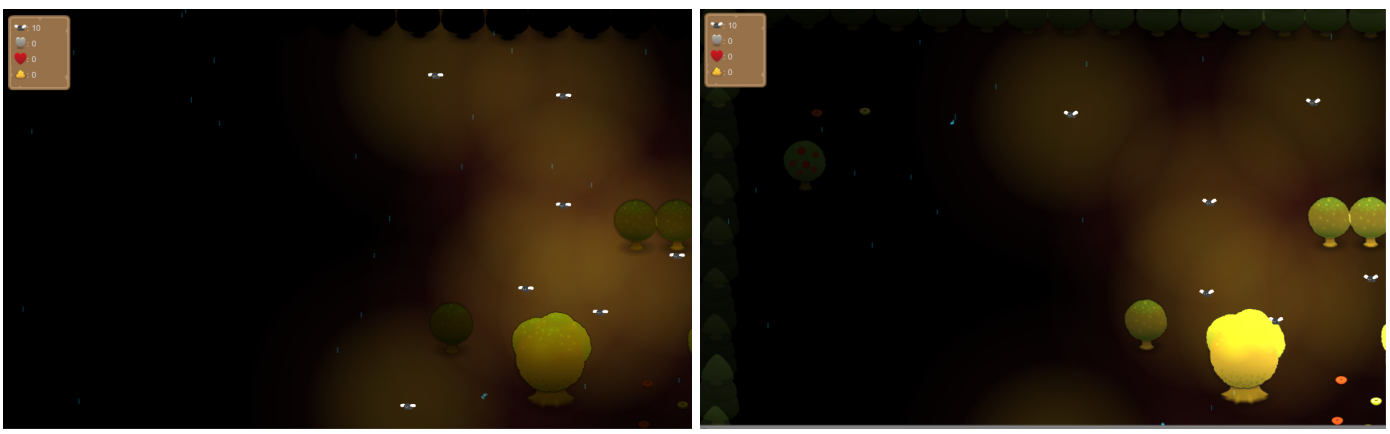
Games and Artificial Intelligence Techniques
Assignment 2

Team 3: Michael Dann and Jayden Ivanovic

October 16, 2014

1 Game Overview

In this section we will briefly talk about the game in which we implemented Neural Networks and Genetic Algorithms. We made a significant number of changes from Assignment 1 to the extent that it is now a completely different game. Therefore, instead of emphasising each alteration, we will explain the new game rules as a whole. In Assignment 2, you control the group of flies and must search the map for resource trees. The resource trees are the apple tree (red) and flower tree (white). The location of these trees is procedurally generated using a Genetic Algorithm at the beginning of the game. At a resource tree, the fly will begin to accumulate points which it must return to the "home tree" in order to increase the players score. For every 100 points scored, a fly will spawn. The player has limited vision from the light each fly emits, however, at random intervals lightning will strike revealing the maps objects as can be seen in Figure 1. During the course of the game, the player must also be cautious of the frog predator which eats their flies. The frog is implemented using Neuroevolution, in particular, Genetic Algorithms and Neural Networks.



(a) The lighting in the scene when there is no lightning. (b) The lighting in the scene when lightning strikes.

Figure 1: The effects of lightning on the scene.

2 Objectives

For this project we had the following core objectives:

- Implement Genetic Algorithms and Neural Networks in a game environment as a learning exercise. Neither of us had applied either algorithm before and we were both interested in learning them.
- Use a Genetic Algorithm to procedurally place trees on the map to add variation for the player and enhance overall gameplay.
- Use a Genetic Algorithm to evolve a Neural Network which controls the steering of a frog agent which acts as the players adversary.

3 Genetic Algorithms

In this section we begin with a brief overview of the standard implementation of a Genetic Algorithm (GA) and include pseudocode for the readers reference. Next, we discuss our implementation and the design choices made so we could easily use our implementation for two different problems. We then look at one of these problems and highlight how a GA is an appropriate choice. This also serves to provide the reader with an example of how a GA can be applied in practice. Finally, we discuss the challenges faced in using the GA to solve this particular problem.

3.1 Overview, Design and Intention

Algorithm 1 is our implementation of the standard GA. Methods such as **SelectParent()**, **CrossOver()**, **Mutate()** and **CalculateFitness()** are abstract and implemented by the subclass. In doing this, it was trivial to use the GA for two different problems. Additionally, it allowed us to experiment with different selection methods, such as proportional roulette, tournament and ranked roulette by simply overriding **SelectParent()**. Similarly, the same can be said for crossover and mutation.

