P6: Evolving Mario Levels Writeup

Individual_Grid

For the first encoding of the genetic algorithm, Individual_Grid, we modified the population initialization, selection, crossover, and mutation processes.

For our **selection** mechanism, we used two different strategies of selection, tournament selection and elitism selection. In tournament selection, pairs of the population are pit against each other, returning the winning ones. Running this through the whole population essentially returns half of the population with the best fitness function. The winners of tournament selection were then passed to crossover and mutation. In elitism selection, we returned the top 5 individuals with the best fitness scores, without performing crossover or mutation. We combined these two to allow for the successors to consist of the children bred from the better half of the population, and the top percent of the last generation.

For our crossover function, we performed a single-point **crossover** on two parents (*self* and *other*) and returned their two children (*child_a* and *child_b*). The function randomly selects a single point on both genomes, which is used as the crossover point. Up until the crossover point, *child_a* receives the first parent's genome, and after that point *child_a* receives the second parent's genome, and vice versa for *child_b*. We used this method of crossover because it didn't mess up how our pipes were done, the platforms of 2-3 blocks that we made, or other particular aesthetic designs we implemented.

For our **mutate** function, we decided to have two types of mutation operators. We first randomly select an individual gene (row, column) pair. Our first mutation performed at a 10% chance, and made a completely random change to that part of the individual's genome. We took that individual gene and randomly selected a block type to change it to. Our second mutation occurs at a 40% chance (and only if the other mutation did not occur) and is more of a "beneficial" mutation (i.e. a modification that we identify as beneficial or more fun to play). For example, if the block we picked was a single block (no neighbors), we would add a block next to it, to make it a more aesthetically pleasing platform. We also removed question mark blocks that could not be reached.

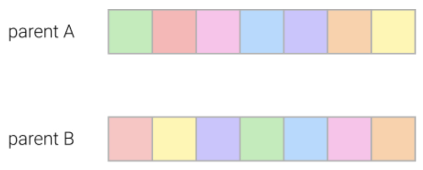
Additionally, we modified our *random_individual* function to start with better **population initialization**. Instead of randomly generating each block with equal probability, we added weights and conditions under which we would make any block. For example, “X” blocks are generated with a 4% chance, given that it is not the very top row, “?” blocks are generated with a 2% chance, given that it is above the 5th row from the bottom and not in the top 3 rows, etc. After generating each part of the map, we overwrite some coordinates so that the map makes sense. For example, we add enemies on top of randomly selected “X” or “B” blocks, make sure pipe segments are consistent with pipe tops, and generate a random number of pits.

Individual_DE

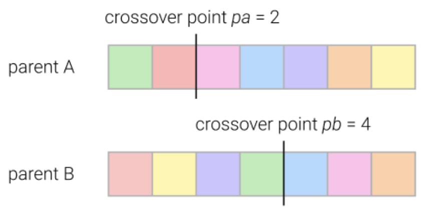
In the second encoding, Individual_DE, a genome and another genome are passed into generate_children. Crossover of the two genomes (or parents) is performed, and the results of crossover (two different child genomes) are each mutated using the mutate function.

The crossover function receives two genomes as input, performs single-point **crossover** between the two, and returns two new children. We felt that a diagram would better show how crossover is performed in *generate_children* (it works similarly to how we wrote ours):

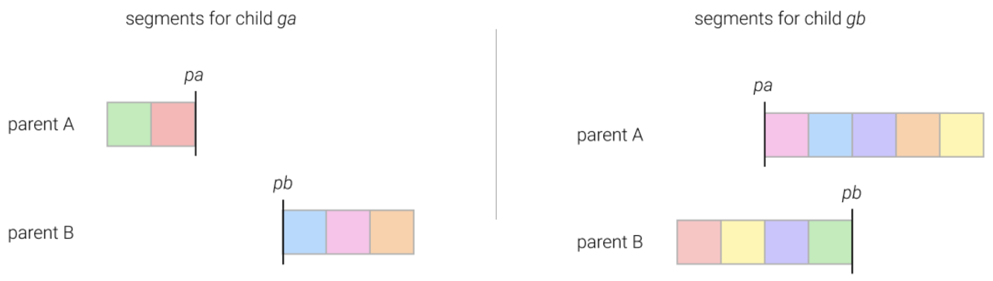
1 Start with two selected parent genomes



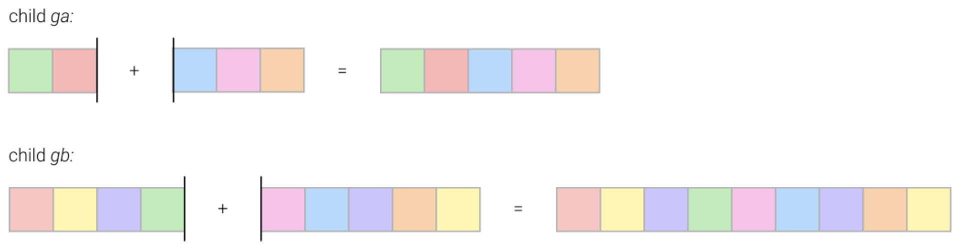
2 Select a random point *pa* on the genome of the first parent, and a random point *pb* on the second parent



3 Check that these are valid indices, then use the two crossover points to create two separate series of segments



4 Create two children using the two pieces taken from the two parents



After producing child genomes, each child is used as input for the **mutate** function. The purpose of this function is to add a random modification that provides more diversity. There is a set chance that an individual will receive a modification, in the code given the mutation rate is 0.1. If we decide that there is a mutation, we then randomly select a random gene we want to perturb (genes include blocks, coins, holes, stairs, platform, enemy, etc.). Each gene has several properties that can be modified, such as x-location, y-location, or a more specific property that only certain genes have (i.e. a ? block may either have a coin or a powerup, a block may be breakable or not, etc.). Using another random number, we select which property to modify (i.e. there is ⅓ chance of us modifying the x-value of a breakable block, ⅓ chance of modifying the y-value, and ⅓ chance of inverting its breakable property). We modify the values using a *offset_by_upto* helper function, which takes in a value and randomly adds some amount of variance (ensuring that it is within some minimum/maximum bounds). After we select the modified trait and new value, we create a new genome with the modified gene.

We also made small modifications to the mutation operator and fitness function of Individual_DE. We changed the mutation rate to 20%, rather than 10%, as the maps started out relatively empty and we wanted to add more diversity. We found that running the code for more generations seemed to make improvement, so we only made slight modifications to our fitness function. Some modifications we made were penalizing when there were more than 10 stairs, rather than more than 5, and adding *meaningfulJumps* as a metric for the fitness function. We made other smaller modifications to the weights, such as lowering the importance of *negativeSpace.*

Our Favorite Level

We submitted this level because we thought it has a nice diversity of the amount of enemies, and creative jumps that you need to perform in order to win the level. In addition, there are lots of paths you can choose to take when completing the level, and we liked that aspect to this level. The pipes are fun, yet solvable heights, the goombas are numerous, but not too numerous, making it difficult but not impossible. We also liked the challenge of having not as many ? blocks, so that you had to be precise and careful with your jumps without powerups. This level took about 50 generations to complete, starting with a small population of 50. Our code was a little slow, the beginning generations took about 0.25 seconds per generation, but the latter ones took about 0.05. In total time it took about 50 seconds to run the entire program. Many of our levels were enjoyable, and our second favorite level was similar to the one we submitted. They are both solvable, but have challenges and difficult jumps that the player has to make.

We do not want to participate in the extra credit competition.