# EVCO Assessment Report

Exam No: Y3838392

# 1 Introduction

## 1.1 Evolutionary Computation

Living organisms have evolved well-adapted structures which are present in the flora and fauna present on our planet. There are many strategies adopted by different species that allowed them to win against natural selection. Biomimetics is the science of implementing the elements developed by nature for solving complex human problems[1]. It has given rise to many technologies in various fields including computer science.

Evolutionary computation (EC) is an artificial intelligence subfield utilising a family of algorithms for global optimization inspired by biological evolution[2]. An evolutionary algorithms can generate a random population of solutions for a given problem which can then be evaluated and selected to proceed to the next generation. Individuals representing the strategies can also be altered by a set of dynamic mutations that are randomly applied with given probabilities before selection. The general idea is to repeat these processes in order to produce better population and extract the individuals representing the best solutions for the given problem.

## 1.2 Snake video game

Snake is the common name for a video game concept that has originated in the 1976 arcade game Blockade[3]. In a classical example of a Snake game the player can change the direction of a moving a chain of blocks mimicking the emonymous “snake”. The game also spawns pieces of food represented by a dot in random locations that the snake can eat to become. The objective of a the game is to get as many food pieces before hitting a wall or the snake’s tail, which becomes more difficult to avoid due to the tail growing longer over time. The ease of implementing Snake has led to creating hundreds of versions for various platforms.

## 1.3 Problem definition

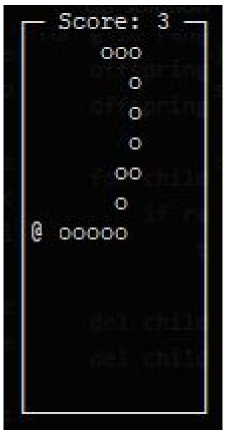

Figure 1: Version of snake to be used for the project.

The task given is to create an evolutionary algorithm in Python that can generate a strategy for playing a particular version of the classic video game, Snake. The version given to us allows the snake to move around inside a 12x12 grid with a single food piece of food spawning in a random location after the previous one is consumed (Figure 1). The snake is initiated with length of 11 blocks and its head is directed towards the right edge of the grid. The snake can change its orientation to any one of the 4 absolute directions (up, down, left and right), and always moves by 1 square at the time, letting the snake agent to define the next action to perform after each move. The maximum score that can be reached is limited to 133 (spaces on the grid minus the initial length of the snake i.e. 144 - 11 = 133), at which there is no space left to place the food as all the tiles are occupied by the snake's body. A snake can also die after it hasn't eaten any food pieces in the last 196 moves, permitting looping behaviours.

Implementing a solution for this project will bring the usual challenges that have to be faced when developing an evolutionary algorithm, for example maintaining diversity in the population by applying the right mutation methods with the probabilities suitable for the character of the problem and choosing the right selection method. Another issue is to propose a sensible evaluation function that will fairly define the fitness of an individual strategy for the training purposes.

The stochastic element of snake game related to the random placement of food pieces will also have to be taken into account to ensure that the proposed strategy performs correctly in a general case. The snake has to be able learn many concepts of the game such as eating food within a fixed number of moves, never turn back as it will result in hitting its own body immediately and staying within the boundaries and avoiding its own tail. An individual snake agent needs to know enough about the environment it exists in, but it shouldn't be overloaded with information to reduce the learning time and preferably make the solution a more general, so that it may be applied to grids of different sizes. In order to do that I will have to equip the agent with sensing methods that can extract the relevant data about the current state of the game.

In order to create a solution for this problem I am going to use some of the features included in an open source framework called Distributed Evolutionary Algorithms in Python (DEAP)[8]. The framework provides a set of tools for producing custom evolutionary algorithms, including and allows you to mix them with your own functionalities. It has all the utilities required for developing a solution for the given scenario.

