<u>**P5 WriteUp**</u>

For *generate_successors()*, we first implemented an *elitist* strategy to filter out and remove any individuals within the population with negative fitness values. We allowed only individuals with positive fitness values to generate children for the next generation until we reached the original population size that we initially started with. To determine the parents of the next generation, we implemented and used a helper function called *get_parent()*. *get_parent()* used a combination of two selection strategies: *tournament selection* and *roulette selection*. We first randomly generated a number between 1 and 10 to make a 50/50 decision on which of the two selection strategies to take. If the randomly generated number was less than or equal to 5, we took *tournament selection* where we would randomly select two individuals and pick the individual with the higher fitness value as the new parent. If the randomly generated number was greater than 5, we took *roulette selection*. In *roulette selection*, we first normalized the entire population's fitness values by adding a *const* and dividing that sum by the sum of all fitness values of the population. After that, we appended these normalized fitness values and their respective individuals to a list called *normalizedList*. We then randomly selected a *chosen* fitness value within the *normalizedList*. This *chosen* value is then used as a comparator against all values within *normalizedList* and once a value greater than *chosen* is found, it is then selected.

For *generate_children()*, we randomly chose a position among each row in the genome grid to split the row into two parts. The left part of our split, we copied the genes from *self.genome* while the right part was the genes were copied from *other.genome*. This allows the crossover algorithm to mix both parents' genomes and randomly implement a different crossover for each row. We therefore can create more diverse offspring and children from which we further diversify through mutation. For *mutate()*, we analyzed the genome and added new elements that were not present previously. For example, we added pipes, platforms, enemies, coins, pits, and stairs if we did not find them present and counted how many of each element there were. We used randomness to place the new elements in the genome to necessitate mutation randomly. We also implemented constraints for placing obstacles in sensible locations using *obstables_zonex* and *obstables_zoney*. Our mutation implementation allows for evolution of the levels of the game as well as adding more playability to each level as our genetic algorithm explores new solutions.

In *Individual_DE*, we observe a different implementation of the crossover and mutation functions. In *generate_children()*, we first begin determining the random crossover points in the parent genomes of *self* and *other* using *'pa'* and *'pb'*. If a parent has a non-empty genome, then a random index is selected for the crossover point within the genome. Next, we create two parts called *'a_part'* and *'b_part'* that represent the genome divided based on the crossover points. The parent genome *self* starts at the beginning and ends at its crossover point *'pa'* while the parent genome *other* starts at *'pb'* and ends at the end of the genome. We create two new offspring genomes called *'ga'* and *'gb'* formed from concatenating the two parts earlier. Before we return these new offspring, we mutate them using the *mutate()* function. This therefore gives us our two new mutated genome offspring. Now let's look at the *mutate()* function. We first start by checking if we should actually perform a mutation every time *mutate()* is called. We give this a 10% chance of mutation happening based on a random chance. If we decide to mutate, then we randomly select an index within the genome to mutate called *'to_change'* and also randomly select an element from the genome called *'de'* to mutate as well. Next, we check the type of selected element called *'de_type'* which determines how the mutation will be applied. The different mutations vary among the different types of elements such as blocks, coins, pipes, stairs, platforms, or enemies. For example, we might toggle a block's breakability in *'4_block'*, change whether a question-mark block contains a power-up in *'5_qblock'*, or change the x, y-coordinates of a coin in *'3_coin'*. After we determine the mutation to be applied to the element, we remove the original element from the genome and insert the mutated element in its place. Since the genome is maintained on a heap, we use *heapq.heappush()* to insert the mutated elements. The *mutate()* function thereby ensures that the mutations and changes to the genome are more meaningful to playability of the game levels as well as exploring new solutions effectively.

For my two favorite levels, I loved the playability of both maps, especially the first level since there was this gap right before a pipe that required a very precise jump to get past. This took me a great number of tries until I finally got it even though it was a little annoying, it was still fun. The first level took 132 generations to generate in 300 seconds while the second level took 113 generations to generate in 252 seconds. The second level also had a similar gap before a pipe but was easier to get past. The second level also had these stairs I could use to jump across enemies and finish the level faster which was fun too. Both of my favorite levels I played also felt very sensitive to the controls and fast-paced which made timing of jumps and movement very crucial to surviving and beating the level. Overall, I enjoyed the two levels and the unique playability of each map.