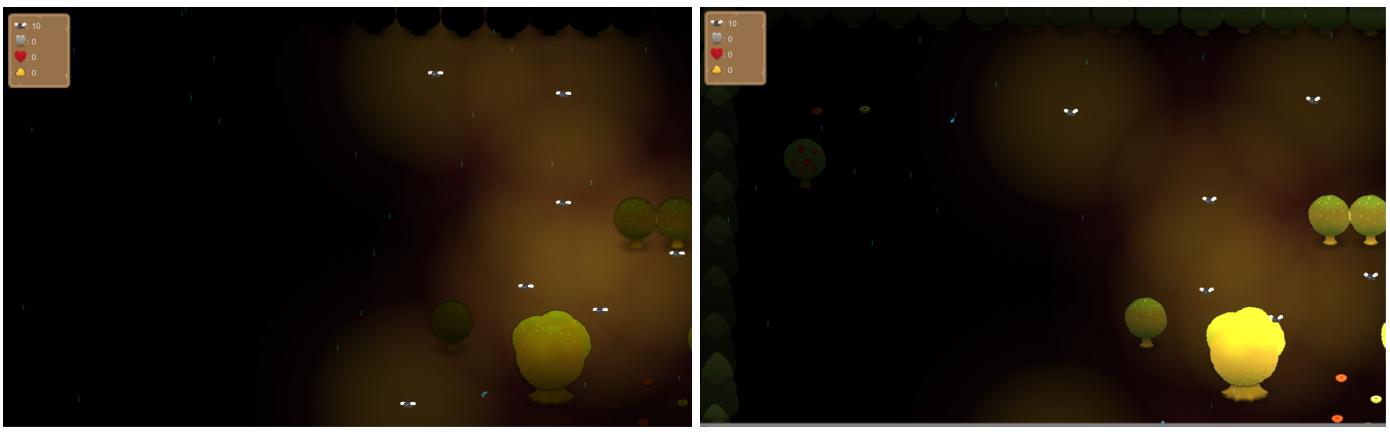
Games and Artificial Intelligence Techniques
Assignment 2

Team 3: Michael Dann and Jayden Ivanovic

October 13, 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
```

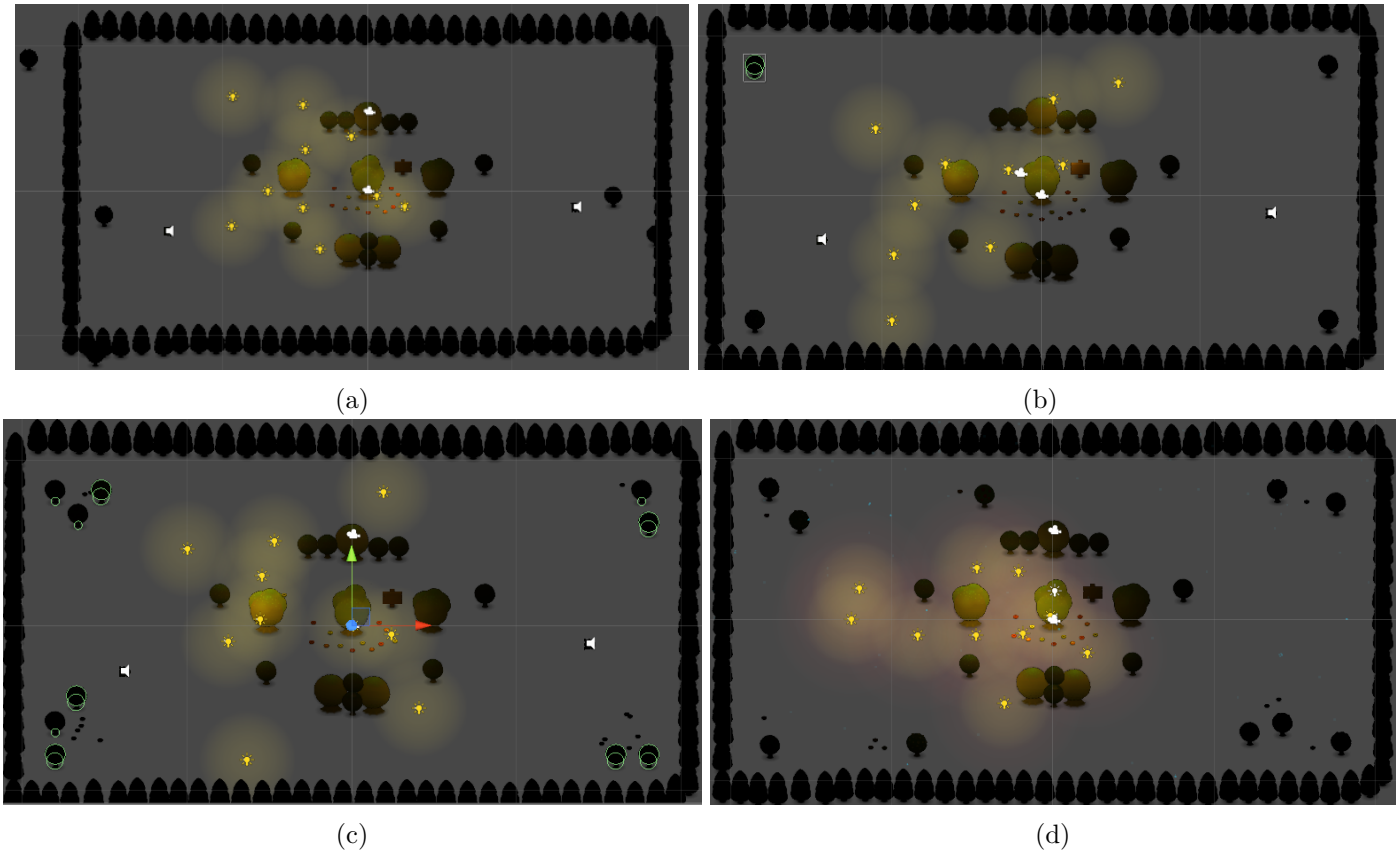
We employ the use of GA's in our work and believe they are strongly applicable for the following reasons:

- 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.

