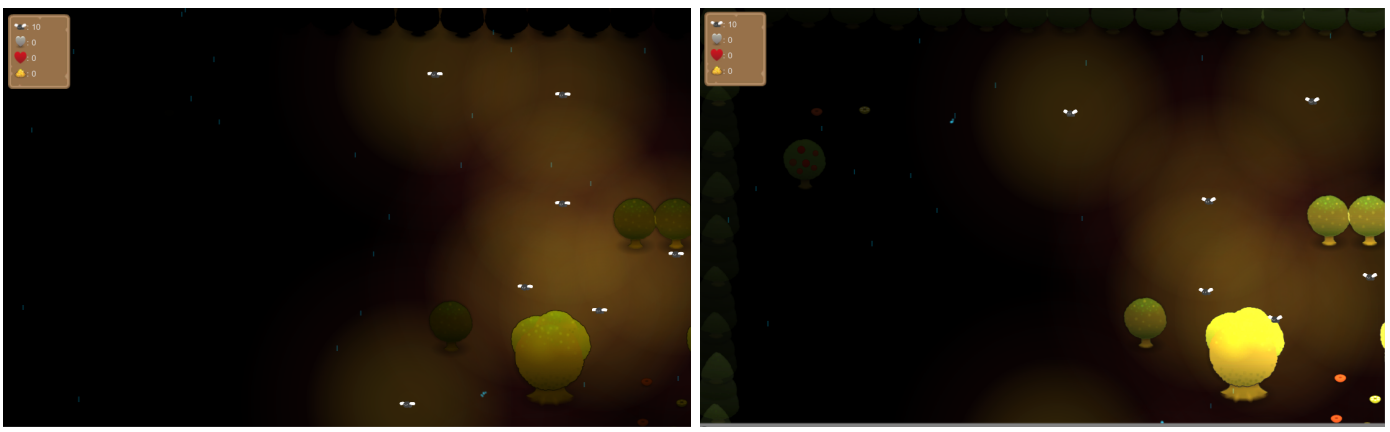
Games and Artificial Intelligence Techniques
Assignment 2

Team 3: Michael Dann and Jayden Ivanovic

October 15, 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. 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 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.

2.1 Overview, Design and Intention

Algorithm 1 is our implementation of the standard GA. For a background please see [CITE]. 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 [CITE PAPER] 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
```

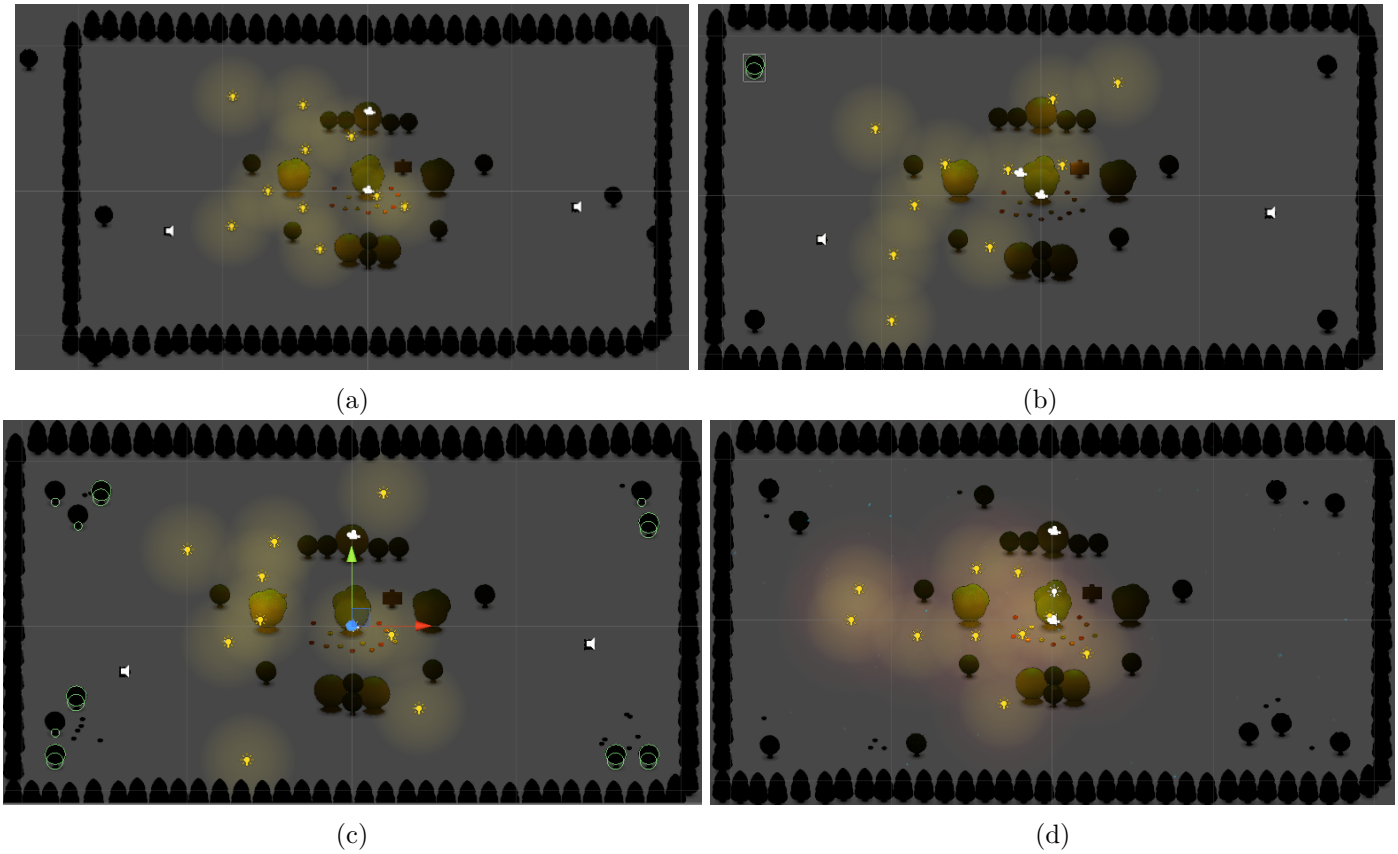
We employ the use of GA's in our work and believe they are strongly applicable for the following reasons:

- 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.

2.2 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 to be clarified. 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.

2.3 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 **??**.

2.3.1 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 2.3.2, here we simply point out the common issues with GA's in general and there reliance on an appropriate fitness function and parameters.

2.3.2 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.

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

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.

2.4 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 fitness of the population increases over time.

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

Table 1: Final parameter settings for object placement.

