E.C. Snake

Image generated by DALL·E

Fluffy Illustrated

Our favorite snake, "fluffy", devouring countless apples, for he is a mighty undefeatable snake.

You may notice the tears running down fluffy's eyes, shed for all the generations sacrificed to achieve his perfection.

Table of Contents

INTRO

What is Evolutionary Computing (EC)?

EC is a sub-field of artificial intelligence in which the algorithm approximates Darwinism.

Charles Darwin’s theory of biological evolution states that all species develop through the process of natural selection. Individuals of a species would mutate slightly and if their mutation benefitted their chance of reproduction then it was more likely that their mutated traits would be passed down through the generations.

EC works on the same principles. It initializes a population possible solution to the problem we wish to solve. We try each solution and evaluate its worth. Solutions with higher worth will have a higher impact on the following generation where as those with a lower worth are phased out. The next generation is created by making small mutations to the more successful solutions. Repeating the process has the result of obtaining increasingly refined solutions to the original problem.

Evolutionary computing can be used for complex optimization problems that have many variables making it difficult for more traditional algorithms to solve. One branch of problems that EC has shown great success in is creating AI for playing games optimally.

What is Snake?

The snake game is one of the earliest video games, dating back to the 70's. it has seen countless remakes and versions on many different types of devices, including arcade consoles, home computers, and cellphones to mention a few. The snake games became very popular when Nokia included their first Snake game on their cellphones in 1997.

The basic idea of the game is that the player controls the head of a snake that grows longer each time the snake eats a piece of food, and the snake dies if it hits itself or a wall. Generally, the game has a grid on which the snake and the food occupy squares and the snake can only make right angle turns, but there are many versions of the game. Some include 360 degrees of mobility, some are in 3D. Other variations of the game include games like TRON where there are multiple snake-like objects, and the goal is to cause the other players to lose. The game Snake.io combines several factors of these variations, it is online multiplayer, involves eating food and other snake players to grow.

For the purpose of this project, we will implement a simple version of a 10x10 grid snake game with 4 pieces of food on the board at any given time.

The Problem

We generate the game board, including the snake and the food, as a 10 by 10 two-dimensional grid. The grid contains 0 in empty spots, 1 in snake spots, 99 in food spots.
The food is implemented as a predetermined list of coordinates such that there are always four food spots on the game board.
The snake is given as a list of coordinates, the last coordinate being the head of the snake.

The gameplay loop is as follows:
The game is initiated with a player object. At each iteration the game asks the player for its next move. The moves are UP, DOWN, LEFT, and RIGHT. The game calculates the appropriate new position for the snake, checking if the head is on a food spot – in which case the snake size should grow by one, or if the snake has touched the edge of the board or itself – in which case the game should terminate. We have also given a maximum number of turns – if this number is exceeded then the game will also terminate.

Our goal is to create a player AI that will maximize the length of the snake.

The Solution

Using evolutionary computing, we will evolve a population of snake players (individuals) based on the final length of the snake at termination as the metric for evaluation. The individuals' brains are neural networks with two hidden layers. The brains receive a 7x7 portion of the 10x10 board and produce the next move for the snake (UP, DOWN, LEFT, or RIGHT). During play, after receiving the brains output, the board will adjust itself accordingly.

For the purpose of evolution, we use random mutation and cross mutation. For random mutation each element of the brain has a chance mutation_chance to be adjusted up to a difference of mutation_size. For cross mutation we mix the elements of two brains allowing us to gain the potential individual benefits of each.

The Code

Our project contains 6 files:

- game.py
- genetic_operators.py
- genetic_players.py
- main.py
- utils.py
- run.py.

We will layout the main functions and purpose of each file.

- Utils.py:
  This file contains various functions for the overall functionality of the project. These functions are as follows:
  - generate_brain() – this generates a randomly initiated neural network with two hidden layers.
  - flatten_brain(brain) – takes a brain and returns a single vector of the concatenated brain layers.
  - inflate_brain(arr) – reverses flatten_brain.
  - evaluate_brain(player) – plays a game with player controlling the snake. Returns the final length of the snake upon termination.
  - mutate_brain(brain, mutation_chance, mutation_size) –
    creates a new brain based on the given brain. For each element in brain the chance of it changing is mutation chance and it can change up to mutation_size in either direction.
- Game.py:
  This file contains the implementation of the game itself. The class Game receives a player object.
  - Class Game:
    - Constructor: initializer of the game, receives a player (brain) and sets up all the required fields.
    - Move() – Calls the next_move method of genetic_player and updates the board accordingly.
    - Play() – Stimulates the game iterations.