# 2 Methods

## 2.1 Representation

Since 1966, when Lawrence J. Fogel has proposed the first method involving evolutionary programming called genetic algorithm[4] there have been developed various other techniques to represent the problems in a suitable form for the purposes of evolutionary computation. I have explored the pros and cons of the following techniques in order to find the most sufficient one to be used in the given scenario:

- **Genetic algorithm**
  In a traditional genetic algorithm (GA) a genotype of an individual is represented by a binary string of a certain length. GA's have already been implemented for use in the snake game[5], where the algorithm functioned by the rating functions affected by the space available around the head and the current food location, allowing the snake to evolve various strategies through which it became capable of reaching the maximum

score. Deciding on the most important variables about the environment and the way to encode them as a single binary string might be very challenging if I chose to use this technique.

- **Genetic programming**
  Genetic programming (GP) utilises a tree structure[6] to represent a genotype of an individual, where the nodes work as a set of specified operators and leafs define operands, allowing for evolving decision trees that can be expressed as a program calling one of the terminal functions depending on the results of a series of logical equations. This method is perfect for problems that can be simplified to a set of boolean values.

- **Neuroevolution**
  Neuroevolution involves evolution of artificial neuron networks[7] (ANN's) made of a set of input values on one side and a set of possible outputs represented by nodes. They may also involve additional multiple sets of empty nodes in between called hidden layers. Each node is connected to every node in the next layer. Each connection has a weight by which the value of the input node is multiplied to get the value for the node on the other side. Each node can have a bias value that is added to the sum of values generated by all the connections. In the end, the output with the highest value will be activated. By manipulating weights and biases we can change the importance of each of the inputs in relation to a certain output. An individual is then represented as a set of weights and biases, and the aim is to find the individual which activates the correct outputs. In the snake game the four outputs would activate a change of direction to one of the directions.

Even though neuroevolution seems like the most general solution, since a single state of the given version of snake can be easily simplified into a small set of boolean values I have decided to use genetic programming that can generate decision trees. I have decided to stay away from a GA representation, as it might get too attached to the size of the grid and hence become harder to generalise.

## 2.2 Inputs and outputs

In order to produce a genetic programming algorithm capable of creating decision trees, I must define a set of primitives and terminals that it can choose from. For the terminal set, we only need four function, each setting the current direction of the snake agent to one of the following: up, down, left and right.

For the primitive set there are many possible ways to specify it, including having a primitive for sensing the state for each of the 144 available locations on the grid. I have decided to base all of the primitives on an if_then_else function, and use various senses that I have equipped the snake agent with to act as conditions. All senses return boolean values. I have used different combinations of the following senses in the testing phase:

- sense_food_up/down/left/right - returns True if food is placed in the specified direction, false otherwise.

- sense_tail_1/2_up/down/left/right - tells the agent if there is a tail 1/2 (meaning there are two distinct sensing functions for each of the distances) tiles away in the specified direction.
- sense_out_of_bounds_1/2_up/down/left/right - tells the agent if a location 1/2 tiles away is within the playable grid the specified direction.
- sense_obstacle_1/2_up/down/left/right - returns True if the specified location is either out of bonds or is occupied by snakes body. 1/2 tiles away in the specified direction. Returns False otherwise.

The algorithm can then use these primitives and terminals to create a decision tree representing a strategy for snake. I am planning to gather statistics about some of the combinations of senses to decide which ones are crucial to find a strategy capable of acquiring the maximum score.

## 2.3 Initial population

To generate the initial population I will use one of the methods available in DEAP called genFull, which generated trees made of previously specified primitives and terminals, where each leaf is placed between the min and max values which are the arguments of this method. I have decided to keep the initial sizes at the minimum, letting the first generation of trees to have depths between 1 and 2, encouraging the algorithm to find the shortest shortest possible strategy.

Throughout the project I have tested different population sizes ranging from 100 to 1000 to examine the impact of the population size on the learning curve of the snake agent through generations.