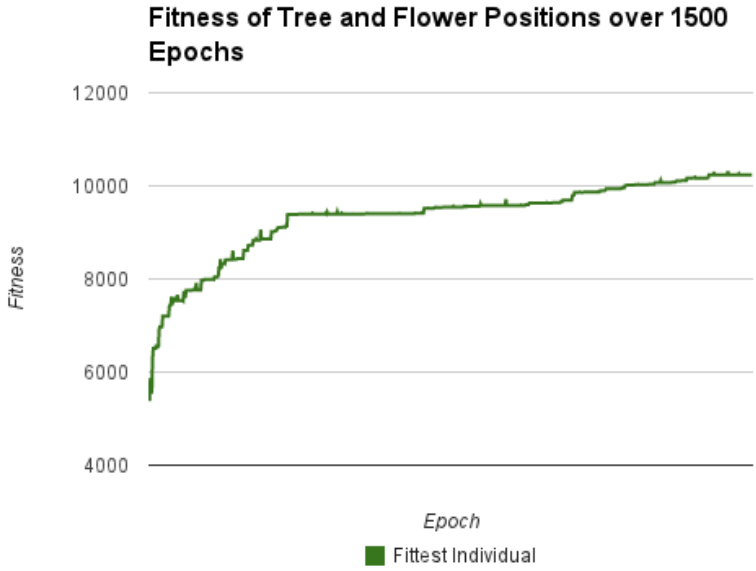
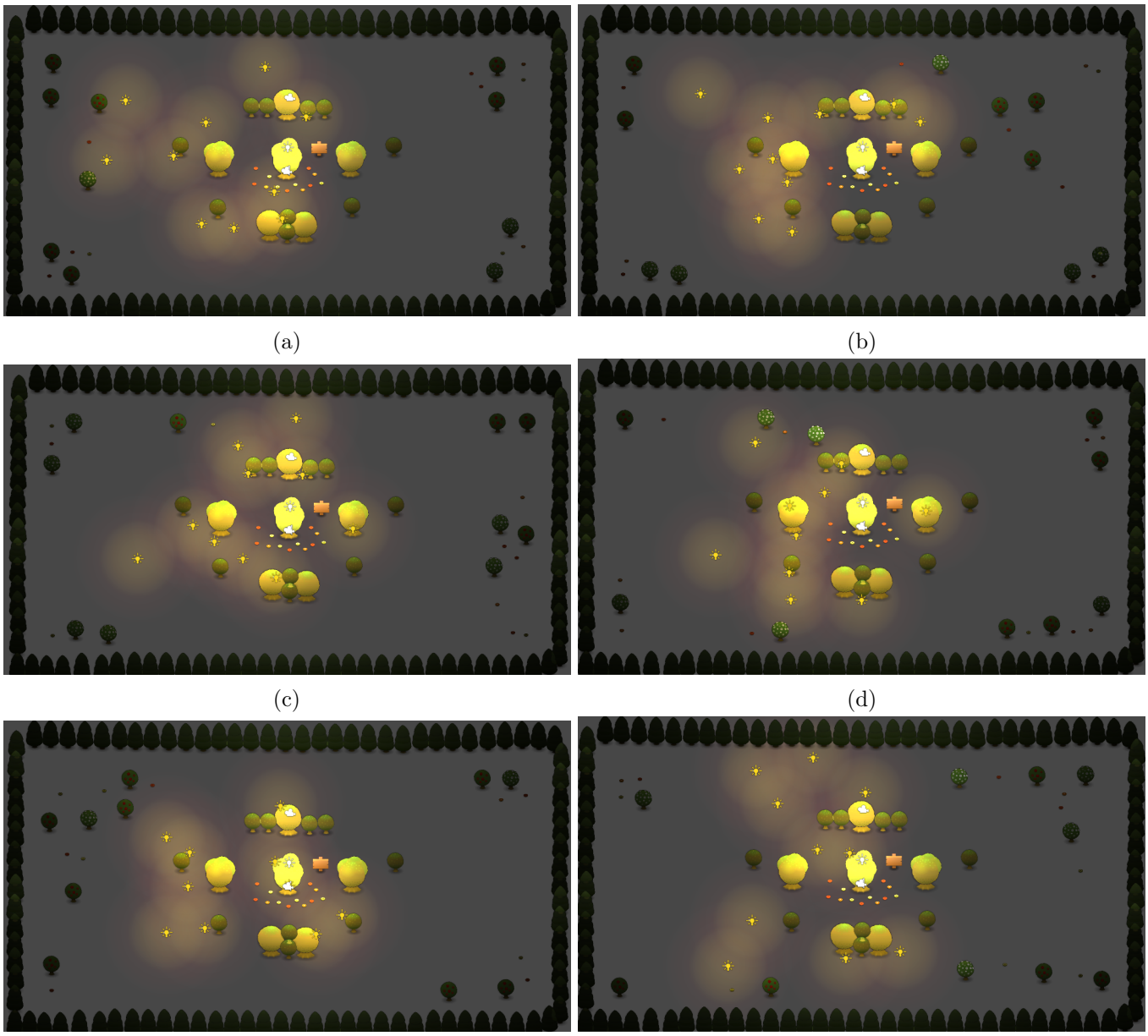


Figure 3: Fitness of the population over 1500 Epochs.



