COSC343: Assignment Two Report

Evolve a species in a game

Caleb Mazure

ID:6594705

# Introduction

In this assignment we were tasked to implement a genetic algorithm to optimise the fitness of a species of creatures in a 2D grid-based game. This game involved two populations of creatures (red and blue) facing off against each other with the goal of the game being to eat strawberries, grow in size and eat opposing creatures. This report will discuss the choices I have made surrounding the implementation of my creature and agent function. I decided to use a list for my chromosomes, and I will talk about how this affected my overall algorithm. The average fitness of the generations showed a general upwards trend due to populations learning overtime. I will also discuss the genetic algorithm itself and how I used methods of a fitness function, selection, cross-over and mutation to achieve this trend. Finally, I will look at the behaviour of the creatures as they evolve and what patterns they learn across games.

# Methods

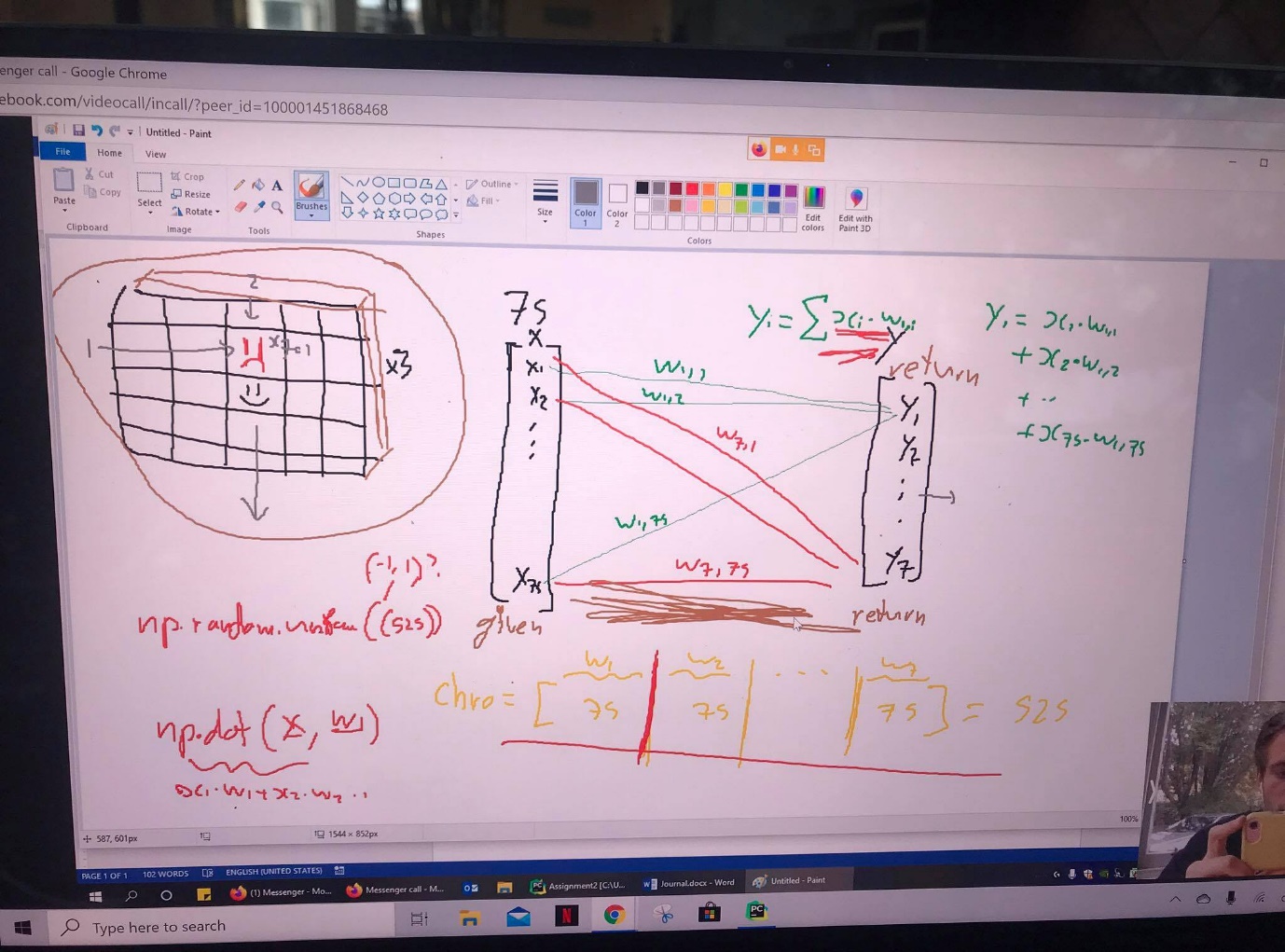
My approach to this assignment was to break it up into three main parts:

