EVCO Coursework

Module Code: COM00071M
Exam Number: Y3856226

Introduction

Background

Snake is a classic arcade game from the 1970s, primarily focused on moving a snake around an enclosed space and collecting food. In single-player variants, the player loses if they collide with a wall or the body of the snake and points are gained by collecting food. This paper shall investigate the use of Evolutionary Algorithms (EAs) to create an autonamous agent for playing snake.

EAs are algorithms inspired by biological evolution, creating a population and carrying out evolution through selection, reproduction and mutation. The algorithm is evaluated using a fitness function to evaluate performance, which makes them very effective at solving complex problems due to their ability to explore of complex space efficiently [1].

The Snake Game

In this instance of the snake game, a 16 by 16 grid is used, with a snake of initial length 11. At each step the snake will move one grid space in whichever direction it turns; up, down, left and right. Importantly this means that the EA will have to learn a representation of relative direction, or it may crash into itself and lose. Initially, one piece of food is placed randomly, with another piece being placed each time the previous one is eaten. The agent is awarded 1 point for collecting food, and the length of the snake increases by 1. The snake can lose by hitting a wall (reaching the edge of the grid), crashing into itself or not consuming food within a turn limit. Starvation is an important factor, as it helps prevent the snake from looping which will reduce CPU usage and encouraging more efficient solutions.

The aim is to obtain the maximum score, however, this becomes difficult as the agent's score increases due to the increasing snake length. This makes the grid more difficult to navigate, so the snake must be able to efficiently navigate towards food, which requires a complex agent.

Related Work

Yamini et al. [2] investigates the use of Genetic Algorithms (GAs) and Neural Networks (NN) to create a multi-purpose agent for game developers to use in non-playable characters. A GA is used as it helps accelerate the learning process compared to the tremendous computational power required for NNs. The GA was used to help generate weights and biases, with the best members of the population selected by game score each generation. This is a very effective method of creating a game agent and it was able to learn quickly, meaning this could be a potential approach for a snake agent.

Rather than using a NN, Yeh et al. [3] takes a look at snake game AI, using a decision function. The decision function consists of; smoothness - a bonus to moves not making turns, space - the number of positions reachable for each decision, and food - a score based on which type of food and how much food is eaten. This function is then tuned by using an GA to weight these functions, with each gene being a weight and fitness being measured as the game score. This shows the effectiveness of GAs in optimising in complex search space.

The chosen inputs (or senses) are very important as Halmosi et al. [4] investigated, finding that it was important to include senses for food and walls in each direction. It was also found that scaling their inputs had either no effect or was detrimental to the performance of the algorithm, therefore this will be avoided when constructing the agent. Finally, they found that with a population of 1000 and 500 generations, their algorithm could get an average of 8 food, often being trapped by itself because it only sees the grid squares around the head, suggesting a greater measure of danger or spacing as carried out by Yeh [3] may be important.

Bialas [5] produces an in-depth analysis of different parameters for a GA and NN used as an agent for the snake game. Importantly, it was concluded that a variation of roulette selection, that uses the top 20% of the population, provides more diversity than tournament selection where the best snake is likely to win every group.The paper also concludes that uniform crossover performs best, so it will be important to test a variety of selection and crossover methods including those proposed by Bialas.

Ehlis [6] uses Genetic Programming (GP) to play the classic Nokia mobile Snake game. This is a brilliant paper as it not only shows great performance compared to other papers, scoring 123 out of 211 with basic its initially function set, but it also dicusses improvements on this function set and the impact this has on the snake's performance. Interestingly, it includes a discussion on particular weaknesses the snake develops based on limitations to its primitive set. With an improved set of primitives, the suggested algorithm can reach a perfect score, apart from a very limited set of scenarios, therefore the use of GP must be strongly considered.

Based on prior research, including the papers above, there are two key approaches to be taken when designing a game-playing agent using Evolutionary Algorithms; Genetic Programming (GP) and Neuroevolution (NE).