(a) (b) (c) (d) (e) (f)

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

3 Neuroevolution

In this section we discuss *neuroevolution*, which is a application of the genetic algorithm discussed in Section ? is used to train a neural network.

3.1 Overview

The key idea behind neuroevolution is to use neural nets as the chromosomes in a genetic algorithm. The weights of the connections between neurons (and potentially the connections themselves) play the part of the genes. Neuroevolution is particularly suited to problems where labelled input/output pairs do not exist, but it is still relatively easy to assess the performance of a solution at the end of a training run via a fitness function. The lack of labelled data at each time step makes it impossible to train the network using backpropagation, but the genetic algorithm approach still works because the fitness of each neural net only needs to be calculated at the end of an epoch. Implementing the underlying neural networks is straightforward under neuroevolution, since the approach only requires feed forward networks.

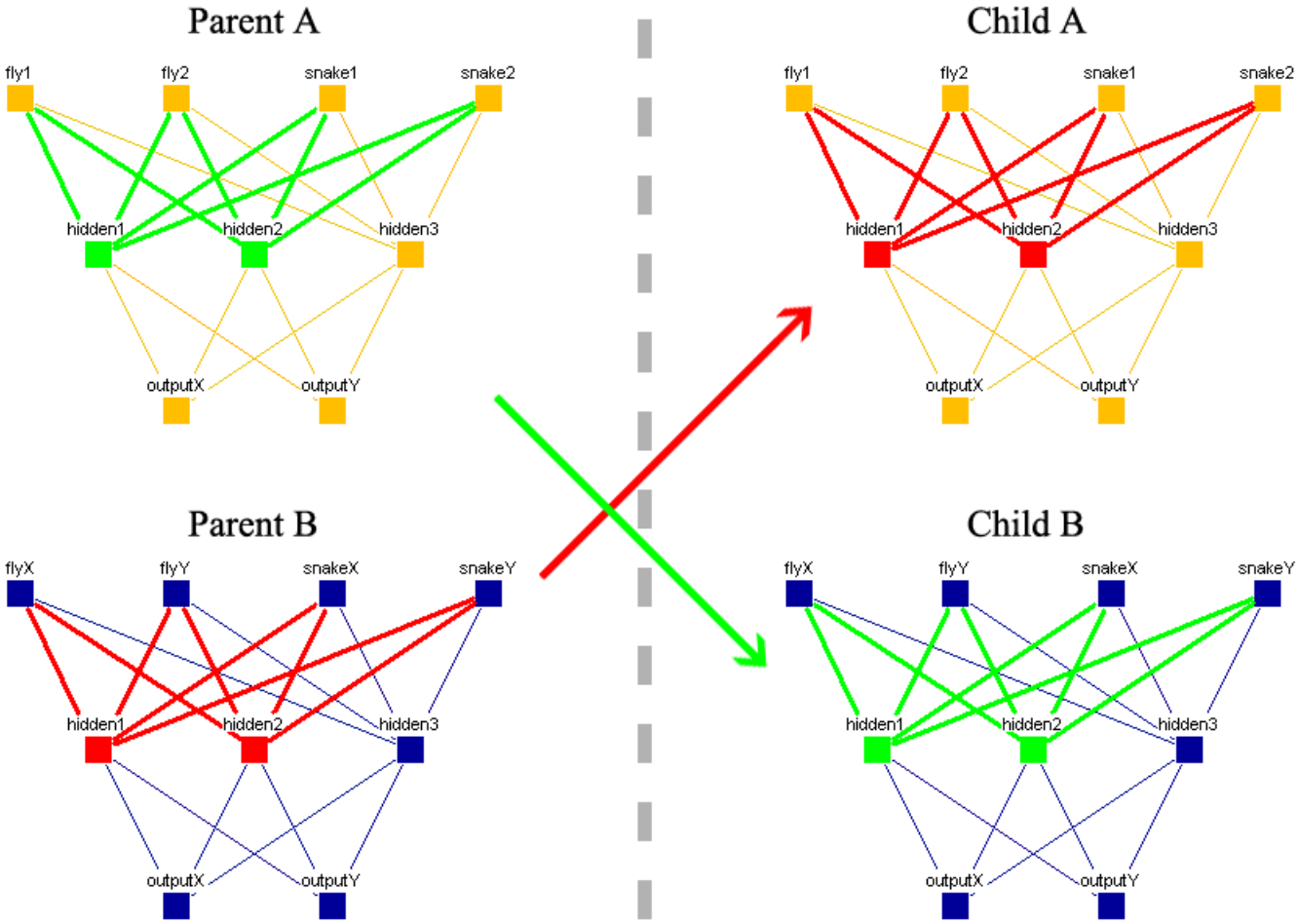


Figure 5: A crossover operation applied to neural networks. Here, there are two genes crossed over.

Neuroevolution generally performs best if each neuron and its input weights are treated as a single, cohesive gene. This is because a neuron's input weights are generally fine-tuned in relation to each other in a well-performing network, so crossing over halfway between weights will likely ruin the neuron's precise configuration. For example, the crossover depicted in Figure 5 involves *two* genes, each consisting of four weights. Mutation does not work this way however; the weights are perturbed individually.

3.2 Neuroevolution in Our Game

