COSC 343

Evolve Game-playing Agents with a Genetic Algorithm

Trent Lim – 5884658

Introduction

In nature, each individual in a species is made up of genes that are stored in its chromosomes. These genes are the 'code' behind an animal's outward appearance and inner workings. When individuals reproduce, offspring are created with a genetic makeup comprising a combination of their parents' genes. A key tenet in Charles Darwin's theory of evolution is that of natural selection: individuals who are 'fitter' – who have superior ability in competition and survival – are more likely to be selected for reproduction. These fitter individuals are said to arise from mutation. Mutation can occur in the reproductive process when the child's genes are altered due to mistakes or environmental factors. This mutation leads to genetic diversity and variation in the population, and hence a likelihood of improved, fitter genetics.

Genetic algorithms (GA) are search algorithms that are inspired by Darwin's theory. The main ideas and processes of Darwin's theory are taken and used as parameters in GA:

- Chromosomes containing genes.
- A fitness function.
- Parent selection.
- Crossover.
- Mutation.

These parameters will be used in a GA which attempts to optimize the strategies used by agents in a virtual game. The game puts two populations of creatures against each other, and one population is controlled by an agent function I implemented using a GA. This agent function and the function used to generate new generations of populations is stored in myAgent.py.

Chromosomes and the Agent Function

Each creature's chromosome is stored in a matrix with 75 rows and 5 columns, resulting in 375 genes in total. Each row corresponds to a square in the given 5×5×3 tensor, and this percept can be seen as the creature's input, a way for the creature to gain information about its environment. The percept informs the creature about neighbouring creatures, strawberries, and walls in the surrounding 5×5 area, and the value of each square in the percept can be <0, 0, or >0.

Columns in the chromosome represent the number of possible actions the creature can choose. The first four actions correspond to 4 movement choices: left, up, right, and down; and the last action relates to the action of eating a strawberry. When a creature is initialised, each gene in its chromosome is set to a random float between 0 and 1 with uniform distribution and can be regarded as the 'weight' for that action with regards to its percept. I decided against hard-coding any of the values in the chromosome to allow for variation. This is intended to more closely mimic the processes of evolution seen in nature and to allow the genetic algorithm to serve its purpose of 'natural' selection while exploring possible new avenues for success. As the genetic algorithm creates new populations and chromosomes, the weight values in the creatures' chromosomes will begin to evolve with respect to the given fitness function with the intention to increase its likelihood of employing game-winning strategies.

$g_{0,0}$	$g_{0,1}$	$g_{0,2}$	$g_{0,3}$	$g_{0,4}$
$g_{1,0}$	$g_{1,1}$	$g_{1,2}$	$g_{1,3}$	$g_{1,4}$
		$\vdots$		
$g_{74,0}$	$g_{74,1}$	$g_{74,2}$	$g_{74,3}$	$g_{74,4}$

The agent's chromosome

Each gene is in the form $g_{i,j}$, with:

g = *gene*
i = *row*
j = *column*

This format was chosen for the chromosome to allow for a simple yet effective agent function that maps from the creature's percepts to its actions. The agent function works by taking the 5×5×3 tensor given to the agent and flattening the tensor into a 75-value 1-dimensional vector. It then multiplies each row in the chromosome by the corresponding value in the list of percepts. An array of 5 numbers is subsequently returned, with each index of the vector corresponding to the columns, or actions, in the chromosome, and each number representing the weight of that action. The agent then chooses the action with the largest weight.

The numbers at each index are calculated by summing all the values of the corresponding column in the creature's chromosome, after being multiplied by its percepts. Other methods for the agent function were considered. One alternative was to take weights of each index from the largest value of the column in the chromosome. Another method involved taking the largest absolute value in the column, as these values could be negative. After some testing and training of the algorithm, it was apparent that the current agent function led to more consistent improvement with a slight degree of variation compared to its alternatives, as presented in Figure 1, Figure 2, and Figure 3.

This agent function seems to me like the logical solution for agents to map its action weights based on the information it receives about neighbouring creatures, walls, and strawberries while also allowing for greater genetic diversity and variation.