- Class GUI: defines and runs GUI.

- Genetic_player.py:
  Here we implement an individual neural network for the snake agents.
  - Constructor: receives brain and initializes the player.

  - Process_board(board, snake): receives the board current instance, processes it giving priority to squares with food and negative priority to squares with snake's body or game border.

  - Next_move(board, snake): Calls process_board and propagates the results through the brain matrices, returns the max value of final layer (4 float values representing UP DOWN LEFT RIGHT)

- Genetic_operators.py:
  Contains operators used for EC-Kity and the project.
  - **Class** BrainCreator – Creates an individual for the ECKity population.

  - **Class** BrainEvaluator – Evaluates an individual by running a game instance on a given brain.

  - **Class** BrainMutator(mutate_chance) – Mutates an individual with consideration of mutate_chance by adding or decreasing values in an individual brain, each cell is randomly mutated.

  - **Class** BrainCross(cross_chance) – Crosses two brains by splicing them randomly, with consideration of cross_chance.

- Main.py:

  Initiates the ECKity main algorithm.

  Sets the SimpleEvolution and runs it.

- Run.py:

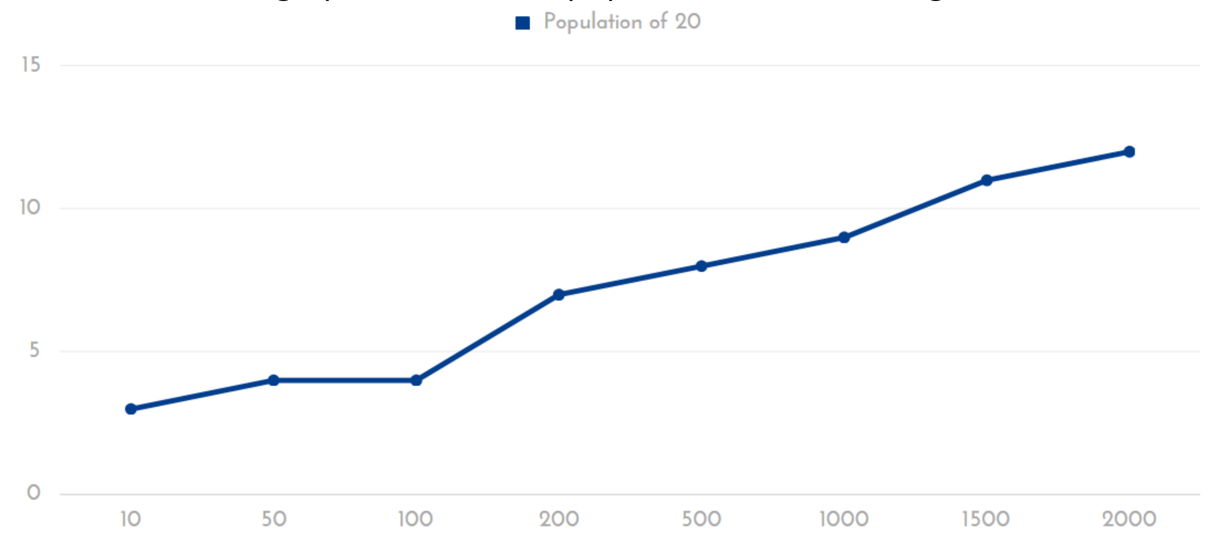
  Runs an instance of the game with given individual (brain).