Automating the steering of the frog in our game seemed like a perfect opportunity to apply neuroevolution. Our aim was to train a neural network that, given a set of real-time game data (such as the positions of the nearest flies, snakes, obstacles and lakes), would output an appropriate (x, y) steering for the frog. It would be difficult to use the backpropagation for training such a network, because it is not clear what constitutes a

"good" steering vector at each instant in the game.[1] However, it is relatively easy to assess the performance of a frog after some time has passed by considering how many flies it caught and how many times it was hurt by snakes.

3.2.1 Network Architecture

As we alluded to above, the inputs our networks receive are the (x, y) positions of the nearest flies, snakes, obstacles and lakes. Our implementation is dynamic so that the number of nearest objects of each type can be modified easily or omitted. This was so that we could 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.

The inputs are calculated relative to the frog; i.e. the vector from the frog to each object is used, not the object's world co-ordinates. It is common to also normalise rotations to the frog's co-ordinate system, but we experimented with this approach and found that it was not of much help. The problem with taking the frog's rotation into account 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). These are presented as different inputs if the input is normalised, but the frog's behaviour should probably be the same either way.

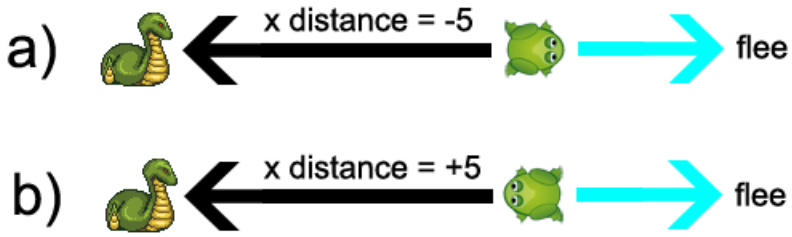


Figure 6: a) ??? b) ???

Our ultimate approach makes full use of the roational symmetry of the game. We present the input multiple times with different rotations applied, calculate the outputs, then un-rotate the outputs and calculate the average steering.

One final note for the neural net input is that using the vector differential directly is not a good idea. It means that the closer and object gets, the weaker the signal it sends!