Algorithm 1 A standard Genetic Algorithm

```
children ← InitEmptyPopulation()
numChildren ← 0
for all members in population do
    fitness[member] ← CalculateFitness(member)
end for
while numChildren < populationSize do
    parent1 ← SelectParent()
    parent2 ← SelectParent()
    if randomVal < crossoverRate then
        child1, child2 ← CrossOver(parent1, parent2)
    else
        child1, child2 ← Copy(parent1, parent2)
    end if
    Mutate(child1)        ▷ Where the mutate method handles which genes of the chromosome get mutated
                            according to the mutation rate.
    Mutate(child2)
    children ← children + child1
    children ← children + child2
    numChildren ← numChildren + 2
end while
population ← children
```

We employ the use of GA's in our work and believe they are strongly applicable for the following reasons:

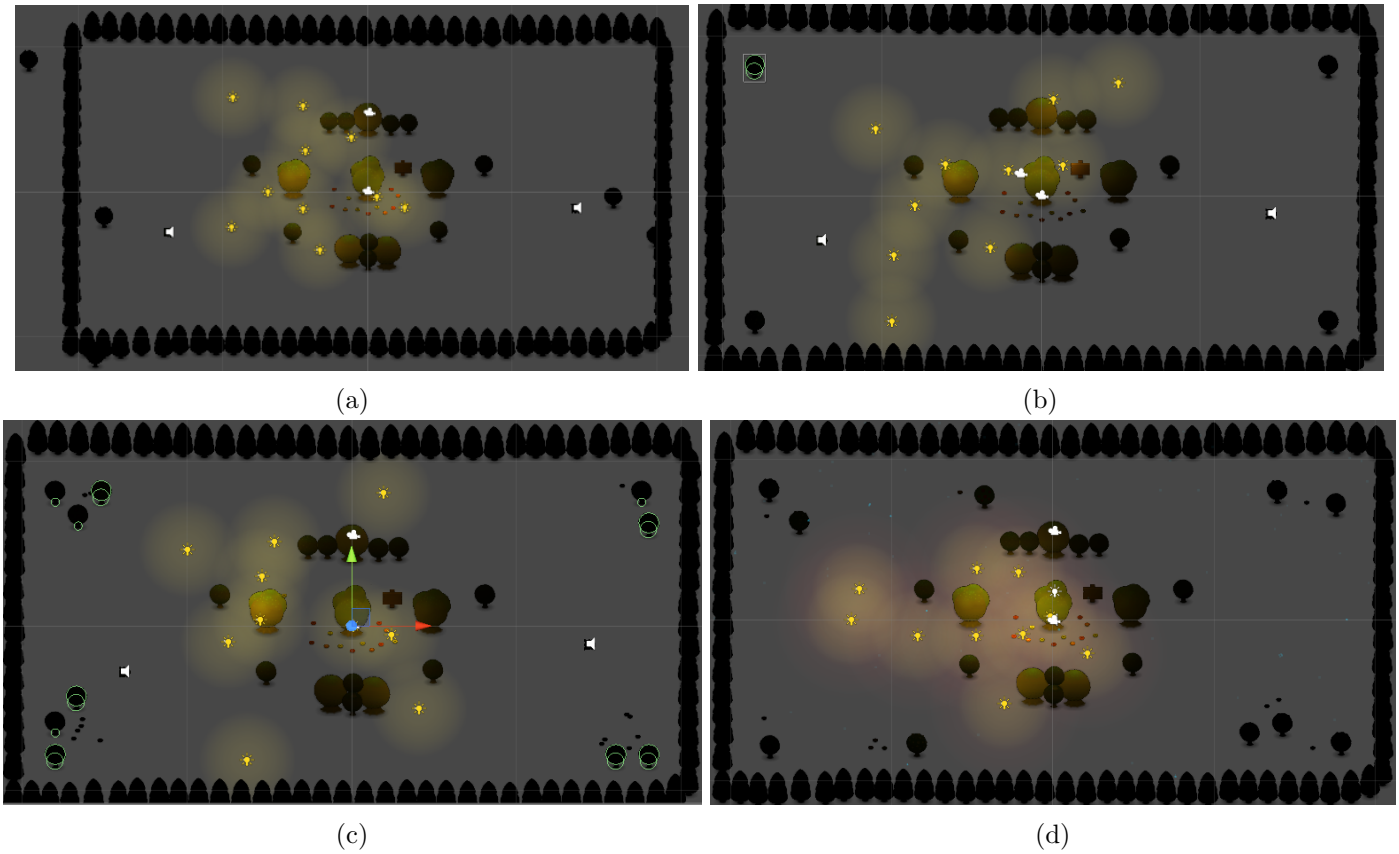
- Many solutions exist in the state space which would be infeasible to manually explore. By defining a fitness function which encapsulates the properties of a desired solution, we can explore far more possibilities.
- We can observe solutions which we may never have previously considered.

- A GA framework is extensible and allows code reuse as the core algorithm can be used for different problems with minor changes.

3.2 Tree Placement

For a player to maximize the amount of resources they collect, a certain subsection of the map may be far more profitable than another. However, in order to know this they must discover it. If the map topology remained static, the player would always head to the most profitable area without much pause for thought. In order to make subsequent games more interesting and enjoyable, we decided to randomize the placement of the resource trees.

The objective of the GA and fitness function was to optimize the placement of dynamic objects around the borders of the map at the start of the game. We chose tournament selection as it can reach a local optimal solution in a smaller number of iterations and for our problem, we do not require the exact solution, an approximate one will do (the time cost of ranked roulette isn't worth it).



(a) (b)

(c) (d)

Figure 2: The steps to implementing an appropriate fitness function. In 2a the trees are placed outside of the game border. If we left the GA to run for 100,000 iterations, the trees were much further away and to see them the scene must be zoomed out. In 2b we remedied this problem , yet created a new one. Here the trees remained in the four corners of the map as the GA found an optimal solution which maximised this fitness function. To prevent this behaviour, we needed to take the distance to other gameobjects into consideration which adds some extra overhead. Figure 2c shows the inclusion of flower positions. As one can see there is quite a lot which are clustered. Removing the number of flowers generated did not solve this issue. Instead, the fitness function needed minor alterations in regards to the required distance a gameobject should be from another. Figure 2d shows the final fitness function and GA parameter set. Here the flowers and trees are spread out far more. Additionally, this iteration of the GA gives a better amount of randomization on subsequent runs.

