P6 Writeup

Partners: Joshua Husting, Jan Yu

Selection Strategy:

For our selection strategy, we employed *elitist selection*, as we felt it made the most sense. First we take the population and sort it from highest to least fit. We then take the fittest $1/8^{th}$ of the population, and use generate children on them. We iterate through the top $1/8^{th}$, and call generate_children on a random individual within that $1/8^{th}$. We do this 4 times, as our generate_children function returns two children, which gives us a population of the same size of our original population.

We also tried random selection, however that got pretty bad results so we decided to stick with elitist selection. Here's what we did for random selection: we picked two random indexes, ra and rb, population/2 number of times, then appended to the new population both children from generate_children. We only generated children a set amount of times so we can keep the length of the population at a constant amount.

Changes for Grid:

For Individual_Grid we were required to change mutate() and generate_children(). We also changed random_individual(). For generate_children() we used the *single-point crossover* strategy. We picked a random width index, and for the resulting child every column before that index was from the left parent, and every column after that index was from the right parent. For our mutate function, we looked at every grid element and gave it a 15% chance to mutate. We used a helper function to determine what type of tile could be placed at that grid location. For instance, a pipe can only be placed at the bottom of the level, or on top of another pipe. We start with a dictionary containing the percent chance of each tile spawning, then if a tile cannot be placed at the location, we set its chance to spawn to 0.

For random_individual(), we do the same process as mutate(), where instead of picking randomly, we pick randomly from a set of tiles that could be placed where we are. We did not change much about the fitness calculator, as we felt that it was good enough.

DE Explanation:

Mutation:

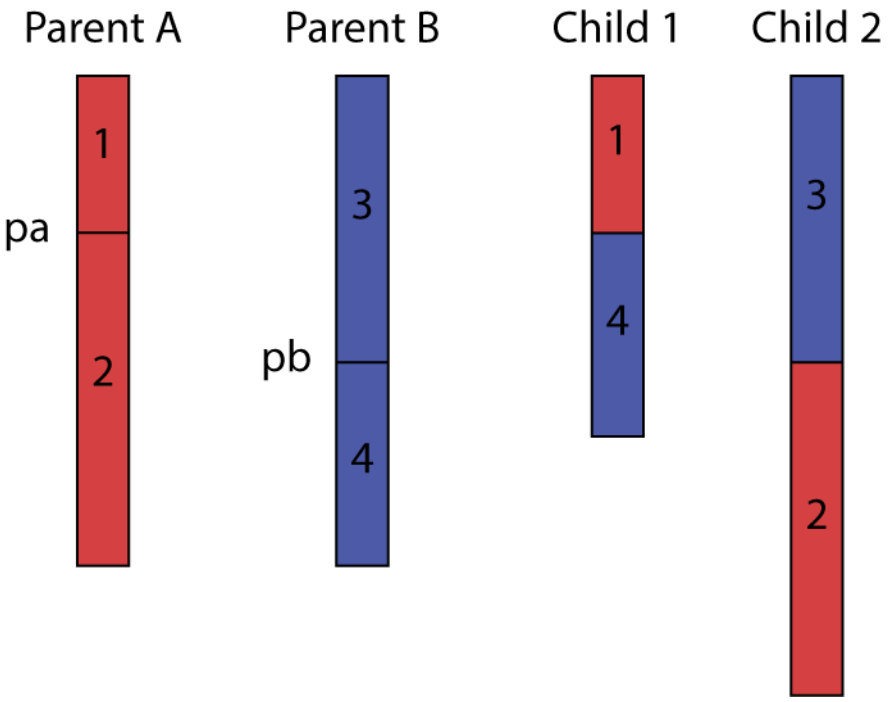
The provided mutate function works very similar to ours, but we have our differences. We both have a 10% chance of mutating a random element. However, for DE, "mutating" an element doesn't mean changing it to another DE, it means changing how the DE is represented. For instance, mutating a block means picking between 3 options: moving the block some value left or right, moving the block some value up or down, or inverting whether or not that block is breakable.

Crossover:

DE's crossover function works a little different than ours but is a very similar idea. It picks two random points, pa and pb, and sets those as the break points. Crossover will generate two children: one child will be all the elements in genome a before pa + all the elements in genome

b after pb. The second child will be all the elements in genome b before pb + all the elements in genome a after pa.

Here's a diagram of how crossover works:



Changes for DE:

For DE we made a couple changes: for instance, we felt as we were getting too many holes, that were too big. So we lowered the max hole width to 4 down from 8, and added a penalty in our fitness function if the genome ever had more than 3 holes. We also felt that enemies weren't being placed as often as we liked, so we added a penalty in the fitness function if there was ever less than 5 enemies. I also changed random_individual to generate between 50 and 128 elements, instead of 8 and 128. One more change we did, is we added to the fitness len(genome)/400 which makes it prefer genomes with more design elements.

Competition?

We would like to submit favorite_Grid for the competition. Thanks!