1 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.

1.1 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($child2$)
    $children \leftarrow children + child1$
    $children \leftarrow children + child2$
    $numChildren \leftarrow numChildren + 2$
end while
$population \leftarrow children$
```

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.

1.2 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.

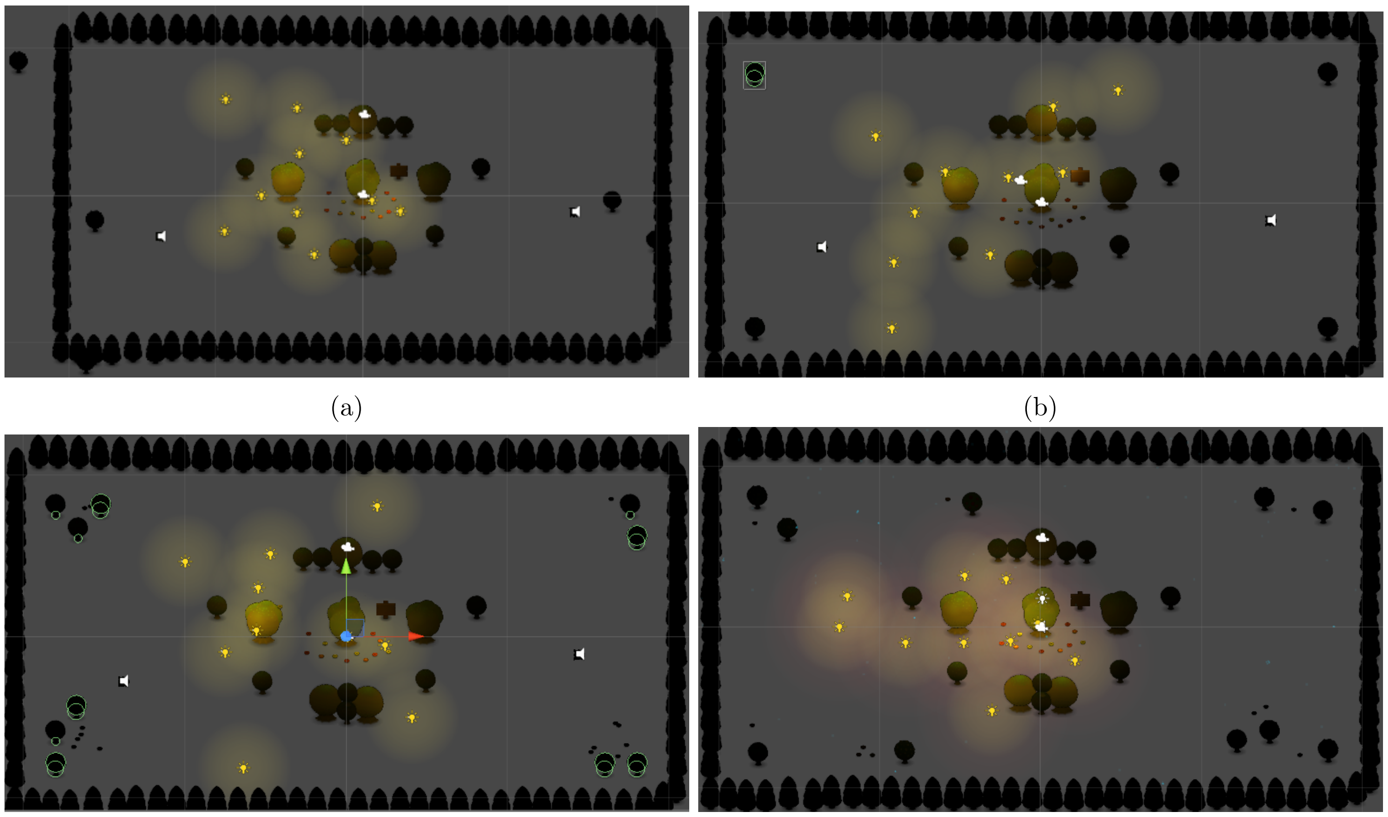
(a) (b) (c) (d)

Figure 1: The steps to implementing an appropriate fitness function. In 1a 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 1b 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 simply run for less generations. Figure 1c 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 1d 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.

