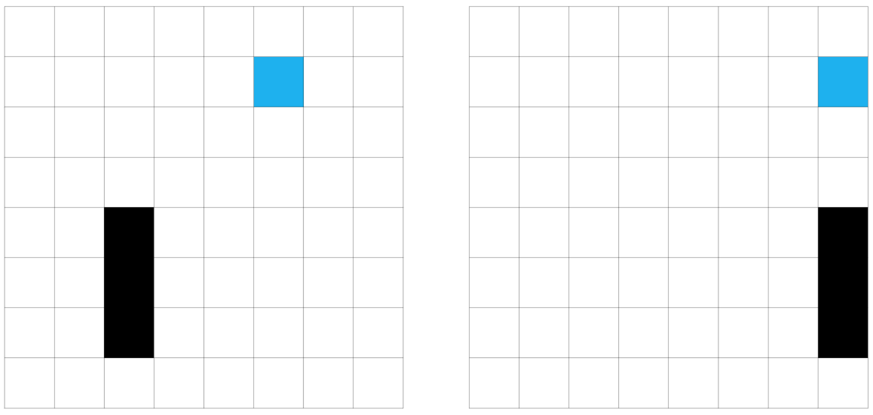
Summarizing the findings of this project, it can be said that the authors achieved the goal of designing and implementing a program that is capable of autonomously learning to play an instance of the video game classic "Snake". The setup that was found to be most suitable for the task is the following:

- Fitness function: $f(age, score) = age * e^{score} + 101 * score$
- Selection function: Tournament selection
- Elitism: Yes
- Fitness Sharing: No
- Multi-Objective Optimization: No

V. References

[1] https://github.com/shubham1710/snake-game-python, forked on 24. April 2021 at 14:40

VI. Appendix



clear_straight = 1
clear_right = 1
clear_left = 1

food_ahead = 0
food_right = 1
food_left = 0

clear_straight = 1
clear_right = 0
clear_left = 1

food_ahead = 1
food_right = 0
food_left = 0

Figure 1: Two exemplary game situations and their resulting values for the input of the ANN; black: the snake, blue: the food item