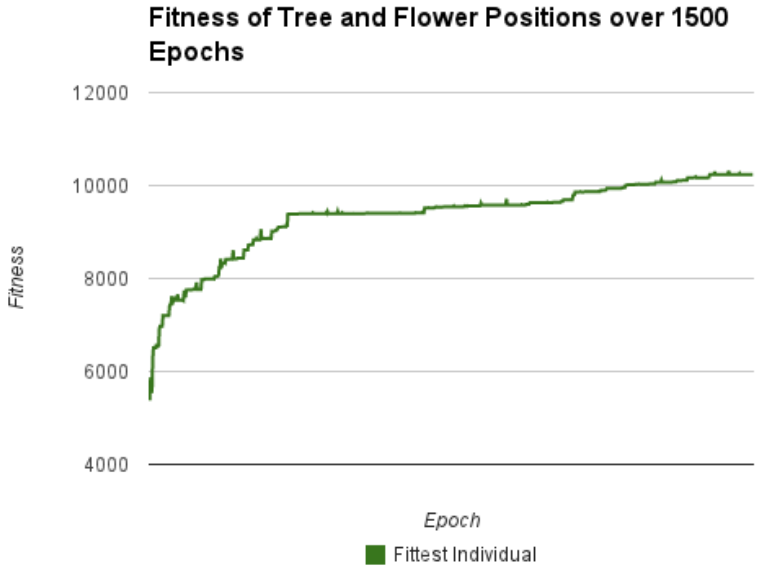
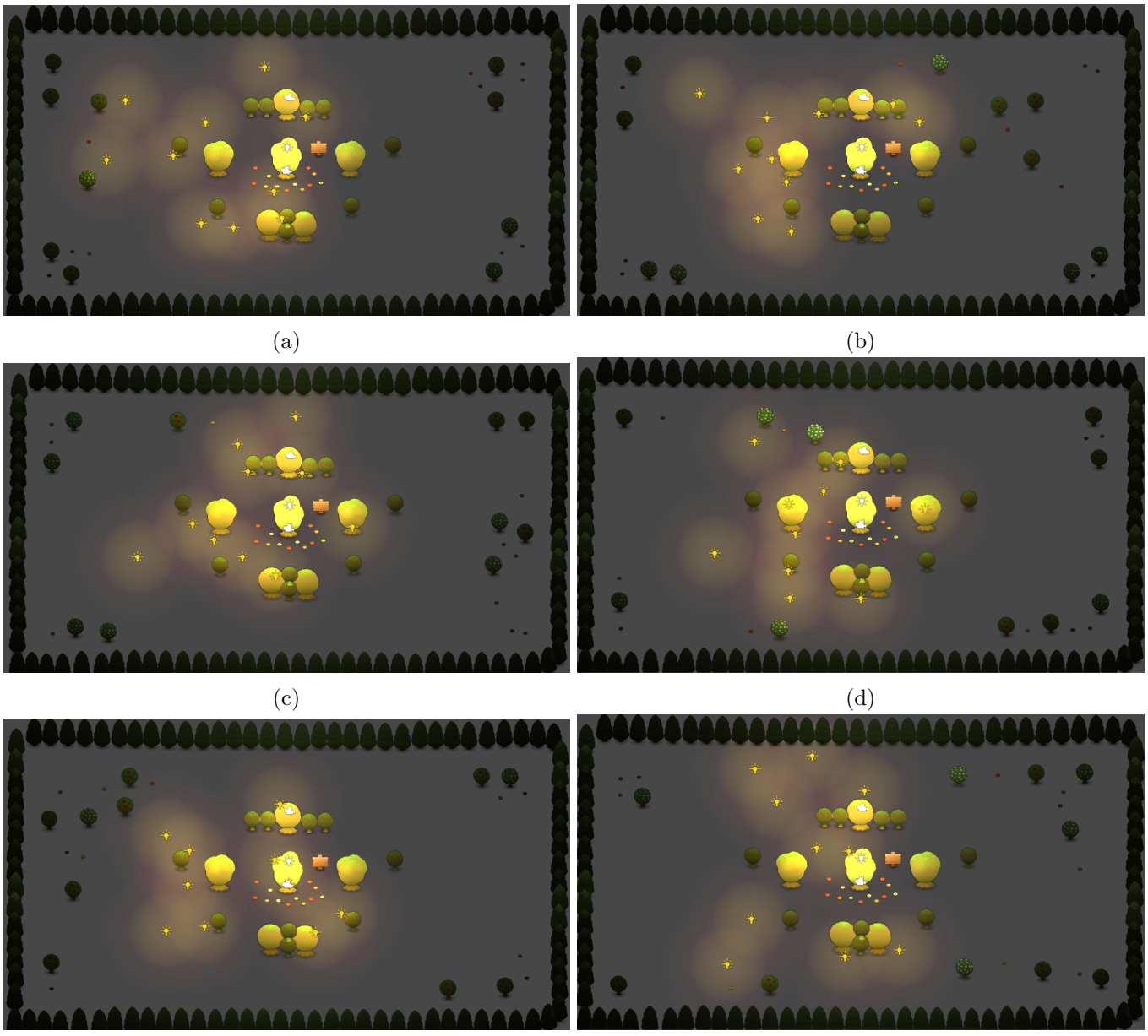


Figure 3: Fitness of the population over 1500 Epochs.



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

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