3.3 Challenges

In this section we highlight the challenges we faced for the specific problem of randomizing the placement of objects in the game world. We leave discussion of the challenges related to evolving weights for a neural net to train a frog controller for section 4.

3.3.1 Tournament Selection

A minor issue we faced with tournament selection was that it was initially selecting too many chromsomes with a low fitness value to breed children for subsequent populations. The inital tournament size was 2 and when randomly selecting chromosomes to take part in such a small tournament, the majority of the population was being filled with poor solutions. We changed the tournament size to be half of the total population and witnessed immense improvements in the solutions that were being generated, as the chromsomes with a low fitness value were unlikely to be present or have offspring in the next population.

3.3.2 Fitness Function and Parameters

Briefly, we note the common issue that the fitness function and parameters of a GA often requires a substantial amount of experimentation and tweaking. Situations which you previously did not take into consideration may arise in generated solutions and you thus need to account for them. Additionally, one must balance the positive and negative costs. For example, in Figure 2a our original fitness function did take into consideration that chromosomes with tree positions outside the boundaries should have a lower fitness. However, as the fitness was calculated as the cumulative distance of the tree to other trees, as the trees got further and further away, the fitness value was always higher than the penalty for being outside the boundary. We discuss how we solved such issues in Section 3.3.3, here we simply point out the common issues with GA's in general and there reliance on an appropriate fitness function and parameters.

3.3.3 Generating Positions at Runtime

One of the issues we faced was generating the positions for the flowers and trees at runtime. Here we did not have the luxury of running the algorithm offline and then plugging in the result later. Therefore, we had to get the right balance of parameters (number of trees/flowers, epochs, crossover rate, mutation rate and mutation amount) to generate the solution in an acceptable time frame. We found that running for less epochs wasn't too much of an issue as it gives us a better random selection of map layouts. However, in some rare circumstances, trees/flowers may overlap or be placed in the middle of the lake with a poor initial population.

The problem was further exaggerated by the fact that there was objects which were already statically placed on the map. In this instance, we could solve this simply by placing an invisible boundary around the center game objects, which if a resource tree was placed inside, the fitness would be heavily worsened. Additionally, in an attempt to speed up generating an acceptable solution, we check for the following:

- Overlapping trees/flowers
- Trees/Flowers outside boundaries of map
- Trees/Flowers in the lake

and if they exist, assign a fitness of 0 as we never want them to occur. Therefore, during selection, there is a very low probability that they will see future generations.

3.4 Outcome

Figure 4 shows several different solutions generated by the final fitness functions and parameter set (Table 1). It's interesting to note the variations in each. Lastly, Figure 3 shows how the largest individual fitness value changes over the course of 1500 epochs.

Parameter	Setting
Mutation Rate	0.001
Crossover Rate	0.7
Vector Mutate (i.e. vec.x or vec.y $\pm$ vector mutate)	1.5
Epochs	1500
Number of Trees	8
Number of Flowers	10

Table 1: Final parameter settings for object placement.