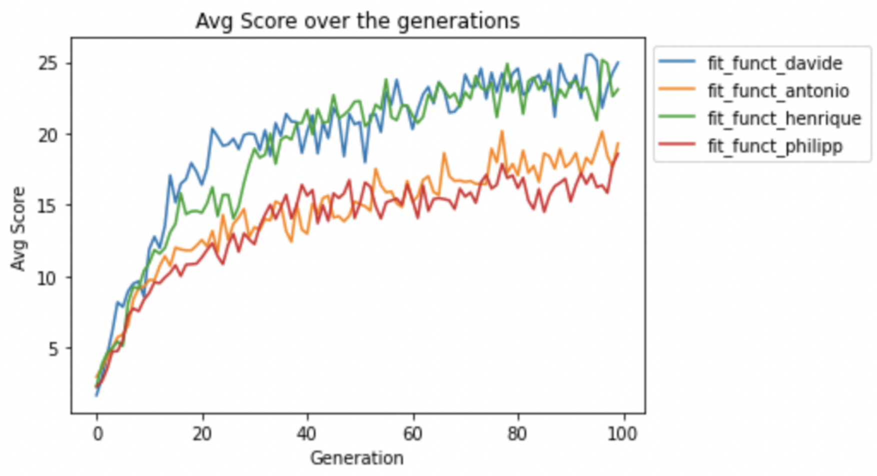


Figure 2: Average scores of the snakes' runs over generations

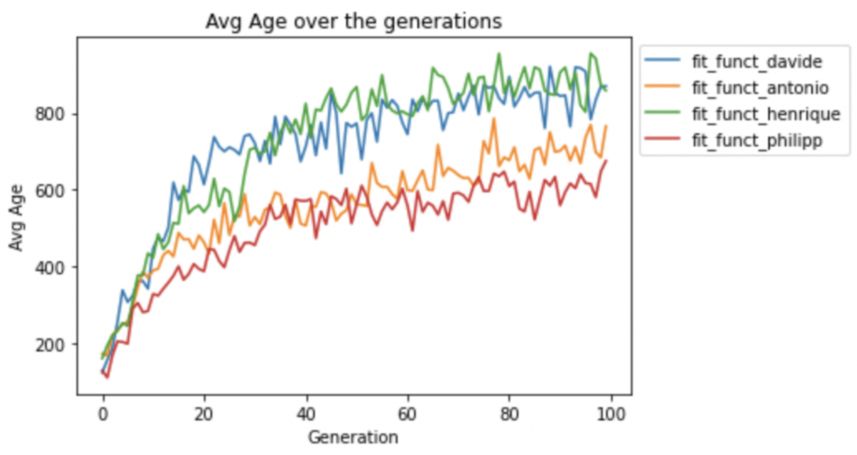


Figure 3: Average age of the snakes' runs over generations

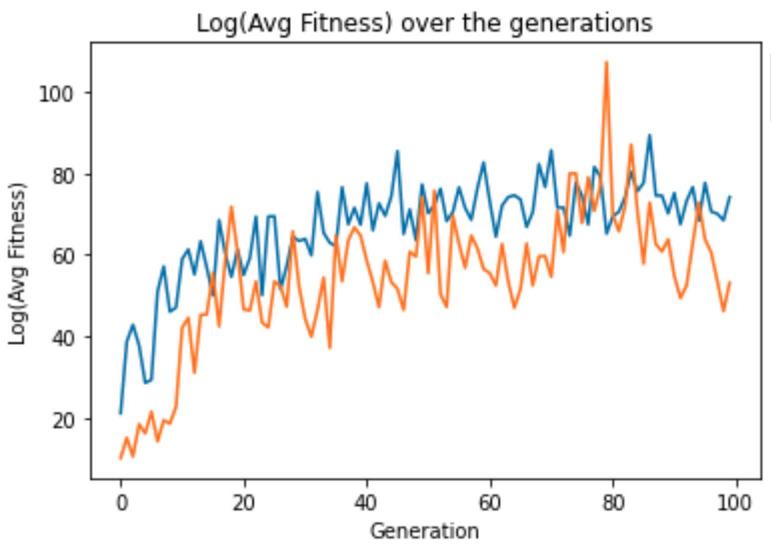


Figure 4: Logarithmic value of the average fitness values over generations; blue: tournament selection, orange: fitness proportionate selection

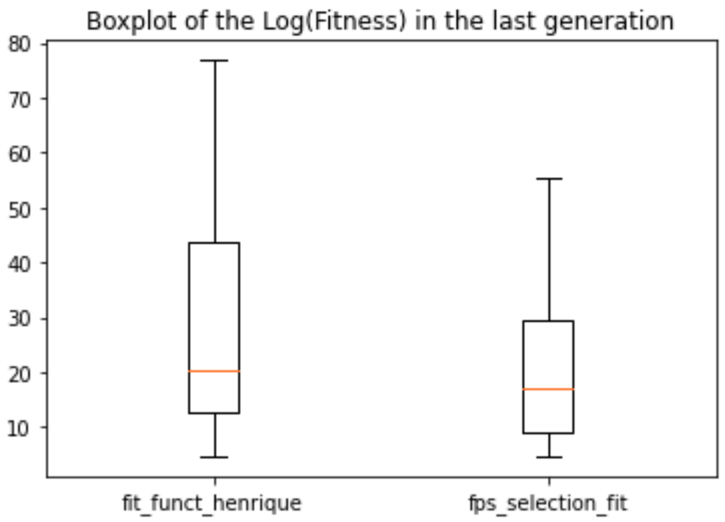


Figure 5: Boxplot with the logarithmic of the fitness values in the last generation; left: tournament selection, right: fitness proportionate selection

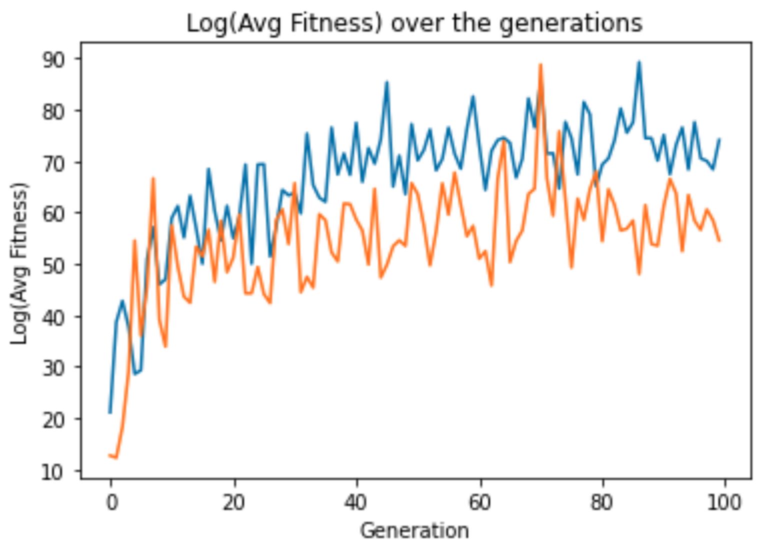


Figure 6: Logarithmic value of the average fitness values over generations; blue: with elitism, orange: without elitism