1. Agent Function: This function tells the agent how to behave. The behaviours of individuals are influence by the inputs and these then decide on what outputs are produced or what actions are taken.
2. Chromosomes: The chromosomes represent the weights used in the agent function. The shape of the chromosomes also dictates how the genetic algorithm will be carried out specifically, how chromosomes are crossed over and mutated.
3. Genetic Algorithm: This is where I implemented different methods to evolve my agents through each new generation. Here I will discuss the fitness function, selection method, crossover and mutation in my genetic algorithm, also the reasoning for these.

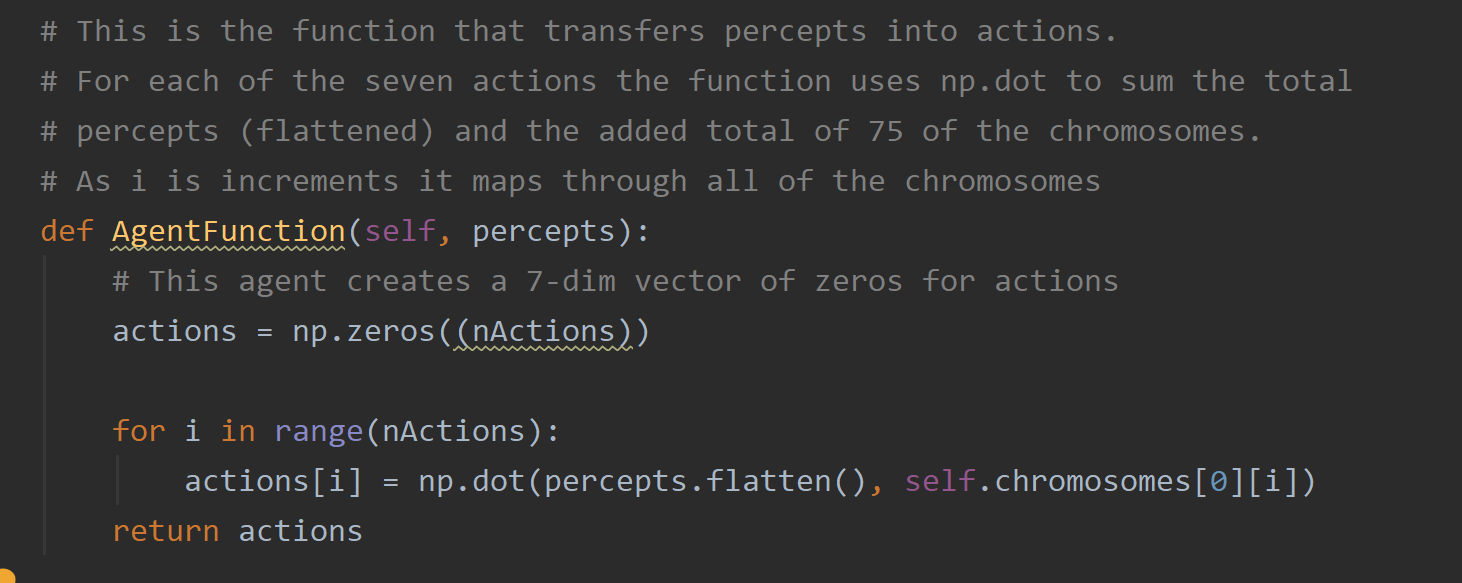
## Agent Function Model

Firstly, the purpose of the agent function is to map the provided percepts into actions of which the creatures carry out. The two parameters of my agent function are *self* and *percepts*. By having these two it allowed me to access both the chromosomes and all the percepts. The percepts are in the form of a 5x5x3 tensor, which corresponds to three maps about the creature’s surroundings. This means that there are 75 possible inputs (5x5x3).