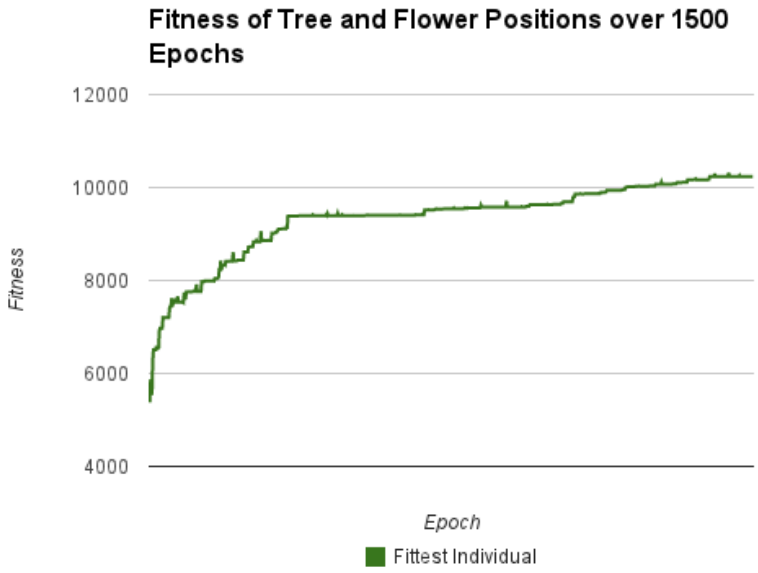
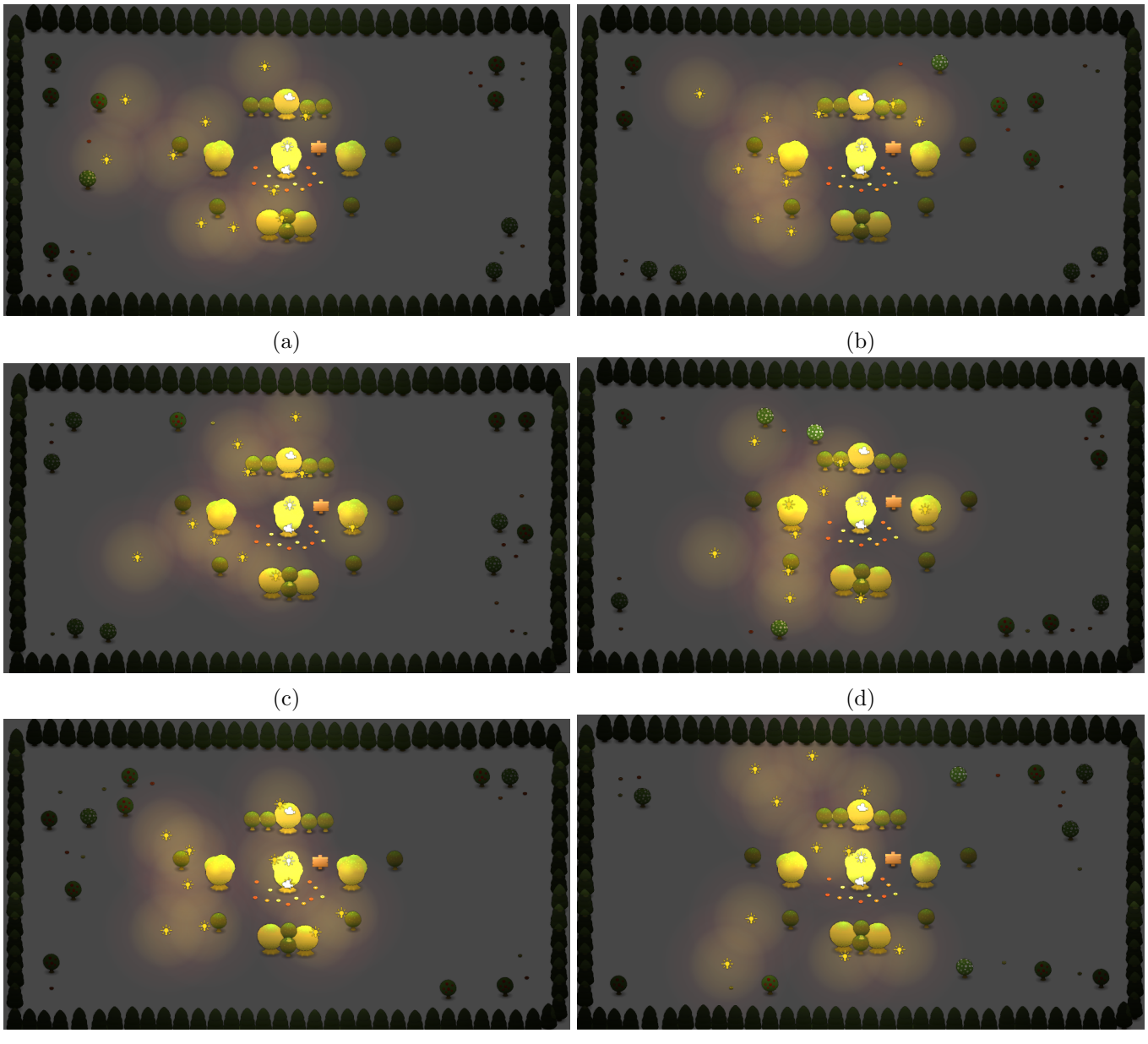


Figure 3: Fitness of the fittest individual over 1500 Epochs.



(a) (b)

(c) (d)

(e) (f)

Figure 4: A subset of solutions generated by the final fitness function and parameter set.

4 Neuroevolution

In this section we discuss *neuroevolution*, a subtype of genetic algorithm that is specifically designed to train neural networks. After a brief overview of some of the key concepts, we describe how we used the technique in our game to train the behaviour of the enemy frog.

4.1 Overview

The key idea behind neuroevolution is to treat neural networks as the chromosomes in a genetic algorithm. The weights of the connections between neurons (and potentially the connections themselves) are treated as genes. Neuroevolution is particularly suited to problems where labelled training data does not exist but where it *is* possible to compare the quality of solutions via a fitness function. The lack of labelled training data makes it impossible to train a network using backpropagation, but evolutionary methods still work because they do not care what the networks calculate; they only care about the effectiveness of each solution.

