COSC343 Assignment 2 Report: Genetic Algorithms

William Wallace
1216661
14/05/2020

Overview

This assignment was designed to get COSC343 students to develop a better understanding of genetic algorithms and how they are implemented. These learning outcomes were assessed within the context of adding Python code to a 2D grid-based, agent-based modelling game, with the goal of evolving a player's creatures evolve as best as possible.

My Approach – rationales, specifications, and observations

myAgent Algorithm

Arguments: `self, percepts`
Returns: `actions`

The agent function was a key component in this assignment because the behaviour of a creature is dependent upon it, and so too are the subsequent generations' performance. For this reason, I decided to implement a base agent for its robustness and reliability, and then build upon that base with guidance from my agent model, wherein weights were adjusted by the chromosome. My function begins by randomly assigning values to the actions and tweaking them based on the perceptual information from each of the tensors' maps. After calculating the weights of each action from 0 to 6, the action with highest weights were picked by `myAgent`. The chromosome model I used worked in tandem with `myAgent` model as a defined list of several percept and object combinations, within each of the pre-existing tensors. Each of these combinations hereby be referred to as a gene. When an agent is first initialized, each gene was given a random action, with a random weight to add. This is the exact same the percept action-map, instead the agent only reacts to certain percepts. Weights tested were 0.1, 0.25, 0.5, 0.75 and 1.5, and after evaluating effectiveness of these weights in worlds containing the default parameters, further finetuning was performed, with 0.05 increments. Percept maps were programmed as follows:

food_map: where there is a strawberry detected in the creature's immediate vicinity, adjust the weight responsible for eating the strawberry by the number of actions taken up to that event. I came to the weight being adjusted by the number of turns because I observed that if the weight was too high for eating a strawberry, my creatures would search for strawberry, even at the cost of running into creatures. This is a costly mistake because the positive reinforcement of the increased energy is counteracted by the negative reinforcement of damage or death to the creature, causing the average survivability of the creature to decrease overall. Constrarily, if this weight adjustment was too low (for example <0.5), I found my creature to not take advantage of the essential and accessible energy-source, and instead placing more value in non-vital actions such as repetitive movements from left to right, for example.

creature_map: where there is an attacker in the immediate vicinity, evaluate own size, and adjust the responsible weight by the number of actions taken up to that event if relatively big (otherwise move away). This mapping was rather fraught with difficulty during the testing stage for me. This was due to the way that monsters of a similar weight would tend to be unpredictable, and moreover, loops would develop in which big creatures would chase smaller monster, resulting in maladaptive behaviours and therefore reducing the genetic algorithm's overall success. I was able to tweak the combinations of variables such as movement to eventually get the command to work as planned, albeit less effectively than hoped for.

wall_map: where there is a wall in the immediate vicinity, increment the weight responsible for movement away from it by the number of movements taken up to that event. Though this may not be the most beneficial or important action to take given the percept, it ensures that the creatures are more likely in an environment where they will learn rather than not. I implemented this because I observed my overall algorithm to learn slower without it. I hypothesise that this is because a creature's adjacency to a wall necessitates shelter from attackers (due to having one less vulnerable flank) but also one less flank for which to acquire energy from strawberries or other creatures. Consequently, I found it more beneficial in terms of evolution for a creature to be in environments of change – for better or worse - as opposed to idleness.

Chromosome Model

Before explaining my current chromosome model, you can view observations and evaluations of a previously implemented model an it's mapping in the Discussion section of this report, for further insight.

My current chromosome model was initialised to contain random uniform floats between -1 and 1, each float corresponding to the possible actions of the agent. My rationale for doing this was primarily for understandability and modifiability, though other advantages include testability and overall performance resulting from these benefits. The relatively few number of values made it more straightforward for me to develop and prototype by circumventing the burdens of keeping track of hundreds of input/outputs configurations, whist simultaneously still maintaining flexibility and functionality by being able to scale the weights with precision.