I decided to map my agent function using a single layer perceptron network (SLP). The reason why I choose this over a multi layered perceptron mapping was because I felt that it would result in a lot less work with similar outcome. I could also use the percepts as input and the chromosomes as weights.

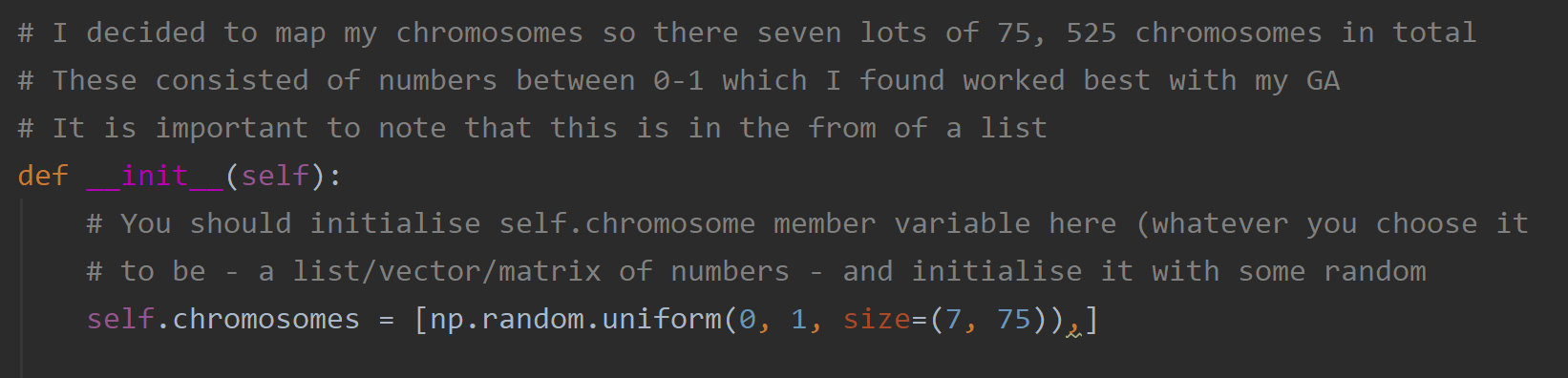


The image displayed here is some of the early workings I did with the lab demonstrator (Yerren) about how to map percepts to actions. It is quite rough, but I think it gives a good visual perspective of what I was trying to achieve. *This was taken before he started to explain it in tutorials.*

The action that a creature took, whether it was to move, eat etc, was decided on the highest value in the list of seven actions. I decided to use the percepts (75 of them) as input and flatten them into a list using NumPy’s flatten function. This would transform the percepts into a list that could be easily multiplied. I used the list of chromosomes that I had created (525 in total) as weights for each of the percepts. As you can see above, this resulted in a function that looked like this *yi = Σ(xi \* wi)*. This is where yi is each of the actions, xi is the percepts and wi is the chromosomes. By mapping the function this way it forces the genetic algorithm I create to involve end to end learning, meaning that there are no hardcoded behaviours.

The screenshot above is the final product of the agent function I used. The agent first initializes the actions as a seven-dimensional vector. I then used a for loop to increment the actions and the chromosomes. Because of the way I mapped my chromosomes, in batches of 75, this meant that I could also increment the chromosomes at the same time as the actions. The NumPy function “*np.dot()*” was used to return the dot product of both the percepts and chromosomes. This function resulted in each of the actions being mapped through the percepts and weighted by the chromosomes. The reason why I implemented the agent function like this was because it was a simple way to achieve end to end learning through using a genetic algorithm.

## Chromosome Choice

The way that I decided to map my chromosomes was in the form of a list. This list contained 7 groups of 75 chromosomes, which totalled to 525 chromosomes. Each of these chromosomes were initialized with a random number between 0-1. The code for this can be seen below.

The chromosome represents the weights used in the agent function that are multiplied by the percepts to produce outputs. I knew that I wanted to use a single layer perceptron network and therefore I realised that I could use the chromosomes as the singular weights. The reason why I specified the size was so that I could easily increment through the weights in both the agent function and crossover/mutation method. I choose this method of defining the chromosome as it allowed me to use them as weights for my agent function and easily select individual chromosomes in the list.