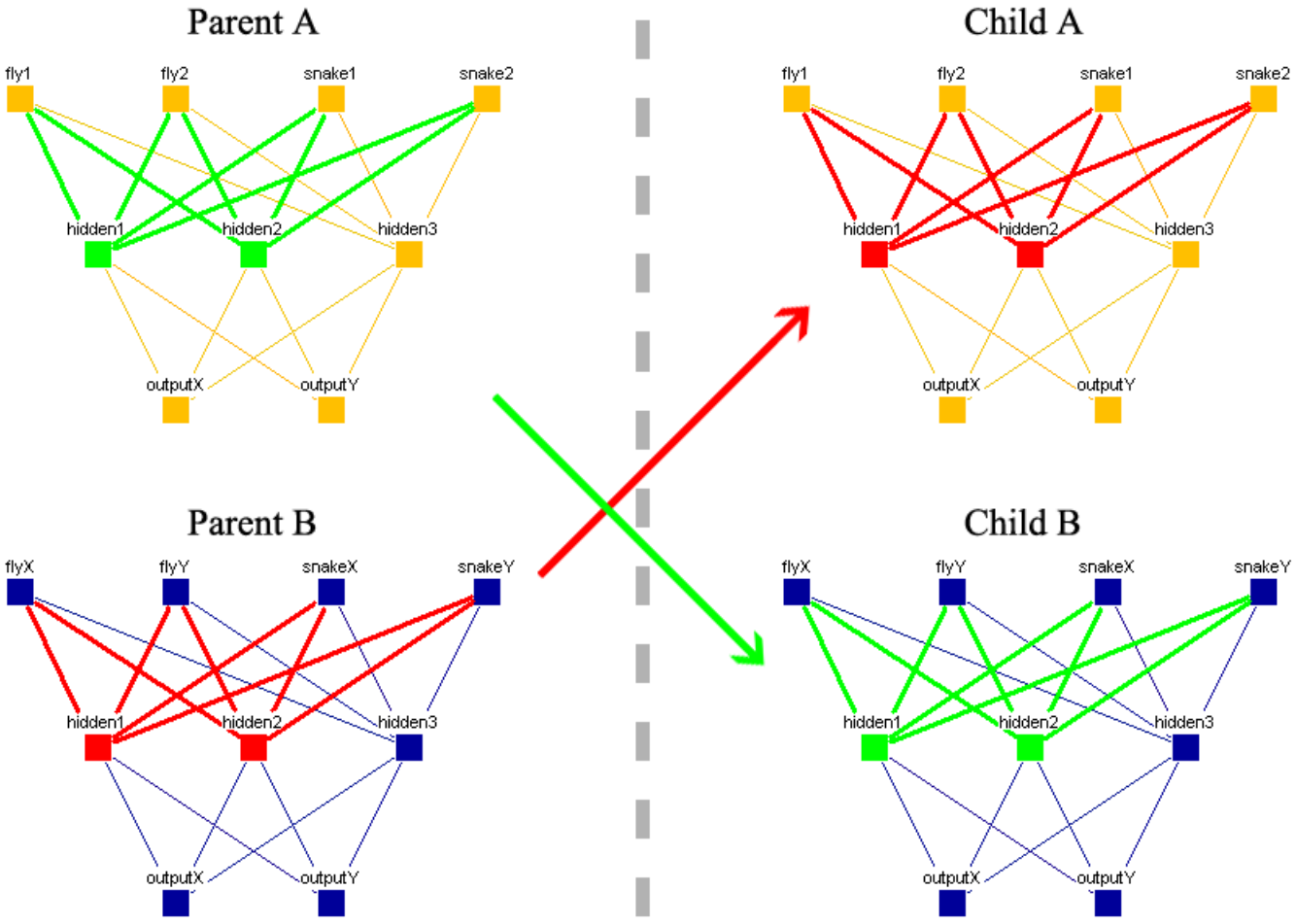


Figure 5: A crossover operation applied to neural networks. Here, there are two genes crossed over.

In neuroevolution, the crossover operation generally yields stronger children when input weights to the same neuron are treated as a single gene. This is because such weights may be finely tuned in relation to each other, so cutting halfway between them may ruin the neuron's precise configuration. Figure 5 illustrates the preferred type of crossover. While there are eight connection weights involved, this corresponds to only *two* genes being swapped. Mutation, on the other hand, still perturbs weights individually.

4.2 Neuroevolution in Our Game

Automating the behaviour of the enemy frog in our game struck us as a potentially interesting application for neuroevolution. Our goal was to train a neural network that, given a set of real-time game data (such as the relative positions of the nearest flies, snakes, obstacles and lakes), would output an appropriate (x, y) steering for the frog. Of course, we could have hardcoded a list of behavioural options for the frog and had the neural net choose between plans, or we could have taught the frog how to behave based on data recorded from human play, but we wanted the frog to learn by itself from scratch!

4.2.1 Network Architecture

We designed the input layer of the neural networks to be dynamic so that the number of flies, snakes, obstacles and lakes fed in could be changed easily from the inspector in Unity. This setup allowed us to experiment with training simple tasks first (e.g. catching stationary flies with no predators or obstacles) then increase the difficulty as we discovered what the frogs were capable/incapable of learning. In the final configuration we used to train the in-game frog, the network was supplied with the nearest two flies, snakes and obstacles, and the nearest lake. Each object requires two inputs (x and y), so this gave a total of 14 inputs. We also fed in the frog's current water level so that the frog could theoretically learn to approach snakes when it had enough water to fire a bubble and return to the lake when its water level got low. (More on this in Section 4.2.7).

The input vectors were calculated relative to the frog; i.e. the vector from the frog to each object was used, not the object's world co-ordinates. In the literature we saw that it is common to also normalise angles to the agent's co-ordinate system, but we experimented with this approach and found that it was not of much help. The problem with normalising angles is that the frog's rotation does not particularly matter in our game. The frog's turn speed is so fast that it does not matter if, for example, the frog is facing a snake or has its back turned (see Figure 6). If angles are normalised then these scenarios are presented as different inputs, even though the frog's behaviour should probably be the same either way.