A Genetic Programming Algorithm (GPA) is a type of EA in which a population of random programs are generated, typically represented in the form of trees. These are built from a set of terminals and a set of primatives (functions). The terminal set consists of inputs and constants (pi, x) or zero-argument functions, often for controlling an agent (moveForward()). The primitive set consists of standard programming constructs such as; arithmetic functions (+, Cos), boolean functions (and, nor) or custom functions with arguments (check_danger_ahead()). These programs are generated like trees, randomly selecting primitives at each branch, and terminals at leaf nodes. At each generation crossover and mutation can occur, on whole subtrees or nodes, with fitness being measured and selection occurring. GPs are very strong as they are a natural representation for many problems and could be suitable for the snake game [6], able to evolve simple representations such as ”if_food_left then move_left”, so these algorithms should be strongly considered and included in my testing.

Neuroevolutionary Algorithms (NEA) may also be a suitable approach, as due to NNs ability to learn relationships. There are two approaches to NEAs; evolving the weights of a neural network or evolving the structure of the network itself. Due to the relative simplicity of the snake game, evolving the network structure may be overly complex and lead to overfitting, so evolving network weights will be the focus. This process works by randomly initializing the weights between neurons, which are then mutated and crossed over before being evaluated for fitness via game performance. The process of evolving weights has been shown to be effective by Yamini [2] and Yeh [3], and due to the wide use of NNs for game agents should be tested for performance in the Snake game.

Methods

The Python package Deap will be used to implement the chosen EAs, primarily due to the wide range of tools available in its toolbox and in-depth documentation. Deap is designed for rapid prototyping, quickly allowing for the creation and modification of EAs, which is suitable for the testing and development of multiple EAs. Other libraries were considered, but Python tools such as gplearn were less suitable for this problem lacking some functionality.

GPAs vs NEAs

As discussed above, both GPAs and NEAs have been used effectively for creating game agents, therefore should both be tested against our snake game. To do this, the performance of both shall be examined with a simple set of parameters and equivalent sensing functions and fitness evaluation.

As shown in the papers at above, the game score is an effective measure of an EA’s fitness. Therefore for the initial tests, the fitness shall be calulated as 1 fitness per food eaten by the snake. With regards to sensing functions, there are two key sets of information; the location of obstacles (walls and the snake itself) and the location of food. Halmosi [4] found that adding sensing functions to detect food and obstacles directly next to the head was effective in producing a good agent, therefore, to fairly test the performance of the GPA and NEA methods, the sensing functions and actions shall be limited to those in Table 1.

While it was important to keep testing methods consistent and fair, some adaptions were made to optimise the performance of both methods to get the most out of the implementations. For the NEA the structure of the NN was tested extensively due to its

Actions	Food Sensing	Danger Sensing
move_up	if_food_up	if_danger_up
move_down	if_food_down	if_danger_down
move_left	if_food_left	if_danger_left
move_right	if_food_right	if_danger_right

Table 1: Functions used for Testing the GPA and NEA.

potentially high influence on performance. It was found that a structure of 8 neuron input layer, 6 neuron hidden layer and 4 neuron output layer (indicating a probability of moving in each direction) performed best, with a higher number of neurons and layers often overfitting and performing inconsistently and poorly. Tournament selection was found to perform better than both Best and NSGA2 selection, while uniform crossover was more suitable than one-point crossover and gaussian mutation performed well.

The GPA was set up similarly with primitives and terminals according to Table 1, however, one key problem encountered was bloat. Due to this, genHalfAndHalf was used for creating and mutating individuals. Similar to NE, tournament selection performed best, however, double tournament selection was chosen to counteract bloat and uniform mutation and one point crossover were both more suitable than the alternatives.