## Genetic Algorithm

The genetic algorithm is what promotes the generations to improve overtime and promote individuals that perform the best. It This is where the core of the testing occurred for my overall program. This involved a lot of time devoted to changing variables like tournament size and mutation methods which would ultimately affect the outcome of the algorithm. I eventually landed on an algorithm which showed effective evolution that would consistently beat the “random player”. The “newGeneration” function is where the genetic algorithm is implemented. Its “old\_population” parameter stores the agents from the previous game, which it uses to create the next generation.

### Fitness Function:

The first of these core parts was the fitness function I developed. After vigorous testing, the solution I landed on was relatively simple. When looking at the basics of the game the main aids of a creature’s survival is strawberries, and this leads to their size. After observing a couple of the first generation’s games it was clear to me that the amount of turns, they survive would impact their evolution. Sometimes this would not be the case as individuals may group together or just hug walls to survive. Based on the knowledge I gathered from observing games I then came up with a suitable fitness function that would allow the creatures to evolve. it looked like this, fitness = (1\*creature.alive + creature.turn + creature.enemy\_eats + creature.strawb\_eats + creature.size). The reason why alive was multiplied by one was so that it either equalled 1 or 0, which only really affected the function when it began to peak as the fitness’s would become more consistent. The amount of turns they survive is important as it stands for a base fitness level (100), and then from there additives like strawberry and enemy eats will lead to fitter individuals.

I also tested the fitness function with the genetic algorithm as a whole and I changed it to see if the algorithm itself was evolving. For example, only including “creature.strawb\_eats” and seeing whether the creatures ate all the strawberries. I also decided to create a method so that I could easily get an individual creatures’ fitness. The main role of this was so that I could compare creatures based on their fitness.

### Selection Method:

A selection method in this genetic algorithm is what is used to select “fit” individuals from a population to use as parents for “children” of the next generation. One method that I was familiar with from Info304 (data science) and lectures was tournament selection. This involved me initializing a list, the tournament, which would be made up of 6 randomly selected individuals from the current population of 34. I then used a for loop to compare each of the 6 individuals based on fitness (using the “get\_fitness” function) and returned the fittest individual from these. The tournament selection had to be run twice to select both parents. It occurred to be that the tournaments could easily select the same individual twice, so to stop this I created a while loop that would continue if the two parents were the same. This would mean that two, different “fit” individuals are selected from the current generation or old generation.

The new population I created from this selection method wasn’t solely based on two individuals. If you look closely at the code (newGeneration) each new individual being added to the new population may have different parents to the last. There is a high chance that these parents will be similar due to the tournament size and the method being run multiple times on a small population. This is how I interpreted tournament selection to work and it worked better than basing a whole generation on two parents. By doing selection this way It made sure that each individual in the new population was relatively fit. [1][2]

The reason as to why I chose tournament selection was because of my previous experience in using it and because of the cost to quality ratio it produces. The other two selection methods I considered were roulette selection and rank selection. Roulette selection would work for this assignment however in my opinion it may work slightly slower that tournament selection. This is because individuals are selected based on chance (related to fitness) meaning that a bad individual could be randomly selected for crossover. Tournament selection almost guarantees that a fit individual will be selected. I briefly researched rank selection however, I felt like I would be better able to code tournament.

### Crossover:

Once parents were selected, I used uniform crossover which allowed the child produced to inherit genes from both parents. This is where there is equal chance of chromosomes being selected from either parent. This means that every chromosome in the new child (all 525) has a 50% chance of either being selected from parent one or parent two. Numbers can vary, due to it being left to chance and will most likely be close to a 50/50 split.

In order to iterate through all 525 chromosomes in the individual and assign them from either parent I used a nested for loop. The structure of my chromosomes looks like this, chromosomes[0][x][y]. The first for loop would define x and increment seven times therefore running through each of the 7 groups of 75. The second for loop would increment y 75 times to then go through each of the 75 chromosomes in the 7. The reason why the first [0] is kept constant is because that is how the chromosome I created was structured. I then had an “if else” statement which assigned individual chromosomes based on which parent is selected by chance (50%).

