Mark Medved and Lucas Bryant

Professor Daniel Shapiro

CMPM 146 - Game AI

16 November 2020

P6 Writeup

Introduction:

This paper describes in detail the nature of the Sixth Programming Assignment for this class. P6 is quite complex;at its core is a rather extensive file called ga.py, for which it may be difficult to discern which changes were implemented by the students. For all intents and purposes, this writeup is a complicated README. This writeup comprises three parts. First, in order to establish a baseline understanding, we will describe the underlying genetic algorithm concepts and give an overview of our programming approach.. Next, we will go through a point-by-point explanation of each of the areas listed in the “Your Job” section of the instructions. Finally, we will discuss our favorite generated level.

1) Understanding the Concept:

To establish the level of understanding from which we are working, let us first review the notion of a genetic algorithm. To quote Professor Shapiro in class, a genetic algorithm, at its core, is "something you can modify, something you can judge, and a way to turn the first into the second." We generally call these a 'genotype', 'phenotype', and 'expression' respectively. In other words, we start some sort of data structure, most often an array of floating point variables. Then, we create some method of understanding what that data structure actually represents (or 'expresses') in regard to some larger purpose in the software, be it a flower, mesh or a *Super Mario Bros.* level. Finally, we create some means of evaluating how 'successful' the product of that expression is. This may be through user evaluation, or it may be through an automated process using a mathematical function, or anything in between.

Taking this idea one step further, we can consider what would happen by creating a slightly modified genotype from an existing one. That is, take an array in our program and tweak one of the numbers a little. If we apply our expression, we should arrive at a phenotype that looks relatively similar to the phenotype of the original genome. This new phenotype could be better or worse according to its evaluation, but it should at least be similar. By doing this in rapid succession, you can slowly change your genome into something entirely different via a smooth process from start to finish. It creates a kind of 'mechanical creativity', iterating on former designs without the need for human intervention. If our phenotype judgment is logically sound, we can even select for the 'better' mutations, and eventually take a very simple genome and evolve it into something much more complex and successful.

In the particular case of this assignment, our genotype was a textual representation of a Mario level, our expression was the functional code which allowed said text file to be interpreted

by the Unity platform, and our phenotype judgment is created through an arbitrary fitness function.

2) Our Job:

We modified or implemented the following functions for the Individual_Grid and Individual_DE encodings, respectively.

Individual_Grid

generate_successors():

- In our generate_successors method, the selection methods we used were ‘elitism’ and a slightly modified version of ‘tournament selection’. The population would be randomly split into two halves, the two halves are then each sorted by the fitness, index[0] being the best in each array, index[1] being the second best, and so on. The first (i.e the best) elements from each sorted half are crossed with each other, then the second best are crossed with each other, and so on. These new children alongside half of the previous generations best members create the next population, utilizing elitism.

generate_children():

- In our file, crossover simply implements the uniform crossover, at any given position randomly choosing a gene from either parent.

mutate():

- Currently in our implementation, mutate has a rate of 100 percent; it does a lot of cleanup in the genome. For example, it eliminates pipes that are above a certain elevation as well as connecting existing pipes below a certain level to the ground. It also changes the elements around according to a list of weights. The weights change as the position gets closer to the ground. It prioritizes open spaces mixed with clusters of blocks to create maze-like jumping puzzles.

Individual_DE:

mutate():

- In Individual_DE, the algorithm first decides whether or not to mutate based on a rate, initially at 10 percent. Then it randomly chooses a design element to change in the genome.
- Depending on what type of element it is, it will randomly decide by some percentage a modification to make. Some of the times it may offset the x position by some number and other times offset the y position by a random value or make other elements wider or taller by a random value.
- Mutations are specific to the type of element. For example, if an element is breakable or contains a powerup, a mutation may make it unbreakable or delete the powerup.
- The **modifications** we made to this function was simply changing the mutation rate to create more variance in the population

generate_children()

- The individual_de crossover method decides on two random numbers between 0 and the length of genome for both of the parents, **pa** for parent A, **pb** for parent B. It then splits each parent's genome by the number it generated for each parent where pa determines the split index for parent A and pb determines the split index for parent B. This creates 4 different sub arrays, or cuts:
    - From parent A we get [:pa] and [pa:]
    - From parent B we get [:pb] and [pb:]

- It then creates two new synthesized arrays from the 4 different cuts. It combines the **second** cut of **parent A** with the **first** cut of **parent B** and it combines the **first** cut of **parent A** with the **second** cut of **parent B.**

The Fitness Function:

- A few changes were made to the fitness function in Individual_DE, mostly penalizing a few levels with certain combinations of design elements. On top of penalizing too many stairs, we added a check to see how many pipes there were in the level. If there were too many tall pipes that level would be penalized as tall pipes mean they cannot be jumped over. If there were too many holes in the ground, that was also penalized because oftentimes levels would be made that could not be traversed. Another penalty was added to levels with too many enemies as it could be too difficult to traverse.

3) Our Favorite Level:

```
Generated successors in: 4.273706674575806 seconds
Calculated fitnesses in: 7.65692663192749 seconds
Generation: 225
Max fitness: 7.645184291685509
Average generation time: 11.273176214430068
Net time: 2536.464648246765
# output for execution after 225 generations
```

We ran our algorithm over an initial population of empty and random levels for 225 generations. Initially, the fitness for the levels was maxed out around 3.1 but after a while and 225 generations, we managed to reach a fitness of 7.6, more than double what was initially being generated. It is, in our opinion, a satisfactory level that may seem random and impossible. It requires many unique jumps to traverse the maze-like landscape, avoiding critically placed enemies to reach the goal. It can be fairly difficult at first but is satisfying to traverse.

*****One important note, the flag was manually moved over one space as it was generated offscreen and unreachable. It is the only thing manually created/edited in the level**