Both algorithms were tested with a population of 100 across 100 generations and run 5 times, the best run was selected from each and is displayed in Figure 1 and Figure 2. It must be noted that these parameters are reasonably small, but they had to be kept low due to resource limitations and high runtimes of the NEA. Despite this, these parameters have been kept consistent and runs were repeated to ensure accurate results. It is clear from Figures 1 and 2 that the GPA performs significantly better than the NEA, reaching an average of 19.45 compared to 12.42, while also showing a more consistent upward trend.The GPA also performed significantly faster than NEA, able to run over 10 times faster, making the GPA a much more suitable candidate under computing restrictions. Therefore, GPAs will be investigated further.

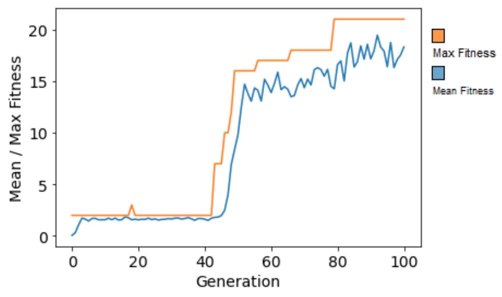

Figure 1: Best Fitness GPA from test runs.

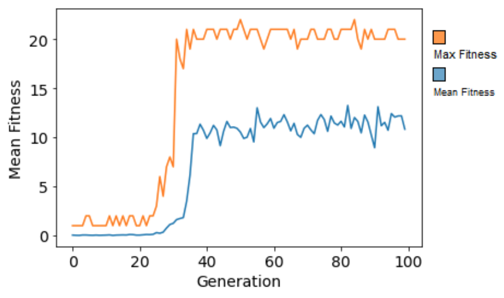

Figure 2: Best Fitness NEA from test runs.

Results

Due to performance limitations, there was a compromise on population size and the number of generations in order to provide adequate evidence of performance while balancing computation time. After some testing, it was decided that a population of 2000 and 500 generations was suitable. This also allowed for any changes to be tested 3 times, as the routine can evolve differently due to the small number of generations and performance can vary greatly. Then to test further, the few best performing individuals could be taken and tested for a population of 2000 population and 10000 generations to investigate performance over a much larger period, however, these could not be run multiple times due to the performance limitations.

The fitness function remained as the number of food eaten by a snake. Adapting the reward function to reward snakes that survive longer, by adding a small bonus based on the time survived was also considered. This was initially tested by adding a score of 1/1000th or 1/100th of the number of moves survived, but this was found to have little to no impact. While this may have been effective in encouraging survival if the snake was struggling to find food, the snakes performed well without this inclusion and snakes which survived longer typically ate more food and therefore, had a higher fitness. The only issue found was that performance may be inconsistent per run, so the fitness was chosen to be the average amount of food eaten across 3 runs.

The choice of evolutionary algorithm was also considered: with deap offering a simple ea (eaSimple), $\mu + \lambda$ (eaMuPlusLambda) and μ, λ (eaMuCommaLamdba). Through a series of tests carried out on the same populations, it was found that eaMuCommaLamdba was the best choice as it consistently outperformed both eaSimple and eaMuPlusLambda when using equivalent parameters.

As mentioned in the introduction, starvation was an important factor in ensuring the snake didn't learn to follow an infinite looping pattern, therefore a feature was added to kill the snake if it went 500 moves without food. There is also the added benefit that this also discourages solutions that survive a long time without food. An alternative to this was to reduce the fitness of the snake the longer it goes without food, but this would also penalise snakes who survive for longer, which is a desirable trait, so this was not implemented.

From the testing above, we can see that sensing functions allow the snake to detect food and danger directly next to its head was effective in producing a working agent that could achieve a reasonable score. Halmosi [4] suggested that this could be improved by giving the agent information on the distance to obstacles or food. While this isn't directly implementable in GPs, Ehlis [6] found that providing senses to incidicate if the food was upwards or right of the snake improved an agents performance. Because of this, the ability to sense food downwards or right of the snake was implemented and had an immediate impact on performance, greatly improving the snake's ability to move towards food, as this wasn't possible before unless they were directly next to food.