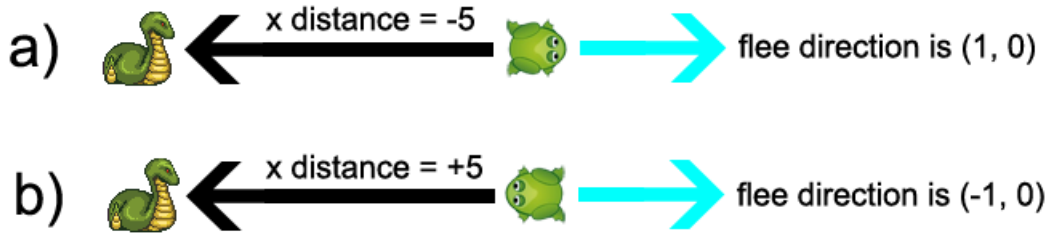


Figure 6: In the frog's co-ordinate system, these scenarios are different and require different responses. The frog has to learn different behaviours for every orientation it is in. World co-ordinates are no better, since the frog must learn different behaviours for every rotation of the vector to the snake.

Our solution to this problem was somewhat crude, but it makes full use of the rotational symmetry of the game. The inputs are fed into the network *multiple* times, with a different rotational transform applied to the input vectors at each iteration (from 0° to 360°). The reverse transformation is then applied to the output, and the average is taken across all iterations. This makes the frog's behaviour completely invariant to rotations.[1] We found that this approach drastically increased the frogs' learning rate.

A final note regarding input transformations is that it does not really make sense to feed in the differential from the frog to each game object directly. This would mean that the farther away a game object became, the larger an input signal it would send. Even worse, a snake that was just about to bite the frog would send only a tiny signal! For this reason, we rescaled the magnitude for each input vector according to Equation 1:

$$Input\ magnitude = e^{-k\|V\|} \tag{1}$$

where V is the vector from the frog to the object and k is a dampening constant (equal to 10 in our implementation for all object types).

For the hidden neurons, we chose to use a single layer for simplicity. Generally speaking, the more complex the architecture, the longer the network takes to train. The textbook[2] example we followed only had four inputs, so we were already worried that our architecture was too complicated! Likewise, the *NEAT* algorithm that we saw in class was very interesting, but we did not think we had enough time to experiment with evolving the network architecture. The number of neurons in our hidden layer was configurable, but we used 7 in our final training run.

Importantly, the squashing function had to be an odd function (in the mathematical sense) to preserve the symmetry between the positive and negative axes of the game world. We chose to use a logistic curve, scaled to the range (-1, 1). Similarly, it would not make sense to use a bias (i.e. to shift the logistic curve left or right) since that would also break the spatial symmetry. Instead, we evolved the co-efficient of the exponent

[1] Technically this is only approximate, but as long as the increments are fine enough then the frog's behaviour will be very close to rotation-invariant. There is probably a clever mathematical way of ensuring that a neural network is invariant to rotations, but we are not aware of such a method.

[2] *AI Techniques For Game Programming*

in the logistic function (i.e. we evolved k in $1/(1+e^{-kt})$) so that the frogs could still learn an appropriate compression for the input signals.

As we already mentioned earlier, the output of our neural net was the (x, y) steering of the frog. The only tweak we had to make was to multiply the output by a constant so that the speed of the frog was sensible.

4.2.2 Fitness function

Our fitness function went through several iterations, but the final one we used was:

$$\textit{fitness} = \textit{(flies caught)} - 7.5 * \textit{(damage taken)} - 0.5 * \textit{Min(0, 5 - (water camping score))} \quad (2)$$

where the "water camping score" increments by 1 for every second a frog spends in a lake when it already has full water. The *Min* function serves as a grace period so that the frogs can accumulate a water camping score of 5 before they actually start to be penalised. For the subsequent parent selection, we used ranked roulette.

4.2.3 Challenges

In this section we discuss some of the challenges we faced while training the frogs and the steps that we took to overcome them.

4.2.4 Smart, Yet Undesirable Behaviour

A major problem we encountered was that the frogs often evolved towards a high fitness solution, but the corresponding behaviour was not what we wanted or expected. Some examples are provided below:

Problem	**Solution**
If the training pens contained a large number of flies that were configured to wander around the pens, the frogs learned to avoid the snakes, but not to target the flies. This is because the frogs were likely to catch many flies by pure chance. The frogs that were slightly biased to target the flies tended to get hit slightly more often too, which cancelled out their advantage. They may have learned to target the flies eventually but it was not apparent in our experiments. If there were too few flies then funnily enough the same thing happened, since there was not enough positive reinforcement for the frogs when they caught flies.	We made the flies stationary during training so that the frogs that deliberately targeted them got more of an advantage. We used trial-and-error to finely tune the number of flies in each training pen. We also tuned the reward size for catching a fly relative to the punishment for being hit by a snake. In retrospect, it may have been useful to decrease this ratio as training progressed.
Since the punishment for being hit by a snake was much larger than the reward for catching a fly, the frogs naturally learned to protect themselves first and foremost. Unfortunately, the most effective defense was to run into a lake and stay there! Once the entire population learned this behaviour, training stagnated because the frogs never left the lake and had little opportunity to learn about the reward for catching flies.	We introduced a "water camping" penalty to the fitness function, as mentioned above in Equation 2. Admittedly this was "cheating" – ideally the frogs would have learned to leave the lake by themselves – but the cheat was very effective!
The training runs were originally fairly short (20 seconds long), but this lead to overly-aggressive behaviour. The frogs that focused on catching flies and completely disregarded the snakes tended to rank first every time, because the snakes were configured to wander, and chances were that several frogs would receive favourable runs where the snakes wandered away from the largest fly clusters.	We increased the time limit to 2 minutes per training run in order to expose over-aggression. We also configured the fly distribution to be the same across pens. The distribution was still randomised at the start of each epoch, but stored in a static variable shared between pens.