## 2.3 Fitness evaluation

The simplest possible evaluation of an individual would be to run the game using a given strategy and set its fitness value to the score once the agent has finished playing. The only flaw about this is related to the stochastic nature of the game, as due to the random placement of the food pieces, the snake agent might get a high score purely thanks to luck. To avoid rewarding an individual in such cases with high fitness values I have decided to run the game 5 times for each strategy and take the average score reached in those runes. This increases the time needed for evaluation by a factor of 5, but it will also reduce the impact of the random element of the game on the fitness value.

## 2.4 Selection

Deciding on a selection policy for choosing the individuals that should influence the shape of the next generation is one of the most important factors of creating a successful evolutionary algorithm. If only the best individuals are selected then the diversity will be lost, causing premature convergence, meaning that the algorithm getting stuck in a local minima and be unable to develop new strategies that might turn out to have higher fitness. To avoid that a good selection method must have an element of randomness involved, keeping some of the individuals with lower fitness values.

Another aspect that must be taken into account regarding the selection phase involving a genetic programming solution is reducing the bloat, which is a term for the increase of the size of a tree without significant improvements in fitness of the population. Bigger decision trees may often be simplified, but the algorithm is not aware of this fact. Deap trees usually introduce a lot of redundancy, resulting in extending  the time required for their execution and evaluation.

To resolve these two issues I have decided to use double tournament which begins with a traditional tournament by partitioning the population to random groups containing a small number of individuals (10 in my case) and sorting them based on their fitness values, but the individual with the worst score in this group can still be selected only with smaller probability solving the problem of diversity. In individual getting the first place has the highest probability to be reproduced. These individuals then participate in another tournament comparing only 2 individuals based on their size, and halves the reproduction probability of the larger individual, encouraging the algorithm to favoritize smaller solutions.

I have decided stay away from using a technique called hall of fame, which guaranties to select and advance a certain number of individuals that have shown the highest performance due to the stochastic element of the game, meaning that a high score might be a coincidence. Because of that I would rather keep the diversity of the population at the maximum level by storing the best individual ever seen outside the population used by the algorithm.

## 2.4 Reproduction

After the selection phase the chosen individuals can be altered by a specified set of operators with certain probabilities in order to refill the population with new individuals that will be evaluated in the next generation.

For my algorithm I have decided to first apply a simple one point crossover offered by DEAP, which given two individuals randomly selects a node in each individual and exchanges each subtree with the point as root between each individual[9] inspired by passing genes during sexual reproduction in biology.  The operator can be applied to each pair of individuals that are next to each other in the population with a specified probability .

The next operator used in the reproduction phase is mutation that be applied to each of the individuals separately with a given probability. Mutation replaces a randomly selected node with a a root of another randomly generated subtree. In order to keep the sizes of the trees low, I am only allowing for introducing mutations of depths ranging from 0 to 5, where 0 replaces only one function from the primitive set with another.

To ensure that the diversity of solutions is kept at the highest possible level I have decided to apply mutation to every individual by setting its probability to 1. In order to discover the most efficient value for the crossover probability I have performed a series of short tests summarised in the next section..

## 3 Results

As mentioned in the previous section, there are a lot of variables that can still be adjusted for better performance and the best way to decide on their final form is through testing. First of all I tested for which of the senses that I have designed are absolutely necessary for the snake to develop a perfect strategy for the snake agent. To do that have run the algorithm using 4 different combinations of senses with the same mutation probabilities and population sizes over 250 generations.