Fitness Function

The fitness function for each creature proved to be one of the key parameters in the algorithm's success, as it determines which creatures will be selected as parents for children in the next population, and hence the success of the populations over time. The function is intended to score the success of each creature based on information given about the creature at the end of the game in the form of seven attributes:

- a - whether the creature was alive at the end of the game
    - $a = 20$ if the creature lived, $a = 0$ if the creature died
- t - the number of turns that the creature lived to
- s - the size of the creature
- b - how many strawberries the creature ate
- e - how much energy the creature gained from eating enemies
- v - how many different squares the creature visited
- x - how many times the creature bounced

Before the first run of the program, I initially decided on a fitness function that heavily weighted whether the creature lived, and energy gained from eating enemies in the formula since I determined that these two variables were the most significant in the creatures' fitness as they are directly related to the final score of the game, and determine which team wins the game. Thus, I formulated my first fitness function:

$$fitness = a + 3e + \frac{7t}{100} + b$$

However, after extensive testing and tweaking, it became evident that prioritising energy gained from eating enemies and strawberries eaten would result in a higher win rate against the random player, and more consistent improvement over time compared to the initial fitness function (see Figure 4). Heavily weighting the a variable without including v caused the creatures to remain still, which did not prove to be an effective strategy. I also discovered that the number of squares visited, and a lower number of bounces resulted in more optimal evolution. Completely removing the number of turns the creature lived and whether it lived until the end of the game as variables in my function also resulted in greater success. This resulted in my final fitness function:

$$fitness = 10e + 5b + v - \frac{x}{5}$$

Parent Selection

To create new generations of populations, each child must have two parents from its preceding population. How we select our parents, given their fitness values, is vital to ensure a good convergence rate. Fitter parents lead to fitter children. A good parent selection method must maintain a balance between selection pressure and genetic diversity. If too much emphasis is placed on selection pressure, genetic diversity would decrease, and chromosomes in newer generations would become too similar. A healthy dose of genetic diversity is required to explore possible winning strategies through its chromosomes. Conversely, excessive variation would result in fewer to no improvement in fitness as the population evolves.

Tournament selection ensures that individuals with lower fitness have a reasonable likelihood for selection, while fitter individuals are also more likely to be selected. The parent selection function selects a randomly picked subset of size 8 from the old population, and the two top individuals in this subset are selected as parents. I concluded that this would be an optimal selection strategy that balances variation and selection pressure. I decided to use this method in my final algorithm, and it proved to be an effective strategy in selecting parents which resulted in consistent improvement over time without premature convergence.

Crossover

Once two parents are selected, the parents' genes must be mixed or crossed over in order to create the child's chromosome. This is analogous to how sexual reproduction in biology results in the child having a genetic makeup comprising a mixture of its parents' genes. There are various methods in crossing over the genes of two parents, including single-point and k-point crossover, however, I chose a uniform crossover in my genetic algorithm.

In uniform crossover, each gene in the child's chromosome has an equally likely probability of being inherited from either parent. I selected this crossover method instead of a method such as single- and k-point crossover as these crossover methods would result in long strings of genes being passed down multiple generations, decreasing diversity. Uniform crossover leads to higher diversity in a child's chromosome and would allow for better gradual convergence towards optimal genes.

Mutation
Following the creation of a child with its genes crossed over, a mutation method is vital to ensure genetic variation in the population. After all, mutation is the driving force behind biological evolution. As a child is passed into the mutation method, each gene has the same probability of being mutated. When a gene is mutated, its weight value is set to a random float between 0 and 1. An optimal mutation rate was determined through a process of trial and error, and the final algorithm uses a mutation rate of 0.5%. This may seem small, however, when we consider that each creature's chromosome has 375 genes, the rate of mutation seems more reasonable. This mutation rate would mean that roughly two genes in each chromosome would be mutated. Figure 5 shows the evolution of agents when the mutation rate is 1% and portrays how this rate of mutation results in excessive variation with little to no improvement over time.

Results discussion
The graphs below show the performance of my final algorithm after extensive tweaking and trial and error of the algorithm's chromosome, agent function, fitness function, and mutation rate. After 1000 games of its training schedule, the algorithm proves to be an effective way of improving the strategies of its agents over time. Furthermore, this improvement occurs in a gradual but consistent manner, showcasing a good balance between variation, and prioritising successful chromosomes.

However, the graphs only convey the increase in average fitness over time and do not show the population's win rate. After roughly 50 training runs, populations would consistently win against the randomPlayer and following the 1000-game training schedule, the resulting generation could win against the hunterPlayer. As chromosomes evolved, creatures would move around the grid more and eat more strawberries and enemy players. Due to the consistent nature of the population's improvement, I suspect that with a training schedule >1000 games, the population would continue to advance its strategies and win rate.