Actions	Food Sensing	Danger Sensing
move_up	if_food_up	if_danger_up
move_down	if_food_down	if_danger_down
move_left	if_food_left	if_danger_left
move_right	if_food_right	if_danger_right
	if_food_downwards	if_danger_two_up
	if_food_rightwards	if_danger_two_down
		if_danger_two_left
		if_danger_two_right

Table 2: Terminals and Primatives used.

Due to the positive impact from the above changes, further improvements to food sensing were considered. One iteration of this was allowing the snake to detect if the food was in the same row (leftwards or rightwards of the head) or same column (upwards or downwards from the head), allowing the snake to navigate towards food more accurately. Initially, this appeared to be a poor choice and performance worsened, however when food sensing functions for directly next to the snakes head were removed, performance improved greatly. This is likely due to there being some overlap between the two senses, causing some interference with its ability to learn about its environment.

As the snake became more efficient at finding food, it began to trap itself and inevitably crash, caused by the snake only being able to detect danger around its head. To improve its ability to detect danger, the ability to sense danger two grid squares away in all directions was added. Other approaches were considered, such as a method similar to the food senses implemented, in which the snake would be able to detect danger in the same row or column, however it was decided that this would not be effective due to the high probability a long snake would occupy at least one square in all rows or columns. This approach worked well, and while performance was not perfect, it helped the snake's behaviour improve.

Since the snake could now know if food and danger were two grid squares away, adding the Prog2 function was considered, allowing multiple moves per cycle. However, during testing, Prog2 reduced the performance of the snake consistently and so was removed due to this. Following this, the final set of primitives and terminals were set and can be found in Table 2.

The next stage of testing was to decide the methods for creating individuals, selection, mutation, and mating. The key issue faced was the size of trees, which often experiencing a large amount of bloat. Therefore, measures will be taken to prevent this, primarily through the use of double tournament selection as suggested by Luke [7], allowing selection pressure to occur based on fitness and size. This worked well and a double tournament selection provided better results than standard tournament selection and alternatives like Roulette and NSGA2. Different values were tested for tournament

	Group 1	Group 2	Group 3	Group 4
Crossover Probability	0.2	0.2	0.1	0.05
Mutation probability	0.1	0.05	0.05	0.05
Min Individual Depth	3	1	3	3
Max Individual Depth	10	10	10	10
Min Mutation Depth	1	1	1	1
Max Mutation Depth	3	3	3	3
Tournament Size	7	10	7	7
Tournament Parsimony	1.4	1.7	1.4	1.4
Average Fitness	24.5	10.2	73.6	97.2
Fitness Standard Deviation	13.8	3.1	21.8	19.6
Max Fitness	33	14	86	105
Average Size	113	21.1	67	62.9
Size Standard Deviation	8	2.8	5.7	3.86

Table 3: Parameter testing for 10000 generations.

and parsimony size but values of 7 and 1.4 were chosen respectively, which kept the tournament focus on fitness while applying size pressure.

It was also decided that further measures should be tested to counteract bloat and overfitting, resulting in the use of genHalfAndHalf being used to generate individuals and mutate them, rather than genFull. These two methods were very effective and bloat was rare, especially considering the reasonably large number of primitives. Static limits on tree depth during mating and mutating were also tested, but this only reduced performance, likely due to the wide range of anti-bloat measures already in place.

Aside from this, as recommended in the previous section when testing EAs, one-point crossover and uniform mutation were used as they performed well in previous tests and are the recommended method in the DEAP Library documentation [8].

Finally, parameters were tweaked to maximise the performance of the GPA, which included: crossover probability, mutation probability, individual min depth, individual max depth, mutation min depth, mutation max depth, tournament size and tournament parsimony. A wide range of parameters were tested and the most significant results were selected to be tested for 10000 generations: the results for this can be seen in Table 3.

It is clear from Table 3 that the parameters in Group 4, performed the best in both on fitness and size by a large margin. The fitness development for this can be seen in Figure 3. These parameters were re-tested to check the performance and as can be seen from Figure 4, a fiitness of 96 was reached within 200 generations on its best run. This indicates that future testing over a larger population and number of generations could be useful, but it is likely there were other restricting factors, as a score of 100 or greater was never reached, most likely due to limitations in the primitive or terminal set.