I chose to start off with random values between 1 and -1 for two reasons. Firstly, random values were primitively required for the first generation, or else the chromosome would not truly involve learning from random actions, and part of the purpose of the assignment would be undermined. Secondly, I found the values of -1 and 1 to be sufficiently large enough to allow for positive and negative reinforcement of adaptive and maladaptive behaviours, respectively. Such behaviours define being attracted to positive stimuli such as strawberries, and repulsion from negative stimuli such as attackers. Additionally, I chose to make the random numbers conform to a uniform distribution so that the effect of my learning algorithm on generations was more predictable, since non-uniform numbers would occasionally result in poorly performing generations when the seed was poorly suited to the creature's environment. I took inspiration for this range from the hyperbolic tangent (tanh) activation function which has similar advantages to the ones described above (Karlik & Olgac, 2011).

Genetic Algorithm

My approach of implementing the genetic algorithm can be outlined in four main areas – the fitness function, selection, crossover, and mutation – all of which are called in the `newGeneration` function. The latter three steps are given by three respective functions but are carried out for each individual in the old population until a new population matches the old population in length.

Fitness function

Returns: `score`

Establishing a well working fitness function proved to be one of the most difficult and crucial challenges of this assignment, since it provided the metric by which success is quantified. After iterating through combinations of variables, I found that the function where the sum of a creature's time of death (in turns) and that same creature's number of enemies eaten and number of strawberries eaten divided by 4.25, worked most effectively. Additionally, I coded the function so that creatures that survived would obtain a bonus of 30, to positively reinforce the creature to perform in future generations. A fitness value, `score`, is returned for each creature.

The effectiveness of this fitness function was derived from learning about - and counteracting - my observation that creatures would run into monsters when the time of death was added to its energy alone. I hypothesise that this is because the benefit of dying earlier meant that the creatures had conserved more energy before their time of death. Compounding this, was that these "suicidal" tendencies occurred more often than those that ate and survived, since time until death and energy both have upper limits of 100 but aren't weighted equally valuable according to the specifications of the game.

Tournament Selection function

Argument: `subset`
Returns: `parent1, parent2`

I decided upon tournament selection for my final submitted version of the assignment, though I tested both tournament selection and roulette selection in the developing process. I found tournament selection to be more effective and modifiable, since I had more control over which individuals I could choose to cross in the next step, whereas roulette selection was, by its nature, left entirely to chance regarding which individuals would be chosen to cross leading to less fit creatures being selected, so I removed it. Tournament selection was implemented by choosing the highest and second highest fitness scores from a random group of approximately 33% of the creatures in the old population (so if the default population size is 34, 33% of 34 means that 11 individuals are in the subset). I decided on a third of the previous generation's population because it gave enough variation in the chosen creatures such that creatures struck a balance between performing well and performing exceptionally well. I found this balance to significantly affect the algorithm's performance over the generations: if the parents perform poorly, their child will perform even worse and the algorithm won't learn; if the parents perform too well, they may not have enough variation between them, which, over time, leads to early specialisation and the potential for convergence towards local minima, not global minima.

I found that adjusting this subset value changed how quickly and reliably my player learns. In the four tests that I conducted under default settings, I could reliably arrive at an average increase of 35% of the fitness function within the first 100 generations when the subset was at 60% of the population size. Contrarily, I found that when the subset was at 10% of the population size, I could reliably achieve an average increase of 32% of the fitness value within the first 300 generations before capping out and not improving. This goes to show how severely the subset size influences the rate of learning in creatures and the result of these tests prompted me to use 33% of my sample size – a value that gave a higher average fitness increase of 45%, whist also achieving this in a reasonable number of 200 generations after effectively capping out.

In implementing this function, I stored the fittest individuals in a list and sorted it according to their fitness value and returned the first two indices of the temporary list. For convenience, I named chromosome with the highest fitness value of the group `parent1`, and the second highest fitness value of the group to be `parent2`.