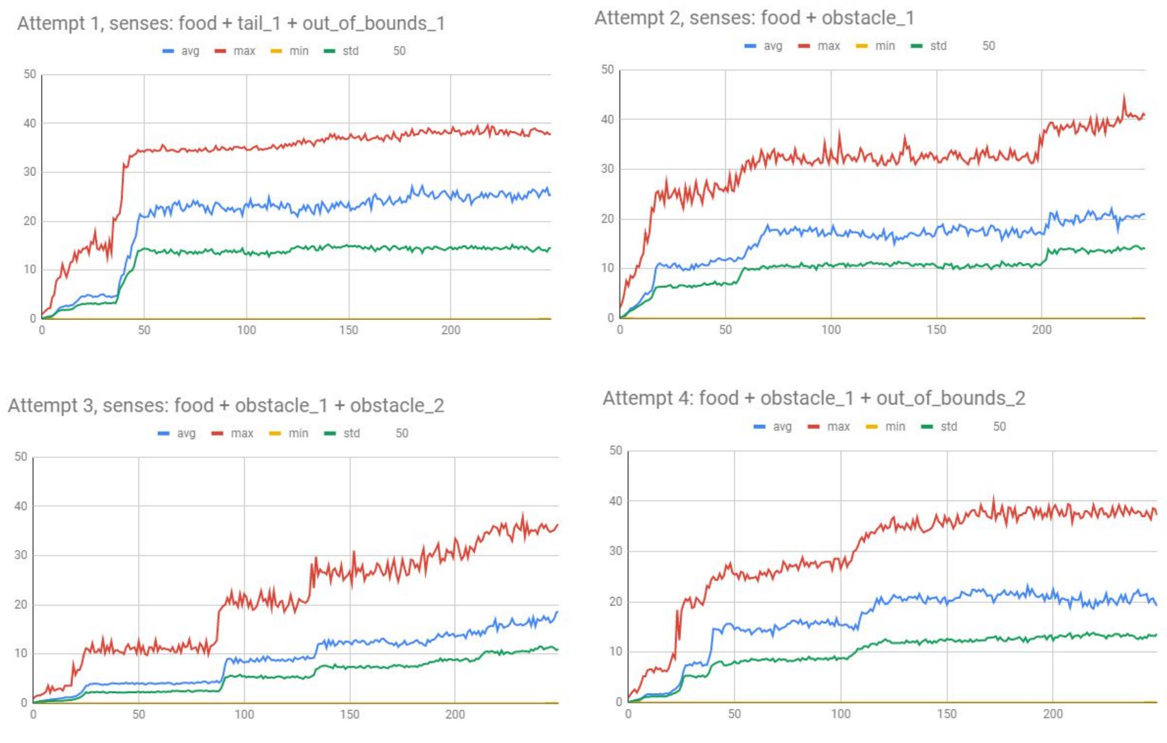

Figure 2: Comparison of senses combinations.

I have started testing by allowing the agent to recognize the direction of the food relative to its current head position allow it to detect if there is a tail or a wall are one tile away using separate senses for each of them. To increase the learning speed I then tried to simplify the problem by reducing the number of senses and merging two of them into one, not allowing the snake to tell the difference between a wall and its own tail. By referring to the graphs in figure 2 we can see that the initial learning curve has indeed gotten steeper and also the average increased in comparison to the previous graph, but the fact that in both of these attempts the maximum has stayed on a similar level, I have concluded that the snake is not getting enough information about the environment to generate better results. To resolve this problem I have decided to increase the range of sight of the snake by introducing another set of senses detecting obstacles 2 tiles away. This reduced both the average and maximum fitness values compared to the previous combinations, but the maximum fitness seemed to increase more steadily. Eventually I have altered the senses to only allow for detection of walls two tails away and since it raised the average and maximum values again I have decided to use this set of senses for my final solution.

To decide on the population size I have run the algorithm for 50 generations with different population sizes on the same hardware using the same crossover probability (0.5) keeping records of the average and max fitness as well as the execution time (Figure 3). After comparing these results I have concluded that using a sample of 300 individuals will be large enough to find efficient solutions and still be able to evaluate generations in a reasonable time. Using population of size 600 would also be a viable option given the results.

| **Population size** | **Average fitness** | **Max fitness** | **Execution Time (in seconds)** |
|---|---|---|---|
| 150 | 3.8 | 9.4 | 34 |
| 300 | 11.1 | 19.2 | 270 |
| 600 | 15.2 | 38.6 | 340 |
| 1200 | 10.7 | 17.4 | 1852 |

Figure 3: Test results for finding the most efficient population size.