The reason why I implemented crossover this way was because it seemed like an effective way of iterating through each of the chromosomes based on the structure of them. I considered using single point selection at first, however with the chromosomes size being an odd number there was no way of getting an even split and If I selected a random point then it could be heavily in the favour of one parent. The crossover method also contains code for mutating the chromosomes. [3]

### Mutation:

The whole point in mutation is to introduce diversity into the sample population and prevent convergence. In the same nested for loop for chromosomes I then have a third If statement that is used to decide whether a chromosome is mutated or not. This if statement compares a random number with the predefined mutation rate. If the mutation rate is higher than the random number, then that specific chromosome will be mutated. There are two main points of this mutation which I played around with. The first of which was the mutation rate. At first, I thought a solid mutation rate would be around 0.001 meaning that roughly one chromosome would be mutated per individual produced. In conjunction with this the way I carried out the mutation was to just make the chromosome equal a random number between 0-1 (random reset mutation). This seemed to work fine however the agent would sometimes lose games late on and was evolving quite sporadically. This mutation rate was then increased to 0.01 which in general performed better. I then decided to try adding a random number to the chromosome instead of mutating it entirely. In early populations this would have a similar affect as chromosomes would be altered to become more/less prevalent, but as the population grows I thought adding a number may have less effect on the population. [4]

Therefore, this could be considered as “limited mutation”. I observed that data it was producing, and it seemed as though the chromosomes overtime were getting larger in general. This means that adding would still have an impact and would be mutating the population. When I looked at the results of this mutation, they were much more consistent. In using an added mutation, the agent was less sporadic and rarely lost games while also performing better.

After I had decided on a mutation rate of 0.01, I then had to decide on which mutation method to use based on the results. I decided to be different and use the additive method despite it possibly being considered as limited mutation. This was purely based on consistency and performance: beating the random player more convincingly. I tested the algorithm with a more aggressive mutation rate, 0.1. This led to algorithm performing even better. Despite this, I wanted the algorithm to learn a little bit slower and it was very hard to justify a mutation rate of 0.1 as this would mean that 50 chromosomes are being mutated. In the end, this is my reasoning behind my mutation method (both are shown in code).

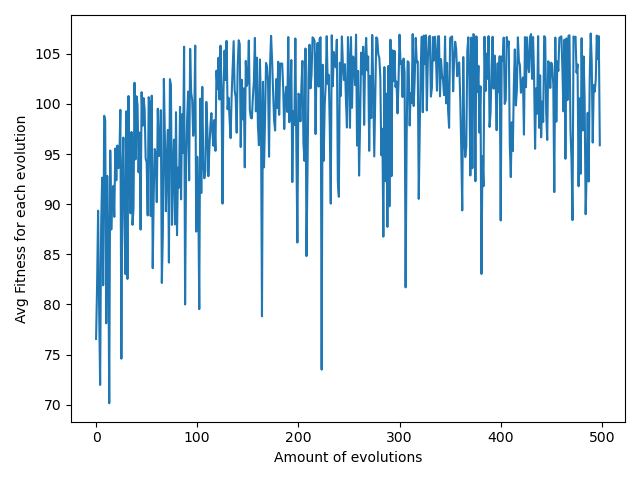
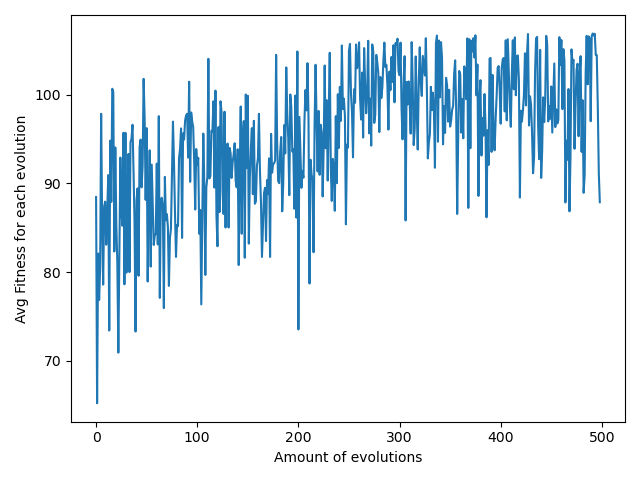
### Review of GA:

The genetic algorithm that I created using these four main functions performed very well and better than I expected. I deliberated a lot on the values I was using which helped lead me to understand what I was doing and get the best result. I decided not to implement other functions like elitism as I felt that by using tournament selection, most of the elites would already be selected and therefore it would not have that much impact.