GRAPHS

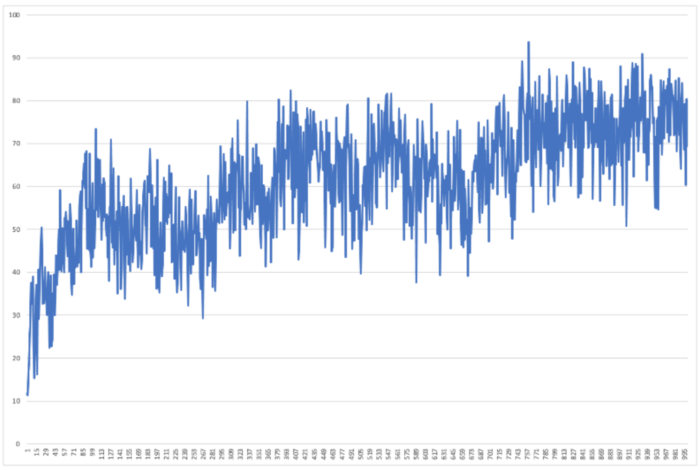

Figure 1: Final Algorithm

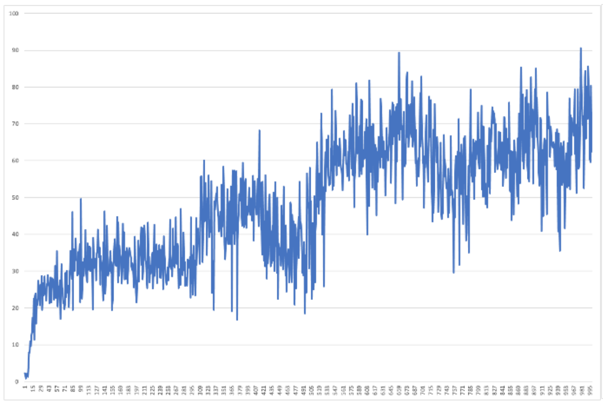

Figure 2: Largest Value Agent Function

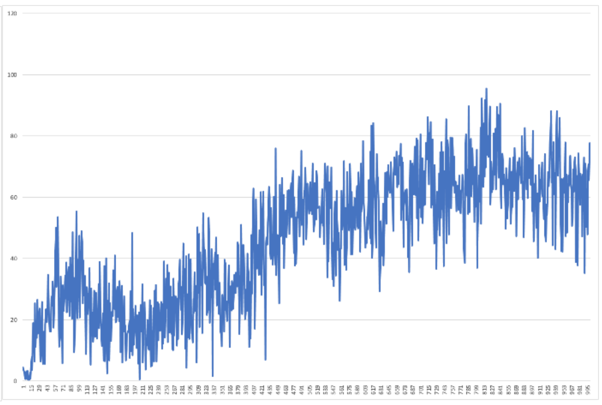

Figure 3: Largest Absolute Value Agent Function

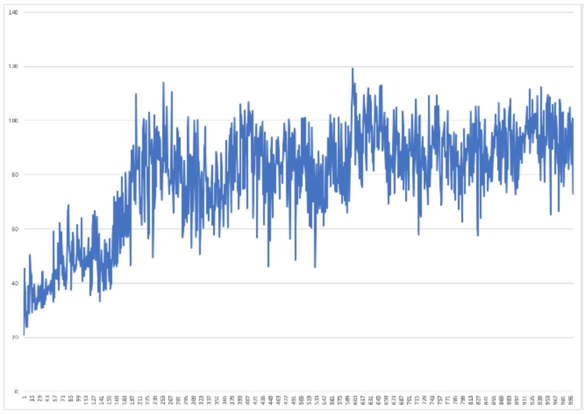

Figure 4: Initial Fitness Function

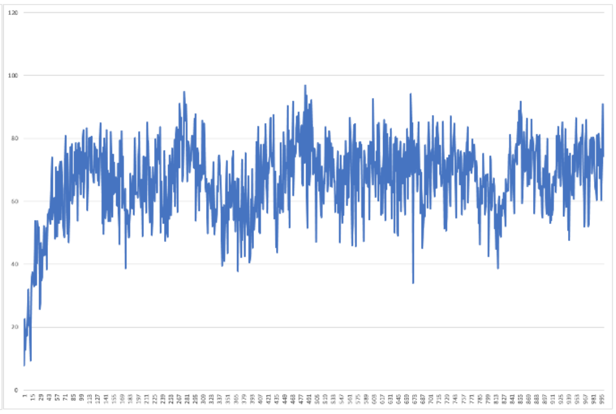

Figure 5: 1% Mutation