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

where V is the vector to the object and k is a constant that controls dampening (equal to 10 in our implementation)

We chose to use a single hidden layer for simplicity, because in general the more complex the architecture, the longer the net takes to train (whether via backpropagation or evolution). (See figure to see how complex it is!) For the same reason we do not evolve connections, just weights. Normally neural nets squash input between (0, 1) but that doesn't make sense for us because an object could be to the left (-ve) or right (+ve) for example. Instead we squashed to (-1, 1). Don't apply a bias to this because it wouldn't make sense! Instead we maintain a weight that controls the gradient (sharpness of the rise).

For output, note that everything is continuous! This was one of the coolest things about our approach. We didn't just have four outputs representing strategies.

3.2.2 Fitness function

Say what it was!

3.2.3 Challenges

In this section we discuss some of the challenges that...

[1] It would be possible to record a set of data from human play and train the frog to mimic such play, but we wanted the frog to learn for itself.

3.2.4 Smart, Yet Undesirable Behaviour

A major problem we encountered was that the frogs often found a "good" solution in terms of yielding high fitness, but the behaviour of the best performing frogs was not what we wanted or expected. The below are just some examples:

Problem	Solution
If the training pen contained many flies and the flies were configured to move around with a "wander" steering behaviour, the frogs only learnt to avoid the snakes because chances were that they would catch several 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 the advantage. 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.	Making the flies stationary during training so that the frogs that deliberately targeted them got more of an advantage. Finely tuning the number of flies in the training pen and the relative sizes of the reward for catching a fly and the punishment for being hit by a snake.
Since the punishment for being hit by a snake is much larger than the reward for catching a fly, the frogs rightly learnt how to protect themselves first. Unfortunately, the best way to do this is to run into a lake and stay there! Once the entire population learnt this behaviour, learning stagnated because the frogs were never leaving the lake and had little opportunity to learn about the reward available through catching flies.	Introducing a "water camping" penalty to the fitness function. If the frogs are already full on water but continue to stay in a lake, their score is gradually decremented. (Although there is a grace period so that frogs do not have to leave immediately after topping up.)
Originally the training runs were fairly short (20 seconds long), but this was not long enough for poor behaviours to be exposed. For example, frogs that only targeted flies and disregarded the snakes would tend to rank first every time, because with a population size of around 50, one of the frogs would tend to get a favourable run where the snakes wandered away from the nearby flies. This encouraged excess aggression.	Increasing the time limit to 2 minutes per training run and using the same fly distribution for each pen. The fly distribution is still randomised at the start of each epoch, but the distribution is shared. Also using ranked roulette instead of proportional (?)

3.2.5 Training Time

As discussed in the third problem above, each training run needed to be reasonably long in order for the genetic algorithm to work properly. However, we found that it generally took around 1,500 epochs for the frogs' learning to start tapering off (see Figure ? later). For a population size of 40 (which is fairly typical in these types of experiments), this would have taken an extremely long time to complete! Our first measure to combat this was fairly obvious – we increased the Unity time scale to speed up the training runs. We found that a speed up factor of around 3 – 4 was appropriate (at higher speeds Unity became jerky). Our second measure was to clone 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.

3.2.6 Obstacle Avoidance

In the original game, the frog was controlled by mouse clicks and automatically avoided obstacles via A* (and a few additional tricks to avoid corner collisions). This time the frog was controlled by a neural network, which for out network was essentially just a type of steering behaviour. This created a problem, because A* and steering behaviours do not really gel. A* requires a goal note, but steering behaviours only provide a direction of travel, not a distance. Our solution to this problem 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 means that the frog is 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.