The last decision to make involved establishing the best value for the crossover probability. I have taken a similar approach for this purpose. Based on the results gathered by running the same algorithm over 50 generations (figure 4) I have decided to set the crossover probability to 0.4 since it managed to get the highest average fitness for the population.

| **Crossover probability** | **Average fitness** | **Max fitness** |
|---|---|---|
| 0.3 | 9.0 | 19.2 |
| 0.4 | 13.2 | 26 |
| 0.5 | 9.2 | 16 |
| 0.6 | 12.6 | 29.6 |

Figure 3: Test results for finding the most efficient crossover probability.

Finally, after all the missing factors have been established I have run the finished version of my algorithm for 1000 generations which taking over 6 hours to execute on my hardware. It managed to evolve a perfect strategy for the snake game (appendix 1) after 983 generations. To ensure that the algorithm can find the strategy for reaching maximum score regardless of the numbers provided by the random number generator I have run the algorithm again, but this time I have changed the random seed from 3 to 0 providing completely different set of numbers. By referring to the graphs representing these two runs (figure 5) we can find that both of them managed to reach maximum score, although seed 0 managed to generate a perfect strategy almost 400 generations earlier (first encounter at generation 597).

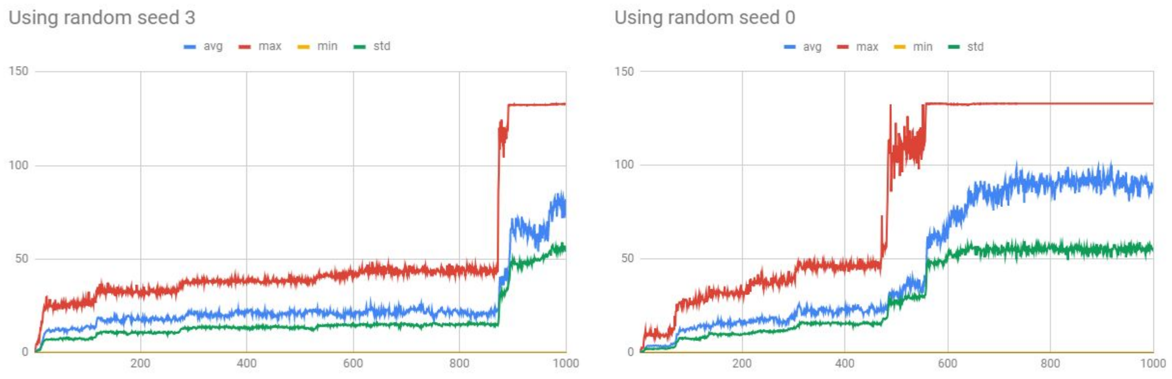

Figure 5: Two runs of the same algorithm using different random seed.

# 4 Conclusion

This project has shown how strongly the evolutionary algorithms can be attached to pure chance. In the last comparison involving using different random seeds both runs have stopped increasing their maximum fitness after reaching the score of 50 before advancing by a significant measure as they had to wait for a particular rare mutation that have allowed the algorithm to achieve the maximum score, the only difference being that one of them had to managed to reach this state in almost half the time. This can be considered a major flaw, but also a blessing of computer algorithms trying to solve human defined problems as stated by Janine Benyus “Computers can generate random ideas much faster than most engineers. And computers, not yet able to feel embarrassment or peer pressure, are not afraid to try off-the-wall ideas. Ideas are just ideas; the more the merrier.”[10]

Even though my solution was able to find multiple strategies for winning a game of snake it would be reasonable to increase its reliability of the algorithm by reducing the potential difference in generations required to evolve it. If given more time I would also like to compare the genetic programming approach against a solution utilising neuroevolution written in using NEAT library for Python that I did not manage to complete on time.

For further work I would suggest to find more efficient operators that would be able to develop a more advanced A.I. capable of winning the game in the shortest time possible as well as being able to handle situations when there are more than one piece of food placed on the grid at one time.