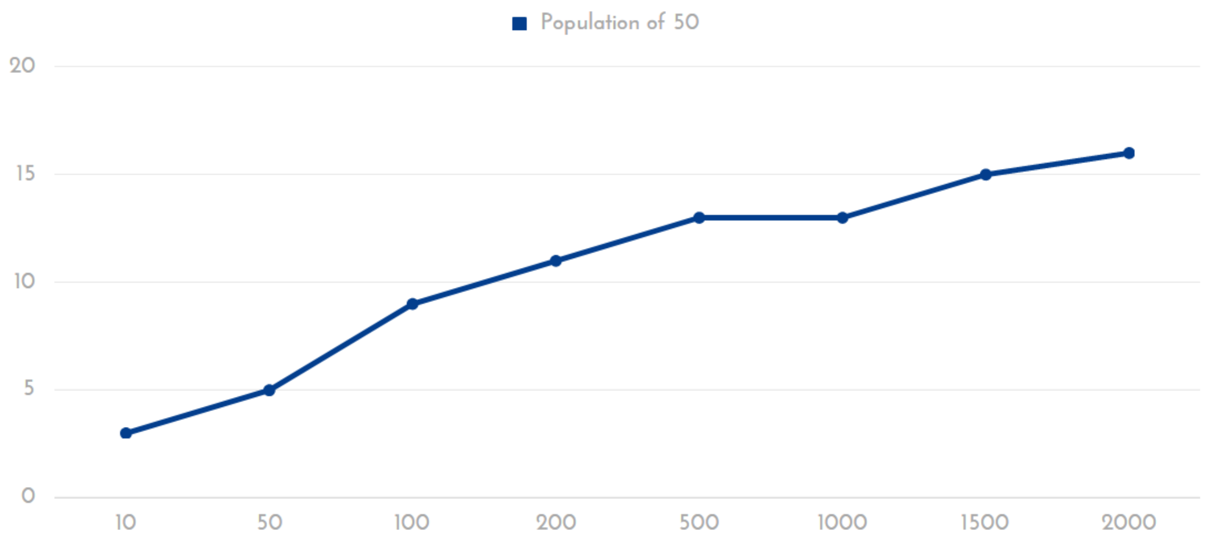
Results

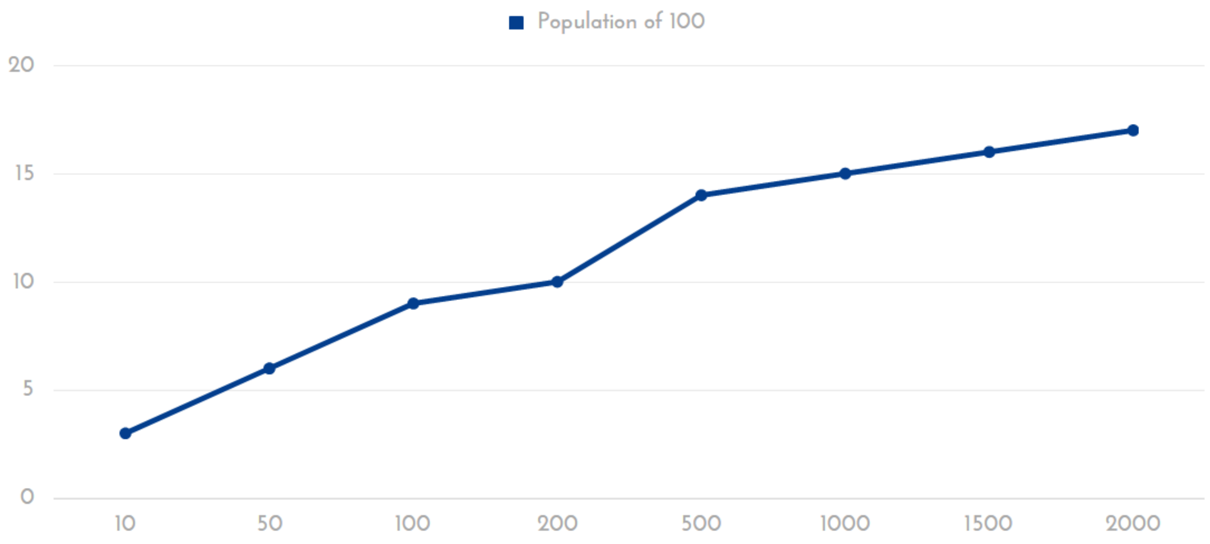
We began running our litter of snakes and testing for different cross-over and mutation chances, below are results of our best achieving variables with

mutation chance = 0.5;   cross chance = 0.3

Shown below are graphs of different population size at 2000 generations.