4.2.5 Training Time

As mentioned in the last problem above, the training runs needed to be reasonably long in order to expose poor behaviours. However, we found that it generally took around 1,500 epochs for the frogs' learning to become saturated (see later, Figure 8). A typical population size in these type of GA experiments is 30 – 50, which would have meant an epoch time of over an hour had we not found some efficiency gains! Our first measure

was to increase the Unity time scale. We found that a factor of 4 worked without making Unity unstable. Our second measure was to duplicate the training pens (see Figure 7). With eight indentical pens, a population size of 40 only required 5 batches of frogs per epoch. We could have created even more pens, but Unity was taking a long time to launch the scene with so many resources loaded at once.

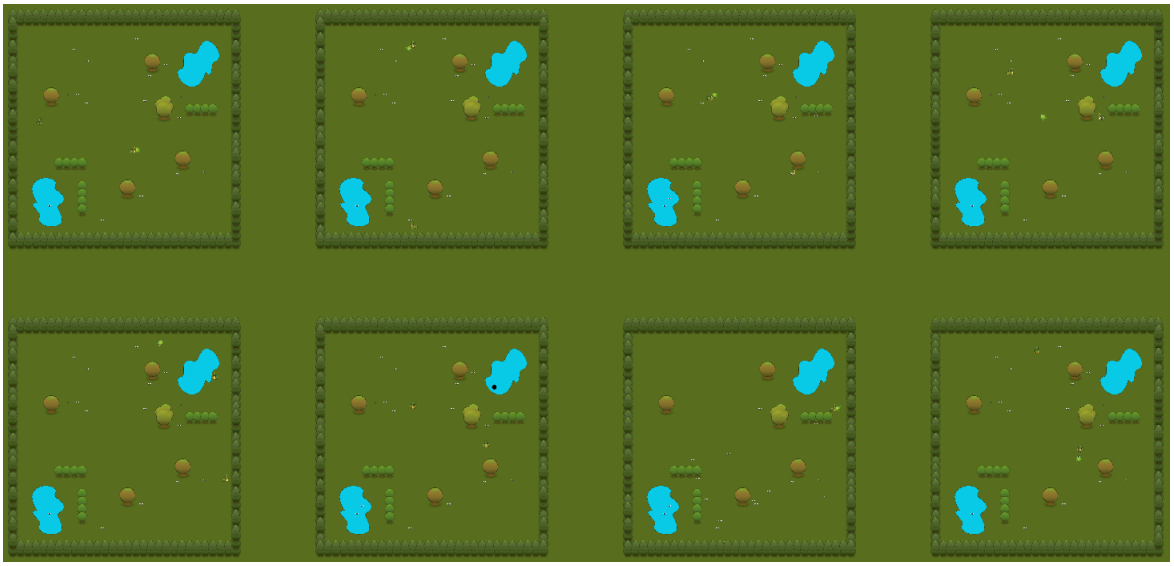


Figure 7: The training pens we set up to train several frogs simulataneously.

4.2.6 Obstacle Avoidance

In the original game, the frog was controlled by mouse clicks and avoided obstacles via a combination of A* and a few tricks to avoid corner collisions. This time the frog was controlled by a neural network, which was essentially just another type of steering behaviour. This created a problem, because A* and steering behaviours do not really gel. A* requires a goal node, but steering behaviours only provide a direction of travel, not a distance. Our solution was to modify the obstacle avoidance technique that the flies used in our first assignment so that the frogs could use it as well. This meant that the frog was no longer capable of navigating mazes, so we had to ensure that the maps we used for training and the actual game did not include any "bowl" shapes that would trap the frog.

4.2.7 Shooting Bubbles

We wanted the frog to learn to shoot bubbles after filling up on water in the lakes, but we decided that having the frog learn this behaviour from scratch would be too difficult, especially given our time limits. Opportunities to fire on snakes are limited, and the overall plan of *obtain water* → *approach snake* → *fire at correct angle* is quite a complex one, particularly since the frogs must learn to avoid the snakes in general. Our solution was to script the frog to fire directly at snakes within a certain range when the water level is sufficient. We added an additional gene for the frogs' shot range, so the frogs learned *when* to shoot, but not *how* to shoot.

Since bubbled snakes present far less danger to the frog, we needed some way of encoding this into the network input. We considering adding new input neurons for bubbled snakes, but this seemed wasteful given that the inputs would be sending a zero signal the majority of the time, and inaccurate in a sense because the frog would be unaware of how much bubble time the snakes had left (without adding a further input). Our solution was to still represent the bubbled snakes as ordinary snakes, but to increase their distance from the frog (in the input encoding) based on how much bubble time they had left. This way, it seems to the frog as if the bubbles push the snakes backwards, then the snakes run back to their bubbled position as the timer runs out.