Crossover function

Arguments: `parent1, parent2`
Returns: `child_object`

Crossover is the process for choosing how the two selected individuals merge their weights in each chromosome into the chromosome of one offspring. In my crossover algorithm, an index is randomly assigned along either of the parents' chromosome such that indices of the chromosome up to and including that point are included in the offspring's chromosome, while every index after that point is assigned to the child from the other parent. In my implementation, `side` is the variable that decides which parent's indices go on which side of the child's chromosome.

In testing, I found that the location of the crossover had little effect on the success of my learning algorithm. Originally, I had a hardcoded crossover point of at the third index, and this was useful for more easily control the environment upon which I was testing different combinations of parameters. After researching however, I found it was more conventional to have a random crossover point, and it more closely resembles how fertilisation occurs in real-life. As a consequence, I changed this function to reflect that.

Comparing the two implementations – that is, hardcoding the crossover versus randomly choosing it – confirmed my understanding of the crossover's role in the learning algorithm. A predetermined, hardcoded crossover point does not have a noticeable effect on learning algorithms when the fitness values from the two parents are similar, which was the case for my algorithm. Whether `parent1` or `parent2` donned their weight for the child did not matter much, since the child was sure to receive a random weight from either of the parents in the first place. Randomising in the location of the crossover therefore merely randomises what are already random weights – the child still receives a mixture of weights from each similar parent, meaning that the child's performance will not be better or worse in the future generations.

If it *had* been the case that a larger difference existed between `parent1` and `parent2`, hardcoding a crossover point would certainly influence the performance of my learning algorithm compared to randomly assigning it. This is because it would influence the distribution of successful weights – weights that aid in a creature's survival – to be passed on to the child. Indeed, my tests validated this claim. For example when I programmed `parent1` to contain the greatest fitness value and `parent2` to have the lowest fitness value, I found that making the crossover point closer to the beginning of the chromosomes lead to the child to inherit more behaviours of the poorly performing `parent2`, while the opposite is true when the crossover point is closer to the end of the chromosome, where the child inherits the more successful genes of `parent1`.

Mutation function

Argument: `child`
Returns: `mutate_child`

In ecology, genetic mutations result in added variation to gene pools, and the possibility to acquire new characteristics; ultimately resulting in increased fitness in species over time (Wright, 1932). In light of this, in the function named mutate, I implemented a function to give each child a 5% chance for weights in its chromosome to be reassigned to a new random float value between -1 and 1. The function worked by looping through each value of a child chromosome with a random chance to mutate it. This process cycles through each child, and each child is appended to the new population thereafter until it's population size matches that of the old population.

I arrived at the 5% mutation value using trial and error, and I gleaned two insights in doing so. Firstly, if I made the mutation value too low, for example under 3%, mutations seldom occurred at all, resulting in generations that would stagnate in local minima from time to time. Converging towards local minima is not as desirable as global minima because it implies that better solutions are possible but unattainable. Mutations allow species to escape this by chancing upon an experimental value that may beneficially propagate throughout future generations. Secondly, I found that making the mutation rate too high lead to marginally faster evolution rates at the cost of a more sporadic and unstable trends of my average fitness values. While I observed that local minima were occasionally escaped from, I also found my creatures to just as likely jump out of the global minima back into the local minima. These findings are illustrated in Figure 1.