# Results

## Graphing of results:

I not only produced graphs of the genetic algorithm that I decided on using, but I also decided to graph results when the mutation method was altered. As I mentioned before this worked multiple ways (additive/equals ways) and I wanted to produce graphs to further my understanding or reasoning. The main dataset that I decided to graph and compare was average fitness.

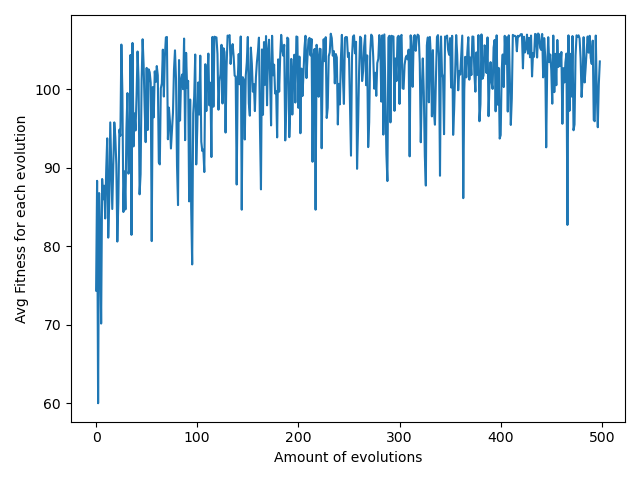
The first two graphs that I will discuss are from the final algorithm I used which had the mutation done by adding a random number. The graph on the left could be considered a “good” example of the code used however most results were quite consistent with the upward trend shown. As you can see from the graph the average fitness of the generations gradually improves until around game 150. The graph on the right uses the same code however it shows how the algorithm can sometimes learn slower through generations while still having an upward trend.

Across generation to generation the average fitness may not necessarily increase, which is in line with most genetic algorithms. However, if we look at the thick parts of the graph which I represented with the black line it shows that the average fitness is increasing. This means that the algorithm is working and that fitter individuals are being selected overtime. Most of the learning occurs within generations 0-100 which can also be seen in the outcomes of the games. The agent usually loses the first 20 games however after this it starts to win, these wins become more frequent as it progresses. From game 100 it is rare for “myAgent” to lose a game. From games 100-250 the games can sometimes be close however after this the agent will start winning decisively and even wiping out the random player population. Occasionally, when the agent is run generations may sometimes learn a different behaviour then what is optimal.

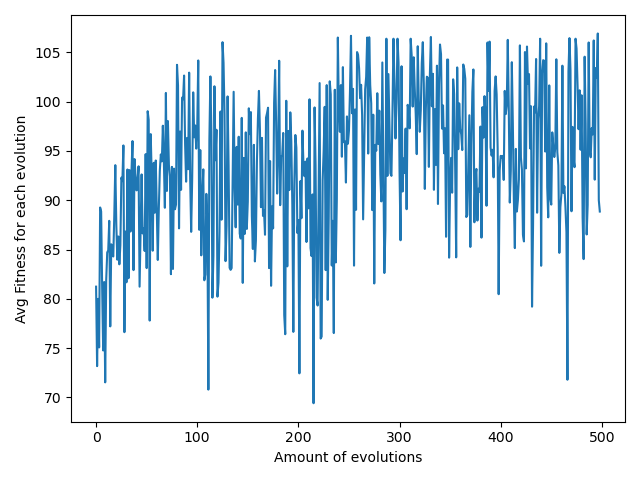
If this happens then the trend will usually have a slight dip and then start to increase again. If you look at around game 70 (first) and 150 (second) you can see this slight dip and then a steady increase again. After running the algorithm several times, I found that it can occasionally lose games at around the 200 mark and then start winning again. This is because of it relearning the optimal behaviour.

The results of this algorithm can vary however it always seems to learn and perform well. What I mean by this is that sometimes after game 100 the agents will start to wipe out the enemies and will consistently hold the random player to under 10 creatures until the end of the game.