## 5 References

[1] J. F. Vincent, O. A. Bogatyreva, N. R. Bogatyrev, A. Bowyer, A. Pahl, “Biomimetics: its practice and theory”, Journal of the royal society, April 2006
[2] Back, , T. Fogel, D. B. Michalewicz, “Handbook of Evolutionary Computation”, 1997.
[3] G. Goggin, Global Mobile Media. Taylor & Francis, October 2010.
[4] L. J. Fogel, A. J. Owens, M. J. Walsh, “Artificial Intelligence through Simulated Evolution”, John Wiley, 1966
[5] J. F. Yeh, P. H. Su, S. H. Huang, T. C. Chiang, “Snake game AI: Movement rating functions and evolutionary algorithm-based optimization”, ResearchGate, November 2016
[6] N. L. Cramer "A Representation for the Adaptive Generation of Simple Sequential Programs", July 1985
[7] K. O. Stanley, "Neuroevolution: A different kind of deep learning", O'Reilly Media, July 2017
[8] F. A. Fortin, F. M. De Rainville, M. A. Gardner, M. Parizeau, and C. Gagné, “DEAP: Evolutionary algorithms made easy”, Journal of Machine Learning Research, July 2012.
[9] DEAP documentation, https://deap.readthedocs.io
[10] J. M. Benyus, Biomimicry: Innovation Inspired by Nature. HarperCollins, May. 1997.

# 6 Appendix

## 1 Strategy for reaching perfect score developed by the algorithm

(Parameters: random_seed = 3, pop_size = 300, ngen = 1000, cxpb = 0.4, mutpb = 1.0)

```
if_food_up(
    if_obstacle_left_1(
        if_out_of_bounds_right_2(
            if_obstacle_left_1(
                if_obstacle_right_1(
                    changeDirectionDown,
                    changeDirectionRight),
                changeDirectionRight),
            if_food_up(
                if_out_of_bounds_left_2(
                    changeDirectionUp,
                    changeDirectionRight),
                changeDirectionRight)),
        if_obstacle_up_1(
            changeDirectionLeft,
            changeDirectionLeft)),
    if_obstacle_right_1(
        if_obstacle_left_1(
            if_obstacle_down_1(
                if_obstacle_left_1(
                    changeDirectionUp,
                    changeDirectionLeft),
                if_obstacle_right_1(
                    if_obstacle_left_1(
                        changeDirectionDown,
                        if_food_up    (
                            if_obstacle_down_1(
                                changeDirectionUp,
                                if_obstacle_down_1(
                                    changeDirectionDown,
                                    changeDirectionDown)),
                            changeDirectionRight)),
                    changeDirectionUp)),
            if_food_up(
                if_obstacle_left_1(
                    if_out_of_bounds_up_2(
                        if_food_down(
                            changeDirectionLeft,
                            if_out_of_bounds_left_2(
                                if_obstacle_up_1(
                                    changeDirectionLeft,
                                    changeDirectionDown),
                                changeDirectionLeft)),
                        if_food_left    (
                            if_obstacle_down_1(
                                changeDirectionDown,
                                changeDirectionLeft),
                            if_out_of_bounds_right_2(
                                changeDirectionUp,
                                changeDirectionLeft))),
                    if_food_right(
                        if_out_of_bounds_left_2(
                            changeDirectionDown,
                            changeDirectionLeft),
                        changeDirectionDown)      ),
                if_food_up(
                    changeDirectionRight,
                    if_out_of_bounds_left_2(
                        if_obstacle_down_1(
                            changeDirectionLeft,
                            changeDirectionDown),
                        changeDirectionLeft)))),
        if_obstacle_up_1(
            changeDirectionRight,
            if_obstacle_down_1(
                changeDirectionUp,
                changeDirectionRight))))
```

*The statistics supporting all the results mentioned throughout the report are stored in the in the ‘results’ folder of my submission