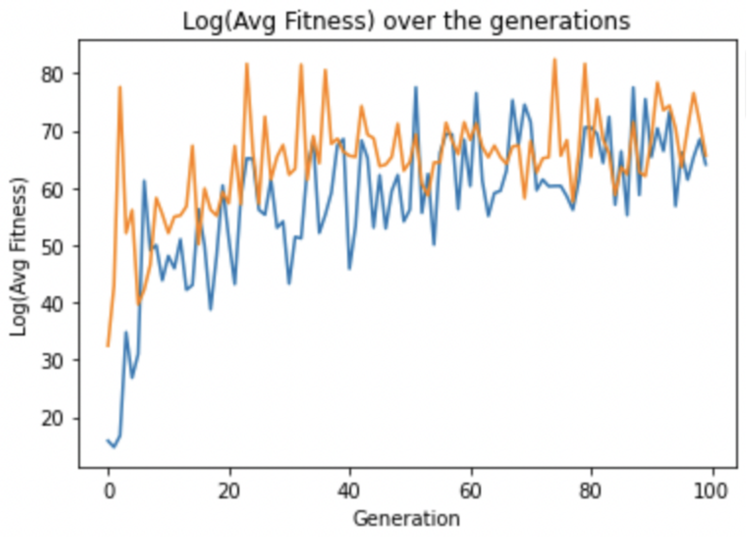


Figure 6: Logarithmic value of the average fitness values over generations;
blue: with fitness sharing, orange: without fitness sharing

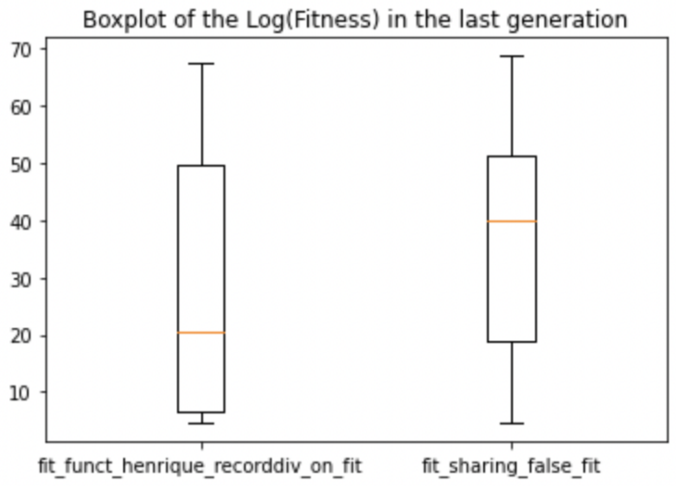


Figure 7: Boxplot with the logarithmic of the fitness values in the last generation;
left: with fitness sharing, right: without fitness sharing

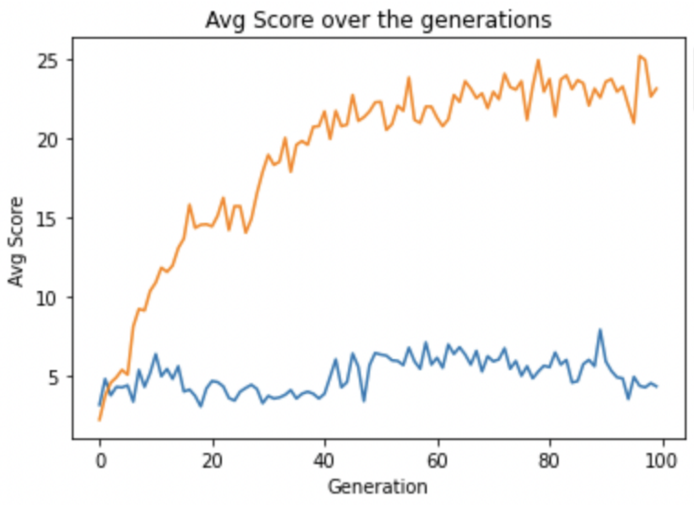


Figure 8: Average scores of the snakes' runs over generations;
orange: without Multi-Objective Optimization, blue: with Multi-Objective Optimization