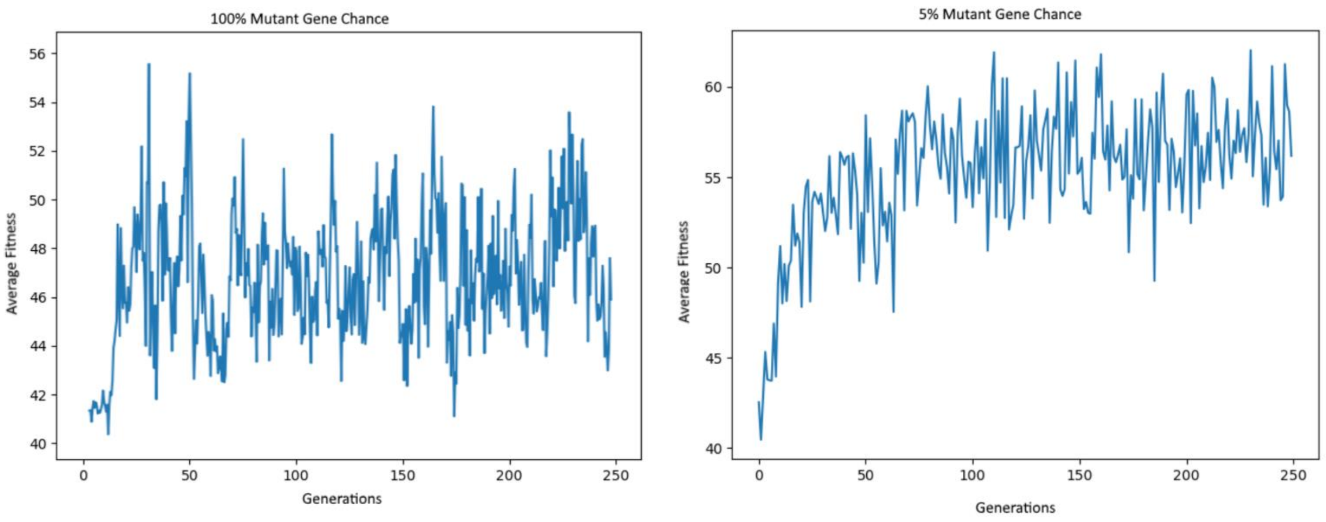

Figure 1. Average fitness for each generation over 250 generations for mutation chances of 100% (left) and 5% (right), respectively. N = 34.

My reasoning for having mutated weights fall in the range between -1 and 1 resulted from testing different ranges and finding that it resulted in the best average fitness value at the end of 500 rounds. I suspect that this finding is related to the similar finding of effectiveness for initialising the initial generation's chromosome. This range also provided continuity and a sense of compatibility between the initial generation's randomised weights and the mutated weights.

I also observed that mutations have greater effects on smaller population sizes. I attribute this to the effect of mutation-induced outliers being diluted by large sample sizes. Indeed, I found phenomena like these to be observable in biology, and cause stunts in evolution and adaptation in a similar way. Such examples include the bottleneck effect, genetic drift, and the founder effect (Maruyama & Chakraborty, 1975).

Discussion

Evolution of Behaviour – an analysis of results

The effect of my chromosome model, fitness function, and propagation resulted in creatures that generally possessed higher fitness values at the end of 500 rounds compared to creatures in the first round. Out of 5 tests that I performed using the default settings using different staring chromosomes, I found that all of them resulted in a winning more likely than not. Additionally, the average fitness of my creatures had shown an average increase of 52% over those 500 rounds. These results are illustrated in Figure 2.

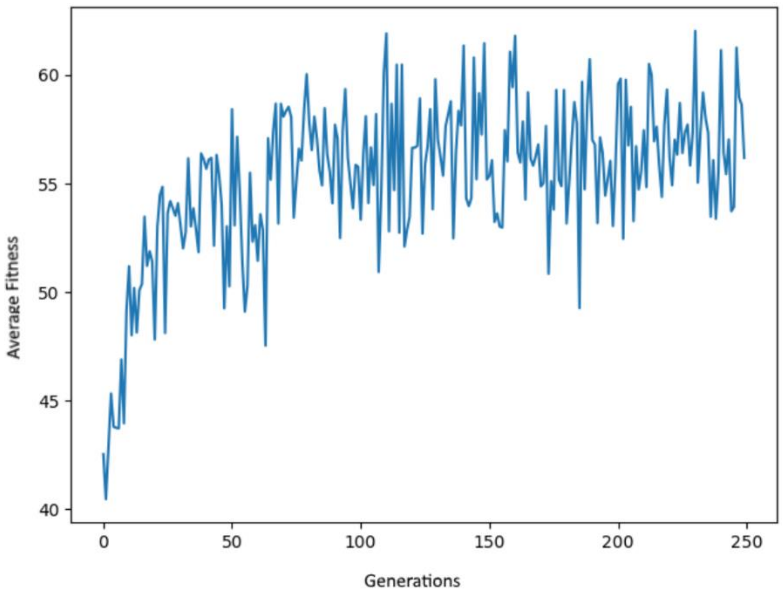

Figure 2. Average fitness for each generation over 250 generations using the final genetic algorithm, for the default settings of the game. N = 34.