In other instances, it will not stop learning until game 150-350 and then start to win decisively. I guess it is based on what individuals are selected in early populations. If the random player did well “randomly” in the first 20 games, then the algorithm may take longer to work effectively. There is one clear trend that I saw in all the simulations and this was that most of the agents learning is done between games 0-150. It can be said that there is a positive correlation between average fitness and the number of generations, which is only further proof that the agent is improving and learning.

As I stated before I played around with the figures of the genetic algorithm and when I increased the mutation rate to 0.1, I obtained interesting results.

That graph on the right shows how when the mutation rate was increased, the learning rate of the algorithm became more aggressive. It seems that all the learning takes place between games 0-100. It does produce very consistent results with the agent being more dominate, however as I explained before I found it hard to justify having such a high mutation rate. It also does not quite represent “end to end” learning as the algorithm evolves very quickly and tends to peak early.

The final graph I would like to briefly mention is one where I used mutation in a more natural way, with the chromosomes **equalling a random number (0-1)** rather than adding. Mutation rate is still 0.01.

The results produced from doing this are in general more inconsistent and sporadic. There is still an upward trend in fitness and the results of the game are similar, but the agent does not start to decisively win until around game 300. This is again why I choose to use the additive method however both work with the algorithm.

## Behaviour of Agents

The behaviour of the creatures themselves was quite interesting to analyse. Initially creatures’ actions were basically random. There would be a couple in the population that would eat strawberries and survive which increased their genes from being passed on. At games 50 creatures would eat strawberries when they were on them and occasionally seek them out. They also have learnt to avoid creatures that are larger than them or just creatures in general. By this time, we can start to see a pattern in which the fitter individuals scan the grid in a certain direction eating straw berries and enemies in their way. By games 100-200 once creatures reach a certain size from eating, they start to do this scanning motion in one direction.

It is almost like the population moves in a wave seeking out strawberries or enemy creatures. This could also be considered them scanning their surroundings in which they have actively learnt to look for things that benefit their fitness (not just surviving). Eventually the whole population will adopt this behaviour and then begin to not only eat all the strawberries but also wipe out the enemy population.

As I have mentioned before this behaviour is not always adopted first and they occasionally learn slowly. When the population learns slower, they do not move around the grid as much and just act on creatures/berries in their general area. Scanning their general area. This will be evident by them not adopting the wave motion by game 250-300. Generally, by generation 400 the population will have adopted this “wave motion” regardless and be winning decisively.

Personally, it was very interesting to see them adopt this method of winning the game and I can understand why it was so effective. By them moving in a set direction and scanning it allowed them to eat effectively and almost “hunt” enemy players. This seemed to be the most frequently occurring behaviour and can be observed in the simulations of the game. Overall, there is a clear difference from game 1 to 500 meaning that the genetic algorithm had caused the population to learn.

# Conclusion

In summary, I was very pleased with the outcome of my genetic algorithm. I implemented methods of selection, crossover and mutation which all contributed to evolving the population of creatures over the 500 generations. I was quite surprised as to how well the algorithm worked against the randomly player in the end. In comparison to the hunter player it did not perform very well but still did show some signs of defensive learning overtime. The reason why I did not extensively test and try to beat the hunter player was because it wiped out the population before it had a chance to evolve. This assignment made me realise how much small changes in a genetic algorithm can have and how they can exponentially grow. If there was anything, I would change about my outcome it would be to implement it so that the creatures learn slightly slower. This could have done by changing the mutation step using equals as I showed above, however I feel like this was inconsistent over testing and in my opinion did not perform as well as my final algorithm. In the end, I learned that Genetic algorithms can be very powerful and capable of coming up with robust solutions.

# References:

[1]<https://cs.stackexchange.com/questions/57186/genetic-algorithms-tournament-selection>, Tournament selection explanation, Date accessed 23/04/2020, published 4 years ago.

[2] <https://www.geeksforgeeks.org/tournament-selection-ga/>, Tournament selected GA, date accessed 23/04/2020.

[3] <https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)>, Crossover, date accessed 25/04/202, Used to give me an idea of different implementations.

[4]<https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm>. Genetic Algorithms mutation, Date accessed 1/05/2020, last updated 2020.

COSC343 lectures and lab material. This was accessed throughout the semester.