4.3 Outcome

As we alluded to in the *Challenges* section, we found that the frogs generally learned tasks in a particular order. Since the fitness lost from being hit by a snake far outweighed the benefit from catching a fly, the frogs always learned to protect themselves first. Before we added lakes to the training pens, this meant that the frogs learned to flee from snakes in the opposite direction. After we added lakes, the frogs learned to dive into

a lake at the start of every training run. Gradually however, the frogs that had a tendency to catch more flies (and to stay out of the lake, after we added the "water camping" penalty) became more and more dominant. Eventually the frogs' behaviour became more daring when flies were positioned close to snakes, and their flee steering became more elaborate and "slippery".

Given that the frogs learned completely from scratch (besides some rather involved cheating for the water bubbles), we were very happy with their overall performance. They clearly became stronger throughout training from a qualitative standpoint, but we were also able to confirm that the genetic algorithm worked in a scientific sense by plotting the average population fitness versus epoch from our final training run:

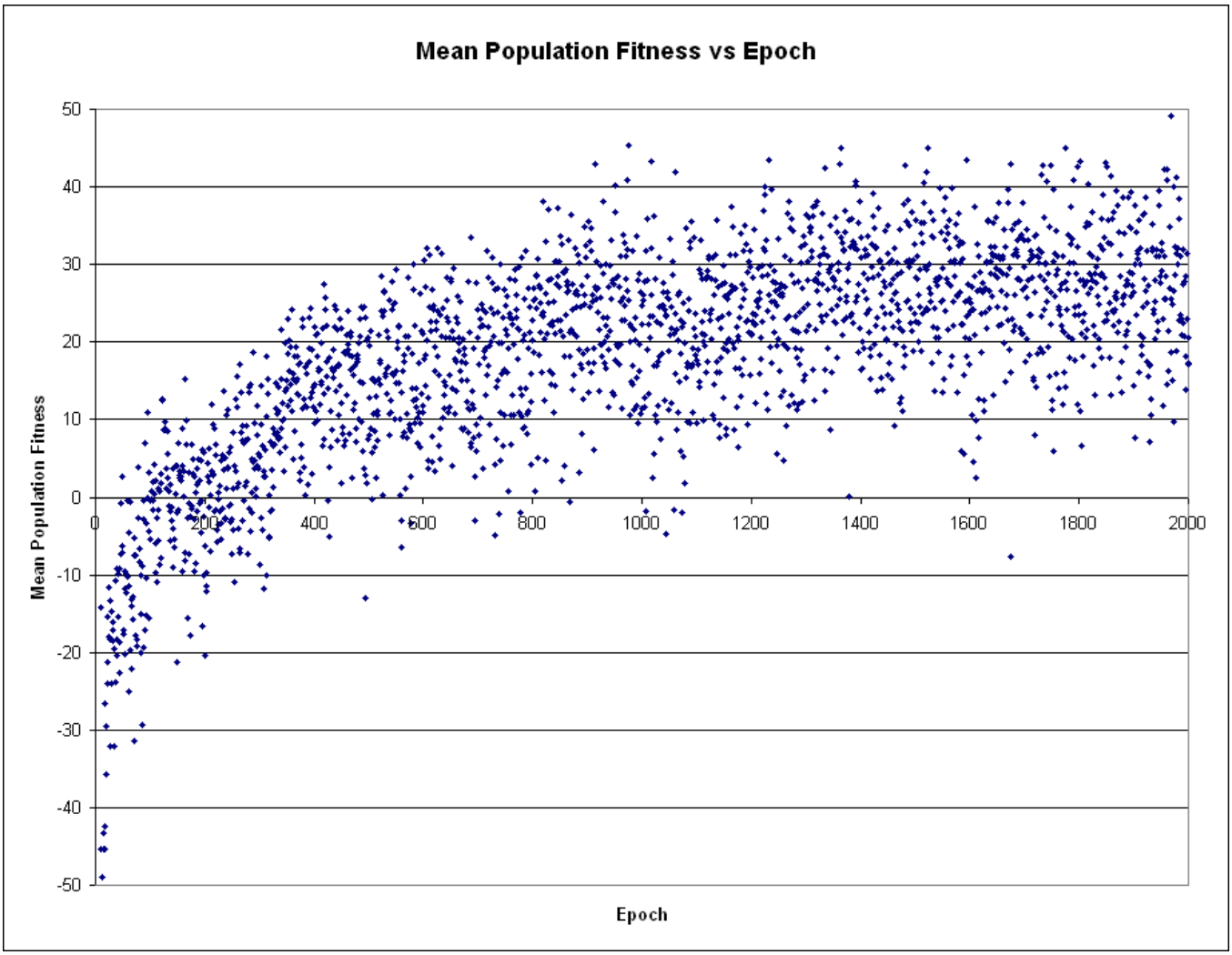


Figure 8: The total population fitness of the frogs versus the epoch count over 2000 epochs. The training time per epoch was 2 minutes.

There is a lot of variance in Figure 8 due to the fly spawn locations changing every epoch, but there is a noticeable trend upwards in performance until around epoch 1,500. At that point the population fitness reaches an average of around 25, which corresponds to each frog catching 25 flies without being hurt, and an additional 7.5 flies for each unit of damage taken.