Evolutionary Computation Open Individual Assessment

Exam Number: Y3868343

Introduction

Evolutionary computation has been a widely researched and developed area especially in recent times due to an increased interest in artificial intelligence. This area focuses on the training of programs / functions to complete a given task using techniques based on biological evolution i.e. individuals crossing over and mutating from generation to generation until a suitable agent is created. The following report will detail the implementation of these techniques to develop an autonomous agent to achieve a high score in a provided game of snake. The goal of the game is to control the movement of a segmented snake sprite as it collects food sprites scattered on a 16x16 grid. Every time a food sprite is collected, the length of the snake sprite is increased by one to increase difficulty. The snake is constantly moving and must move one square per tick in an orthogonal direction (left, right, up, or down). If the snake collides with a wall, collides with itself, or does not collect any food within a time limit, then the game ends and the final score is outputted. Score is defined as the number of food sprites collected.

To solve the problem of achieving an effective agent for the given snake game, I will utilise a rule-based syntax tree to represent the agent's decision process. Most other attempts at solving snake games opt for a neural network approach [1] [2] but I have decided to instead use a syntax tree due to the simpler structure in the hopes that training is more efficient and because I found evidence of working examples in literature [3]. The game used also provides some simple preinstalled sensing functions that produce a Boolean output depending on whether a wall/food/tail sprite is directly in front of the snake which is more appropriate for a syntax tree. If I were to implement a neural network, I might need to alter these sensing functions to output the distance to sprites instead of just detecting their presence. These functions will be used initially to gauge the need for other and more complex sensing functions.

The only other stated rules for evolving my solution is that the run_game() function be used in evaluation, route solutions cannot be hard-coded, and the movement system of the snake must remain unchanged.

Methods

Each individual in a generation is represented as a rule-based syntax tree that influences the snake's movement decisions. Example sensing functions were provided but these have since been adapted for the purposes of the agents. This is because after running initial tests, I found them to be insufficient at creating an intelligent agent even with several thousand generations. The original functions would simply detect the sprite type immediately in front of the snake's head, but the snake would seemingly never adapt to avoid obstacles effectively. I decided to adapt each detection function into four separate subfunctions that detect sprites in each orthogonal direction. This is partly to complement the orthogonal movement system which is in relation to the constant grid space as opposed to the original functions which would sense in relation to the snake's orientation. The other reason for this is to encourage agents to react to food either side of the head as these would otherwise have no effect on the agents' behaviour. After some more testing, I realised that local sensing would not be sufficient, so I adapted the food detection functions to have extended range to the edges of the grid space.

In total I then have 12 primitives / sensory functions which detect walls and tails locally and food globally in each orthogonal direction: “if_food_upwards”, “if_wall_up”, “if_tail_up” and their orthogonal counterparts. I also add three additional primitives, “prog2”, “prog3”, and “prog4” that give the tree sequential processing capabilities such that actions and decisions can be made consecutively. Terminals in the tree will run direction change functions during the next game tick; “turn_left”, “turn_right”, “turn_up”, “turn_down”. These are what I output back into the game to get the snake to react to stimuli.

Then I define the toolbox and the associated genetic functions to be used in the evolutionary process. Individuals are initialised with a ramped half and half method where half of the individuals are encoded with a full method and the other half, a grow method. Neither of these methods alone allow for a diversely shaped population of individuals so both are used. The ramped refers to how a range of depth limits are used to help ensure that trees are generated with a variety of shapes and sizes.

Agents are selected from each generation using a double tournament algorithm that performs the usual tournament selection technique twice. This is a form of selection pressure and helps relieve bloat in the agent’s tree. Bloat is a common issue with tree representations where the anatomy of a tree gets increasingly complex during mating and mutation. This has been tackled in additional ways in the solution which will be discussed later. Double tournament parameters are as follows: fitness size of 7, parsimony size of 1.4, with fitness first argument. Agents are mated and mutated using one point crossover and subtree systems respectively:

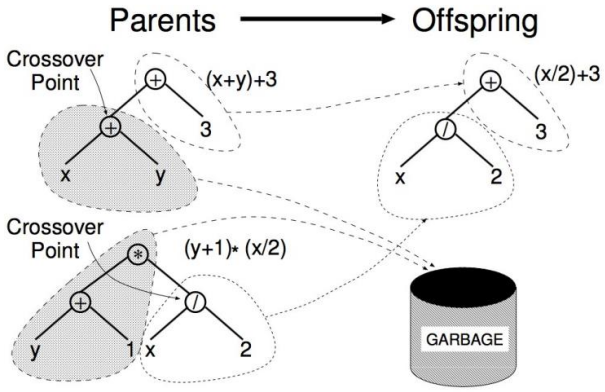

Figure 1: Crossover

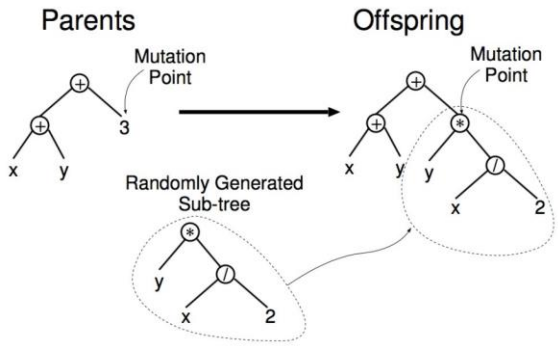

Figure 2: Mutation

In one point crossover, a non-terminal point is selected on both parent trees and the trees are clipped and merged. In mutation a subtree (of specified depth limits) is generated at a terminal point. For the purposes of the agents, mutation depth is limited between zero and five additional layers to promote any necessary complexity of the decision tree.

Agents are evaluated by a multi-objective fitness after playing the game three times. The purpose of this is to reduce the chance of snakes getting lucky with random food placements. After some initial testing, three fitness parameters were decided on; food collected (weighted by 50), steps taken (weighted by 1) and tree height (weighted by -0.1). Food collected is an obvious choice because that is the main objective of the game, hence the high weight. Steps taken is a secondary objective I have included so that snakes that live longer are deemed fitter even if they collect no fruit. This is useful for helping agents figure out to avoid obstacles and promote survival. The final parameter is the tree height which I have included to negatively effect the agent's fitness to further reduce bloat and promote smaller and more efficient decision trees. This has lowest priority so the weight is set to a small negative value.

Before any generation is created, a logbook is registered to keep track of evolutionary statistics such as mean, standard deviation, and minimum/maximum fitnesses to be used to analyse the results. For the purposes of this evolution, 500 individuals are generated 200 times with mutation and crossover chances of 20%. These are standard generation parameters and even when other values were tested, no significant improvements were found. Within the actual generation loop, offspring are crossed over and mutated from the parent population while revalidating their fitnesses.

Results

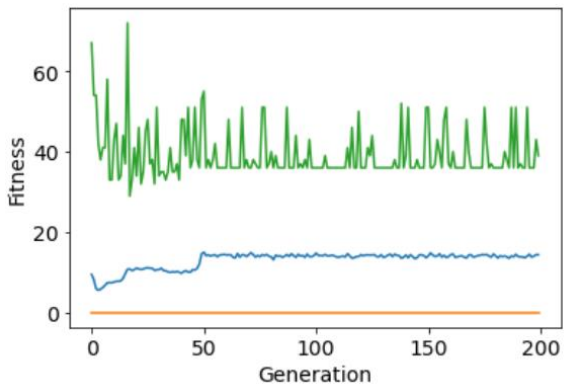

Figure 3: Fitness graph

As mentioned before, a logbook is created and filled over the course of 200 generations that stores maximum, average, and minimum fitness. (Figure 3) shows the improvement of agents over 200 generations reaching a plateau around the 50th generation mark with an average fitness value of ~17. Remember that fitness is

affected by three variables; food collected, steps taken and tree height which can explain some of the sporadic lines.

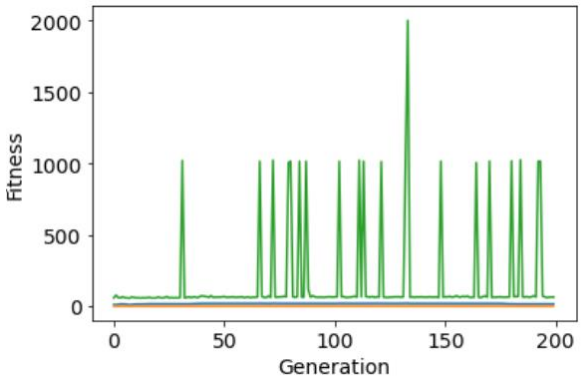

Figure 4: Spikes

Graphs are not always consistent as shown above. Sometimes after running the evolution, we can observe large spikes in fitness data implying that there exist agents in that generation that manage to complete the game and get absurdly high scores. Why this does not appear to be consistent throughout all generations is unclear, but this is probably due to lucky food placement and snake routes.