3.2.7 Shooting Bubbles

We wanted our frog to be able to shoot bubbles after filling up on water in the lakes, but we decided that having the frog learn to shoot from scratch would be too difficult (given the time limits) since opportunities

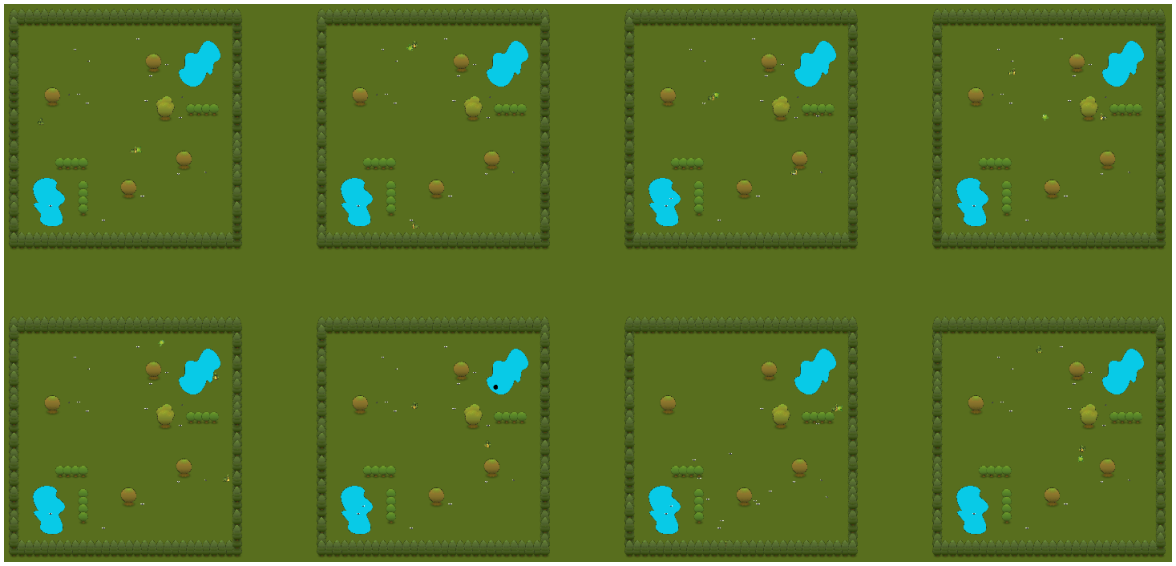


Figure 7: ???

to fire on snakes are limited and the overall plan of (obtain water – approach snake – fire bubble at correct angle) is quite a complex one, especially given that the frogs generally learn to avoid snakes. Our solution was to simply make the frog fire directly at snakes whenever they are within a certain range and the frog's water level allows it. We made it so that the shot range was an additional gene in the frog chromosome, so the frogs learned when to shoot, but not how to shoot.

[TO DO: Mention representation of bubbled snakes!]

3.3 Outcome

In the course of all our experiments, we found that the frogs generally learned tasks in a particular order. Since the fitness loss from being hit by a snake was far more than the benefit from catching a fly, the frogs learned the art of self preservation first. Before we added the lakes, this meant fleeing in the opposite direction from snakes. After we added the lakes, this meant diving straight into the lakes at the start of each training run. After learning self preservation, the frogs that had a tendency to catch more flies (or to leave the lake, after we added the "water camping" penalty) gradually became more and more dominant. Their behaviour gradually became more daring when flies were close to the snakes, and their flee patterns became much slipperier!

Given that we did not pre-program any behaviour for the frogs (besides cheating with the water bubbles), and the output of the neural nets was simply a steering (not just a selection between different behaviours), we were very happy with the performance of the frogs. Aside from our qualitative assessment that the frogs appeared to become much more intelligent during training, we were able to confirm that the genetic algorithm worked in a scientific sense by plotting the total population fitness versus epoch (see Figure ?)

INSERT FIGURE

There was a lot of variance in the fitness from epoch to epoch due to the random spawn locations of the flies, but there was a noticeable increase in performance until around epoch 1,500. The population fitness reach a peak of ?, corresponding to an average fitness of ? per frog, which is equivalent to ? (some number of flies).