Servin Wayne Vartanian
A14802532
COGS 160
Genetic Algorithms

Using Genetic Algorithms To Solve The Knapsack Problem

For this project, I utilized 3 Crossovers, 1 fitness, and 1 mutation function in 3 variations to run 3 different Genetic Algorithms. Below, I will explain what each crossover function and mutation function does/what their construction is like:

- Split Crossover Function: For my split crossover function, I randomly split each parent into two parts from a randomly selected index. The first offspring consisted of the first part from parent 1, and the second part from parent 2. The second offspring was vice versa.

- Reversed Crossover Function: For the reversed crossover function, what I did was again split each parent into two parts using a random index. However, this time what I did was I took the splits from each parent and reversed them using a reverse function I created earlier in the code. Offspring 1 was composed of the reversed-second part of parent 2 and the reversed-first part of parent 1. Offspring 2 was composed of the reversed-second part of parent 1 plus the reversed-first part of parent 2. Here's what I mean:
  Parent 1 = [1,2,3,4,5,6,7,8,9,10]
  Parent 2 = [11,12,13,14,15,16,17,18,19,20]

  Parent 1(part one) = [1,2,3,4,5]
  Parent 1(part two) = [6,7,8,9,10]
  Parent 2(part one) = [11,12,13,14,15]
  Parent 2(part two) = [16,17,18,19,20]

  Offspring 1 = [20,19,18,17,16,5,4,3,2,1]
  Offspring 2 = [10,9,8,7,6,15,14,13,12,11]

- Halved Crossover Function: For this halved crossover function, I simply split each parent down the middle. Offspring 1 consists of parent 1's first half, and parent 2's second half. Offspring 2 consisted of parent 2's first half and parent 1's second half.

- Mutation Function: What I did for my mutation function was take a random sample from the "individual" allocation input, and replace that random sample with a randomly chosen number of either 0 or a 1. This made it so the newly returned allocation individual would have one item from their list switched to a random 0 or a random 1.

In Figure 1, you can see the Fitness Table for this assignment. It has 3 entries from my 3 different GA/crossover combinations, as well as shows the final Dynamic Programming Solution to compare with my GA results:

Figure 1: Fitness Table

Algorithms	**Average Final Fitness Score**
Split Crossover GA	1743.6
Reversed Crossover GA	1601.2
Halved Crossover GA	1740.6
Dynamic Programming Solution	1757.0

Next what I will show are the fitness graphs of each of my GA/crossover combinations. Figure 2 shows the Fitness Score graph, per iteration, for my Split Crossover GA, Figure 3 shows the same but for my Reversed Crossover GA, and Figure 4 shows it for my Halved Crossover GA:

Figure 2: Split Crossover Fitness Graph

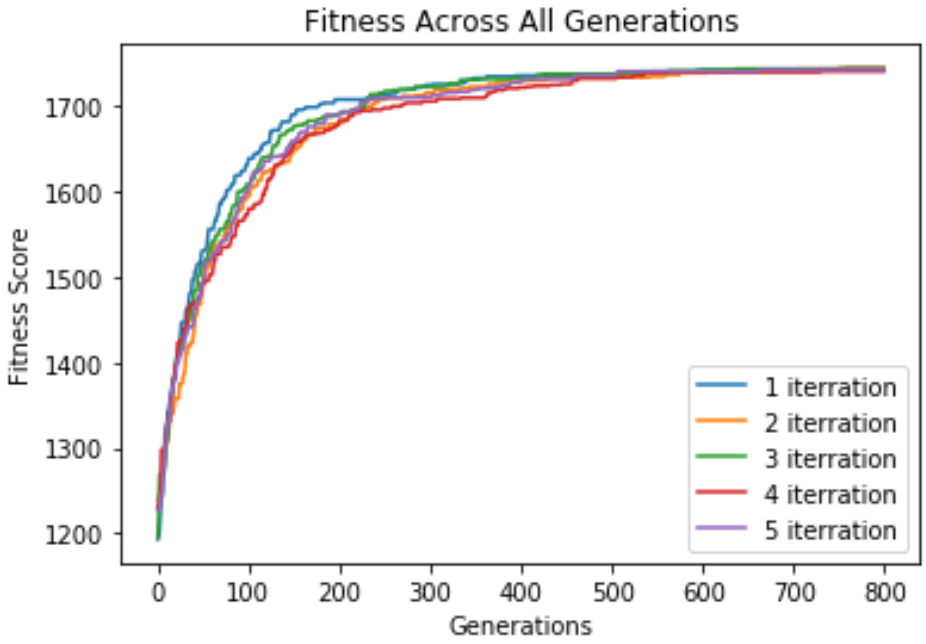


Figure 3: Reversed Crossover Fitness Graph

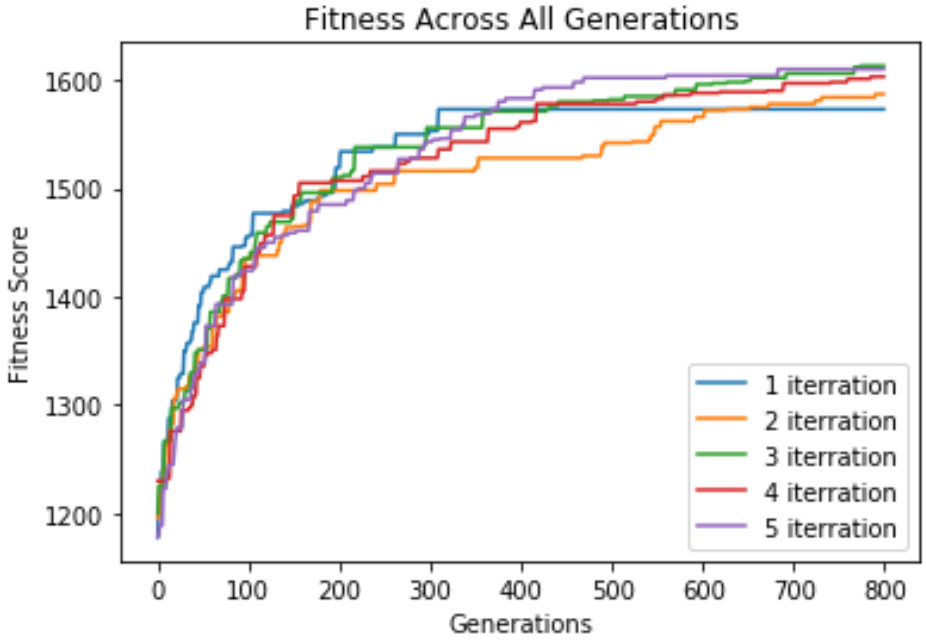
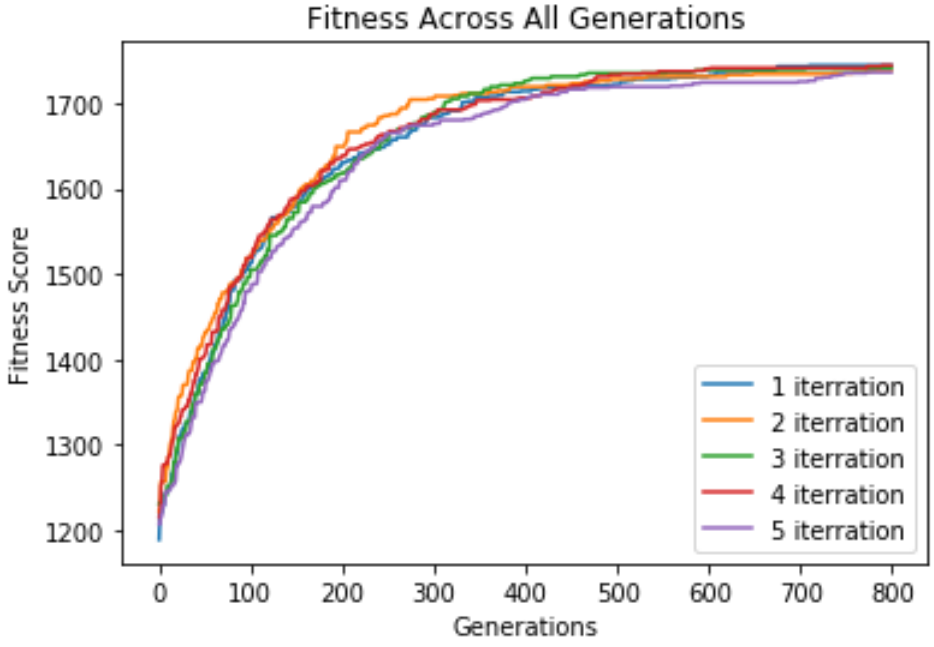


Figure 4: Halved Crossover Fitness Graph



With all this being said and done, I believe my best performing crossover function was the split crossover, followed by the halved crossover, and finally the reversed crossover being the weakest. This is not only because it reflects this in their Fitness scores, but also because I believe that reversed and split crossover are able to preserve most of the gene-continuity from each

parent, and when they crossover half (or a random slice) from each parent, any useful/well performing combination from the parents has a chance of being preserved in the slice. So part 1 from parent 1 which might be causing really good performance can be passed down to their offspring to be tested and preserved continually. However, in the reversed crossover I think when I “flip” the genes from end to beginning, their crossover is now technically in a different order than what it was before the flip. So if the combination of [1,2,3] did really well, giving an offspring [3,2,1] might not carry the same useful properties because the genetic order has been reversed. Due to this, their overall fitness is lower for 800 generations. Lastly, I believe my mutation function performed spectacularly as for each GA it gave very concise and consistent performances in mutating 1 entry from each individual per allocation! I think having it only mutate 1 item instead of mutating multiple items from the individual list ended up giving a high Fitness score by not accidentally getting rid of useful traits by mutating them away.