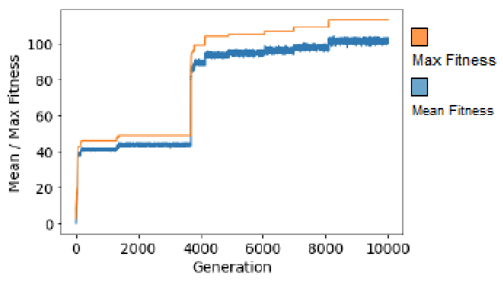

Figure 3: Best Fitness of Group 4 GP for a population of 2000.

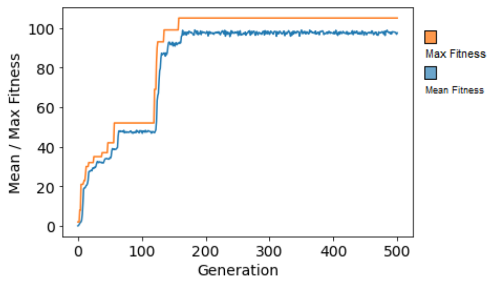

Figure 4: Best Fitness of Group 4 GP for a population of 2000.

Conclusion

Conclusion

This paper has presented an investigation into the use of evolutionary algorithms to create an agent for the classic snake game, and the use of a genetic algorithm to solve this. The agent has been evaluated through the testing of different primitive set features and parameters for evolving the agent. While the results did not produce an optimal solution to the game, they obtained a good score and comparisons between runs show successful improvements through certain avenues of exploration. This paper builds on the work by Ehlis [6], advancing work further by solving the problem on a different sized grid and experimenting with a more limited terminal set, in which the snake isn't able to make relative turns, which grealy increased the complexity of the problem.

Future Work

The work presented in this paper provides many opportunities for further development. It is clear that the removal of relative directions has impacted the fitness of an agent, therefore it may be interesting to investigate how a more complex terminal set affects bloat and performance, and following on from the increase performance when providing global food information the impacts of a larger primative set.

There are also other avenues; multiple snakes could compete through co-evolution to pursue food while avoiding the same obstacles and each other, or in a competitive game attempting to trap each other similar to the Tron light bike game. The problem could also be generalised into a navigation based problem, where an agent has to navigate around a more complex environment such as an office or maze.

References

[1] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, Morgan Kaufmann Publishers Inc., 1989, pp. 762–767.

[2] R. Yasmini and A. Jain, "General artificial intelligence model for developing adaptive non playable characters in computer games," in *Journal of Critical Reviews*, 2020, pp. 78–82.

[3] S. et al., "Encountering stronger password requirements: User attitudes and behaviors," [Online]. Available: `https://cups.cs.cmu.edu/soups/2010/proceedings/a2_shay.pdf`.

[4] B. Halmosi and C. Sik-Lányi, "Learning to play snake using genetic neural networks," in *Pannonian Conference on Advances in Information Technology (PCIT 2019)*, vol. 4, 2019, p. 126.

[5] P. Białas, "Implementation of artificial intelligence in snake game using genetic algorithm and neural networks," [Online]. Available: `http://ceur-ws.org/Vol-2468/p9.pdf`.

[6] T. Ehlis, "Application of genetic programming to the snake game," [Online]. Available: `https://www.gamedev.net/reference/articles/article1175.asp`.

[7] S. Luke and L. Panait, "Fighting bloat with nonparametric parsimony pressure," in *Parallel Problem Solving from Nature — PPSN VII*, Springer Berlin Heidelberg, 2002, pp. 411–421, ISBN: 978-3-540-45712-1.

[8] D. Project, "Symbolic regression problem: Introduction to gp," [Online]. Available: `https://deap.readthedocs.io/en/master/examples/gp_symbreg.html`.