As for the agents' actual performance itself, we can observe some clear competence playing the snake game. Agents learn to avoid wall collisions by following the edges of the play grid round until food is detected on the axes. Agents' then turn appropriately to face the food and travel towards it. In most cases, the evolved agents would finally game over by colliding with its tail in pursuit of food behind it or even just when the snake's travel route encircles the agent with nowhere else to go. An example of an agent's evolved decision process is shown below:

if_wall_up(if_food_downwards(turn_right, if_food_rightwards(turn_up, turn_down)), if_food_upwards(turn_left, if_food_downwards(turn_right, if_food_downwards(turn_right, if_food_rightwards(turn_up, turn_down)))))

As an extra exercise, I also ran the evolutionary algorithm again but with the original primitive decision set to evaluate if my improved sensing functions made a substantial difference in the learning rates of agents. (Figure 5) compares how these two approaches improved over the course of 200 generations displaying mean fitnesses along with each generations' fitness standard deviation represented by shaded regions either side of the mean line.

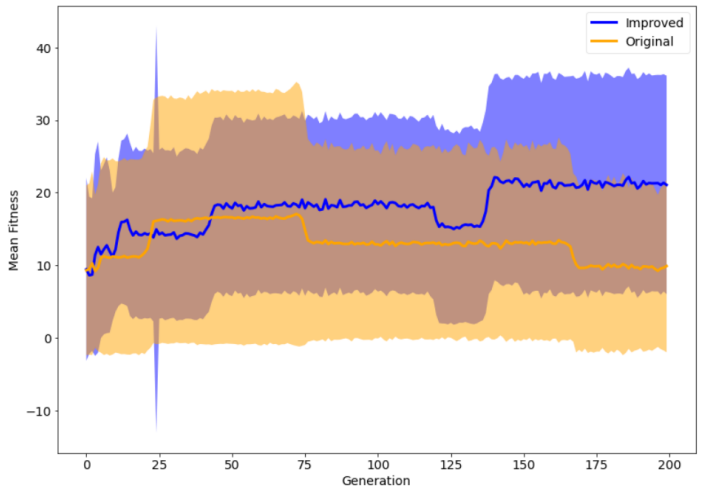

Figure 5: Sensing Function Comparisons

As shown by (Figure 5), the improved sensing functions reach greater fitness at the end of the evolution cycle while the original seems to drop slightly in later generations. This could possibly be because agents begin by learning to avoid walls, but the lack of efficient food detection (only local sensing available) restricts growth, so agent trees begin to bloat in an attempt to fix this issue and, in turn, lower the overall fitness. We can also observe greater variation in the original designed agents at the start of the evolution cycle possibly due to the limited sensing capabilities causing needlessly complicated trees that produce wildly different results. Although mean fitness seems better in my improved sensing functions, the mass amount of overlap in standard deviation areas could imply that the two distributions belong to the same population. However, when performing a Mann-Whitney U test, we get a p value of 0.000 which signifies a highly likely chance that the two distributions are distinct.

Conclusions

As shown by our results, I have managed to create an agent capable of amateur skill to play the provided game of snake. Some agents even manage to complete the game and achieve what appears to be near maximum scores if food placement is lucky enough. Snakes can react to their environment effectively, avoiding wall and tail collisions while turning to collect food sprites, only failing in the event of more complicated snake routes and decisions.

For future investigations, it might be beneficial to approach the design with a neural network architecture which allows for more complicated behaviours. This approach is often used in most past efforts [1] [2] so it might produce a better solution. If the syntax tree architecture were to be used again however, I might implement more complicated sensing functions that might allow agents to avoid their own tails during complicated food collection routes. Within the researched literature [3], example sensing functions use the snake's current movement direction as a decision which would save agents from having to figure out their direction otherwise. The cited paper also altered the relative direction of the senses to match the snake's movement similarly to my implementation. It might also be beneficial to readjust the fitness calculation to minimise the spike anomalies seen in the fitness timeline as well as giving a more intuitive understanding of how fitness is calculated.

Regardless of any future changes however, I believe the produced solution acts a suitable benchmark for future work and shows excellent potential as an autonomous agent for the provided snake game.

References

[1] P. Białas, *Implementation of artificial intelligence in Snake.*

[2] C. S.-L. B. Halmosi, *Learning to play snake using neural networks.*

[3] T. Ehlis, *Application of Genetic Programming to the Snake Game,* Myopic Rhino, 2000.

End of Paper