1.3 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 ??.

1.3.1 Succint Fitness Function

1.3.2 Generating Positions at Runtime

1.4 Outcome

Figure 2 shows one of the solutions generated by the final fitness functions and parameter set.

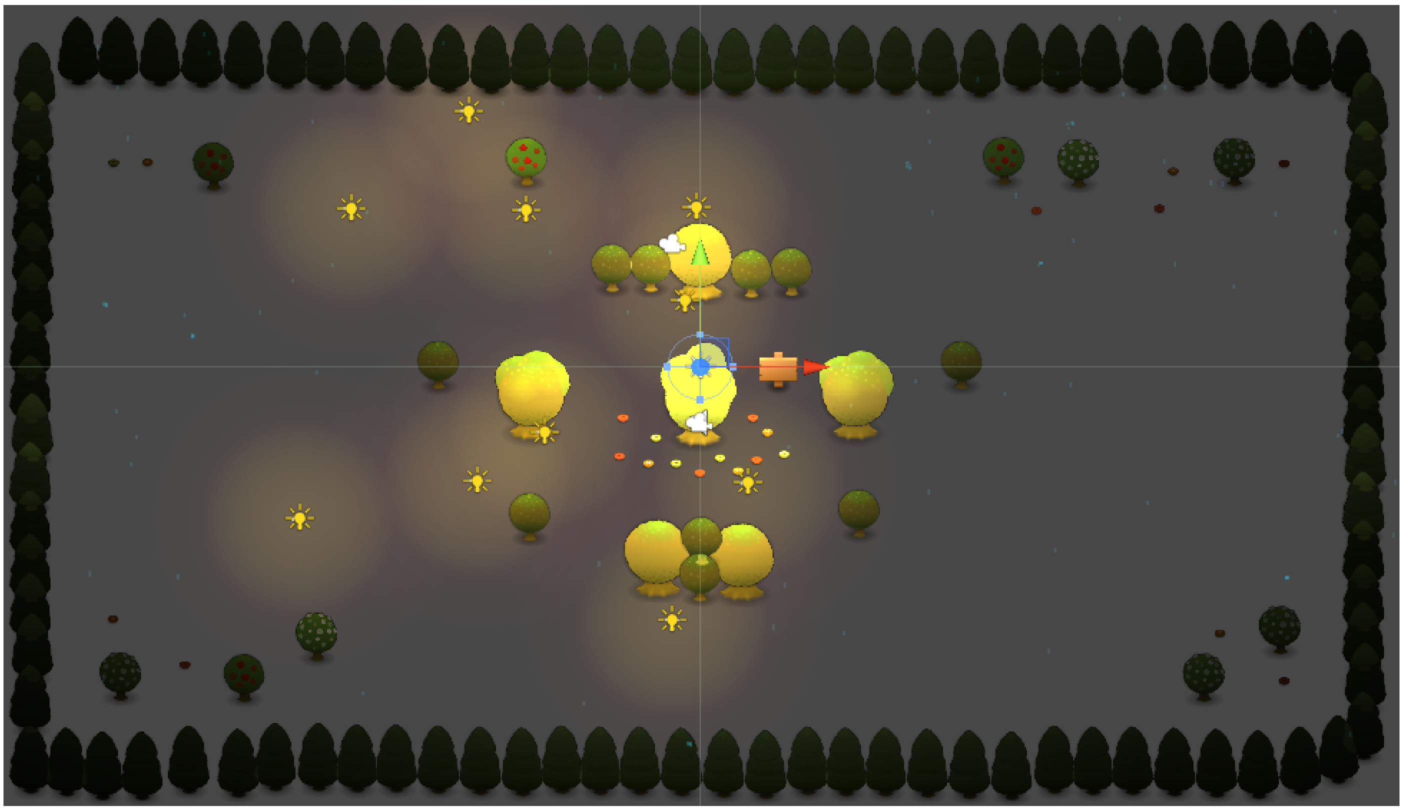

Figure 2: One of the solutions that is generated via our GA.