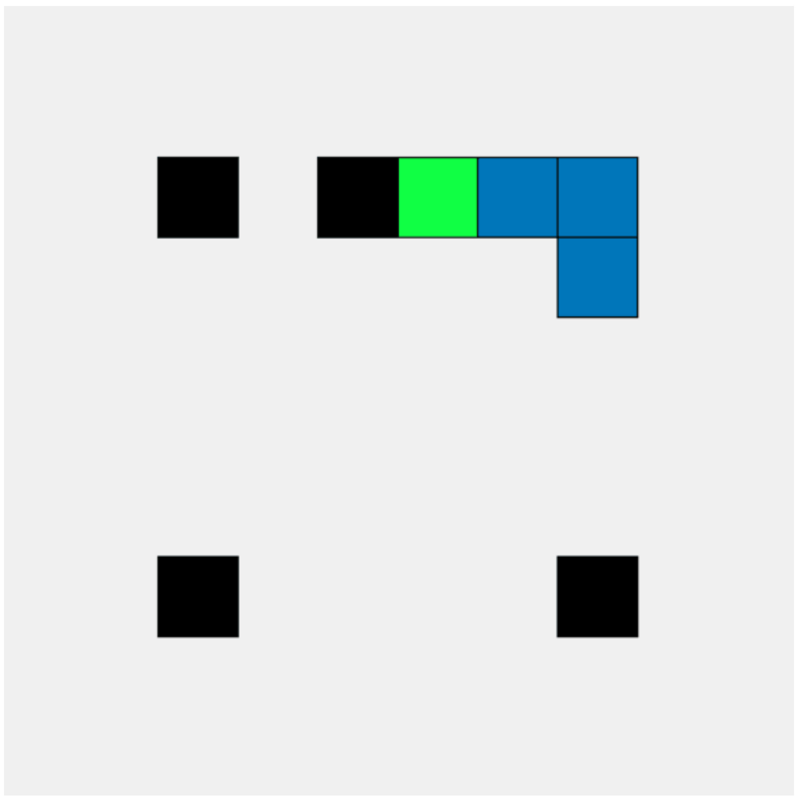






We noticed that initially the snake demonstrates a high learning rate, achieving above 10 in fitness (# of food eaten / snake length) then it's growth rate is dramatically reduced, even at 100 individuals running 2000 generations it could not top 17 fitness.

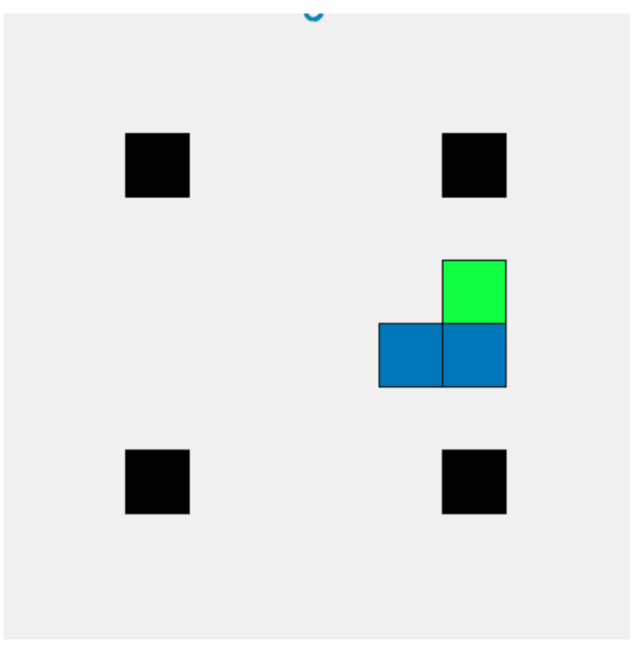
We concluded that the problem lays in our simple implementation of the individual brain, although we did our best.



Simulation of an individual that reached 13 fitness

(It had decided to end its existence abruptly although we strongly advised it not to by giving that option a negative value, some tragedies cannot be prevented I guess, we deduced it has recently had its heart broken)

FLUFFLY



Our dear fluffy reached 17 fitness like the true champion he is before also ending his own life

Conclusion

The amount of generation directly correlates with the individual's ability to play snake, given enough generations (how many they may be) we believe we can produce near-ideal snake player, we also think some sort of depression infected our snakes, hence their tendency to end their life by their own volition.

(we've checked and we definitely strongly manipulate against said choice, alas they still sometimes chose to do so).

Bibliography

[1] https://en.wikipedia.org/wiki/Darwinism

[2] https://www.techtarget.com/whatis/definition/evolutionary-computation#:~:text=Evolutionary%20computation%20is%20a%20sub,many%20variables%20for%20traditional%20algorithms.