Primarily, it appeared that my creatures adapted by usually eating strawberries when they were available, eating creatures when they were small enough to be eaten, and avoiding walls. Collectively these behaviours illustrate three very simple evolutionary lessons: eat when possible, avoid danger, and prefer movement over being idle. I also noticed that chasing behaviours existed even in the final game of the 500 rounds, indicating that there is still room for improvements in the context where settings are default.

A more complicated behaviour that I noticed, was the tendency for creatures to forgo eating strawberries when it the creature perceived enemies to be in the vicinity. This delay of gratification displays the beginnings of intelligent behaviour because it recognises that survival should be prioritised over immediate energy, since the creature can still gain energy in subsequent movements if it lives.

Another behaviour that I observed, was what I call *team-forming*. In this behaviour, creatures would occasionally delay attacking largely sized enemies – a battle that small creatures would most certainly lose – and instead wait until they had support from other creatures before attacking said enemy. This way, due to the baked-in mechanics of the game, two creatures would be able to overpower one large enemy, provided that their combined size was large enough. Admittedly, it took me a great number of tests to recognise this behaviour because of how inconspicuous it is, particularly due to the fact that it involves tracking the movements of three creatures. This made me wonder how many other behaviours I have not recognised that also lend towards increased survival for my creatures.

I/O Mapping and Chromosome Models – comparing an alternative approach:

Each of the decisions made for this assignment were tried and tested before implementing them in the final code. For the purposes of describing what I learned, I believe it is important to explain why some early approaches to this assignment did not feature in my final code. Specifically, I abandoned the approach of mapping each of the 75 possible percepts to each of the 7 actions in my creature's chromosomes.

In trying this approach, I applied the *flatten* method provided by the numpy library to construct a list containing 75 elements, each corresponding to a percept, and I initialised a creature's chromosome as a list containing 525 random weights, rather than 7 in my working model. The 7 sets of 75 weights (e.g. weights form 0-74, 74-149, …) within this chromosome then corresponded to the 7 possible actions given the 75 inputs. Each of the elements in the output were assigned as the evaluation to the expression $x_{[i]}*w_{[y, i]}$, where i was the index from 0-75 for each input, and y was the action. This mapping expression was inspired by widely adopted neural network architecture.

I found one advantage of this approach was that there was a greater control and visibility of how myAgent was mapped. I also believe that this function may have had more reliability and potential for success over long evolution periods.

I personally found the disadvantages of this mapping system too complex for what the task required, however. For one, there were more value to sift through and access, which proved frustrating for me as an individual who does best when they have a concrete understanding of the inputs and outputs at any given point in my algorithm. Contrarily, the simplified approach that I ended up going with allowed me to map the inputs to outputs in a way that was more manageable for me. There were less values to worry about and it accomplished the same result, even if it was not so conventional.

Evaluation

Overall, I am pleased with how my algorithm performed given that genetic algorithms and Python were both relatively foreign concepts to me before this assignment. In fact, this assignment tested me possibly more than any other assignment at university. I am aware that my algorithm could be better in some ways, and I would have liked to get a better average fitness score but it was satisfying to see it learn some behaviours over generations.

In future I would be interested in implementing Elitism Selection if I had the time, to see what effect that may have on the outcome. Additionally, I would be interested in seeing the results of other chromosome mapping configurations to see what the most effective method is for this assignment.

References

Karlik, B., & Olgac, A. V. (2011). Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, *1*(4), 111-122.

Wright, S. (1932). *The roles of mutation, inbreeding, crossbreeding, and selection in evolution* (Vol. 1, pp. 356-366). na.

Nei, M., Maruyama, T., & Chakraborty, R. (1975). The bottleneck effect and genetic variability in populations. *Evolution*, *29*(1), 1-10.