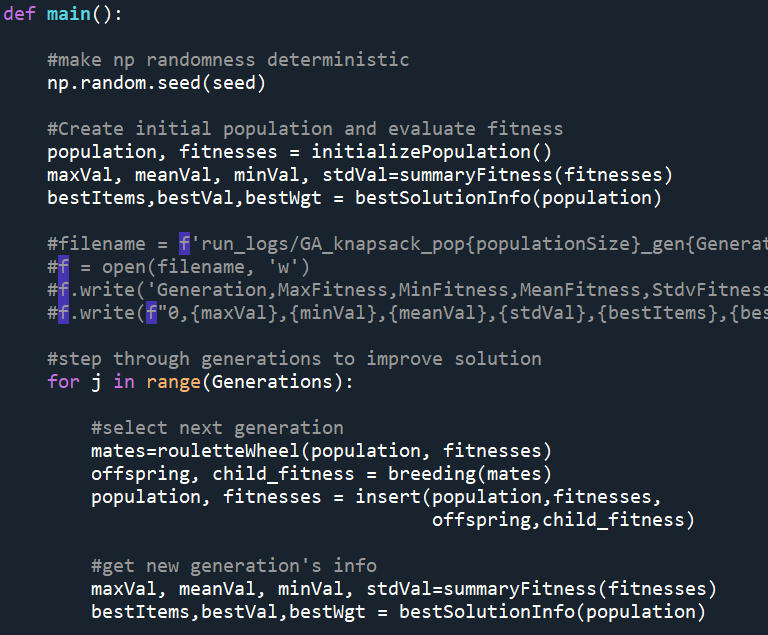
DSA AAM Homework 7 Group 11

Alex Nozka

Marshall Baldwin

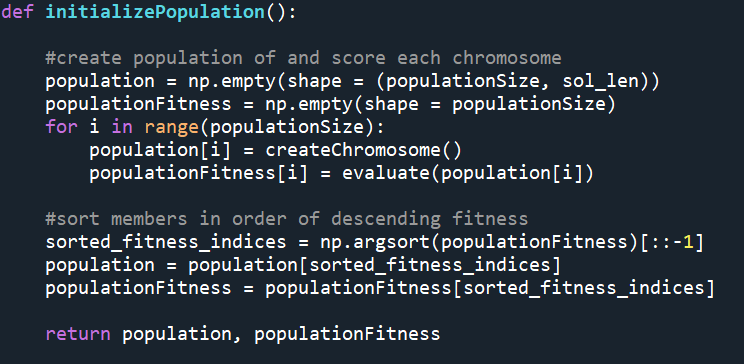
QUESTION 2:

This question involves building a genetic algorithm to solve the knapsack problem from the previous homework. The main logic of this program can be seen here:

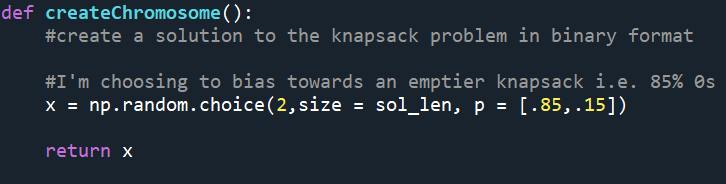


Walking through this code, it begins by creating an initial population of solutions and scoring each member with a fitness function. Then for *j* generations, the population is updated. This is done by selecting a pool of “mates” by selecting with replacement from the existing population pool subject to a probability density function defined by the proportion of total fitness score explained by a given solution. These mates are then used to breed a new population of solutions which are then scored. This new population is then subjected to the same algorithm after advancing to the next generation.

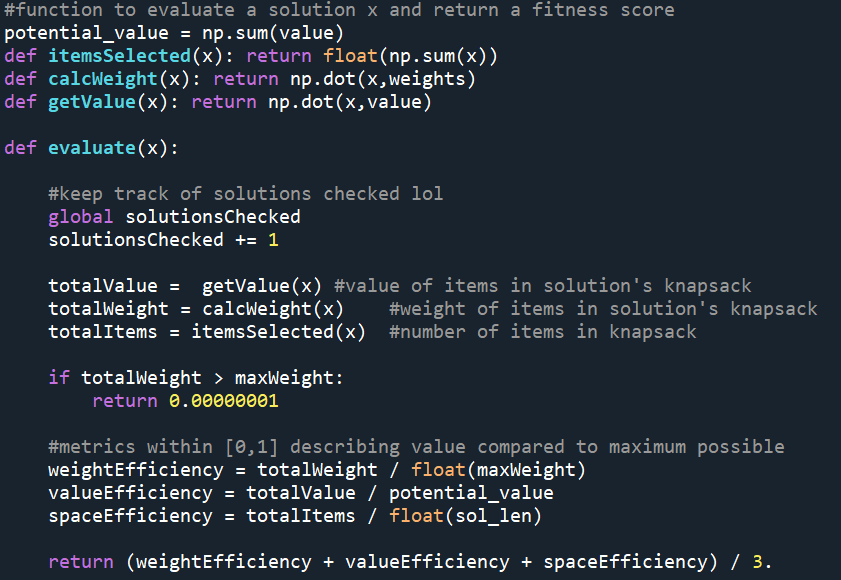
We should first cover the process of initializing a population of solutions which is performed by the following function:



The population is defined as an *nxm* numpy array, where *n* is the total size of the population (a hyperparameter) and *m* is the length of the solution (total number of items to choose from). Creating a single chromosome is done with:

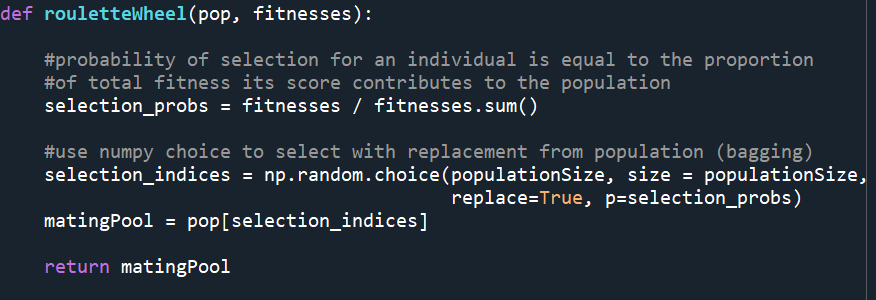


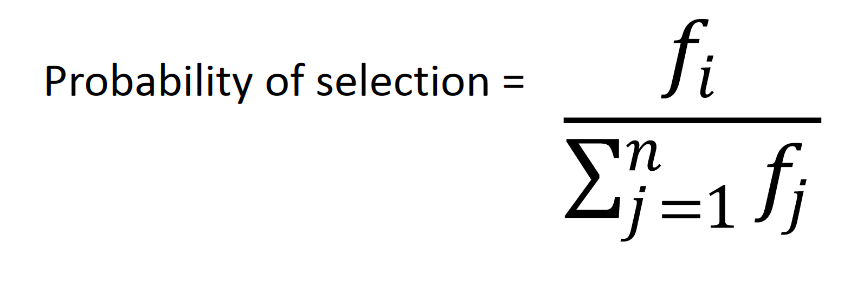
This simply creates a randomly sampled binary array with a bias towards an emptier knapsack. The .85 can be interpreted as “each entry in the array has an 85% chance of being 0.” This function is then run *n* times to generate *n* distinct initial solution guesses. Each of these solutions is then evaluated by a fitness function (shown below). After each solution in the initialized population has been scored, they are sorted in order of descending fitness value for ease of access.



This function returns the average of three solution characteristics: weight efficiency, value efficiency, and space efficiency. Weight efficiency is calculated by dividing the total weight of items in the knapsack by the maximum possible value (in this case, 2500). Value efficiency is calculated by dividing the value of items in the knapsack by the total value of every possible item. Space efficiency is calculated by dividing the number of items in the knapsack by the total number of possible items. If the solution is infeasible, then return a very small fitness score. Theoretically these fitness scores should range from 0 to 1, but the upper bound tends to plateau around .5 in practice.

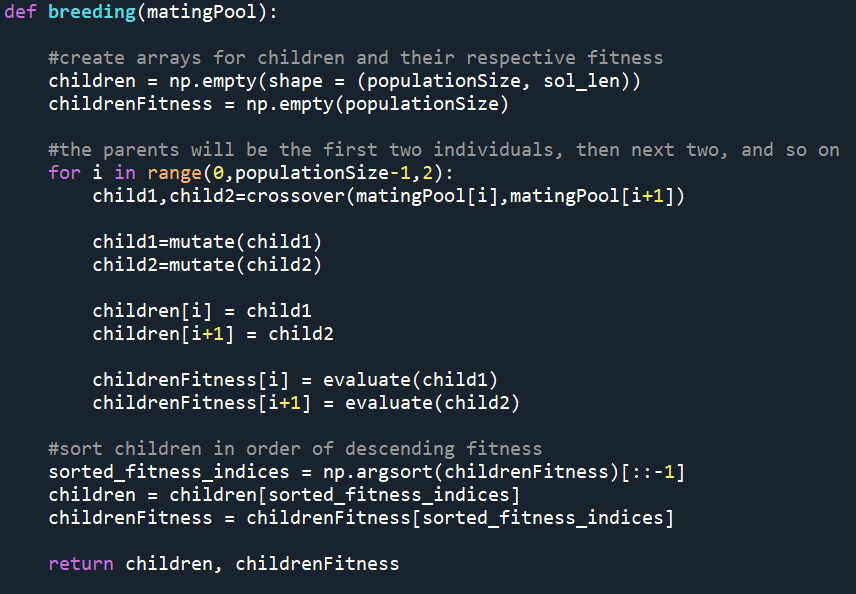
Once these population has been scored in fitness, the mating pool can then be selected as follows:



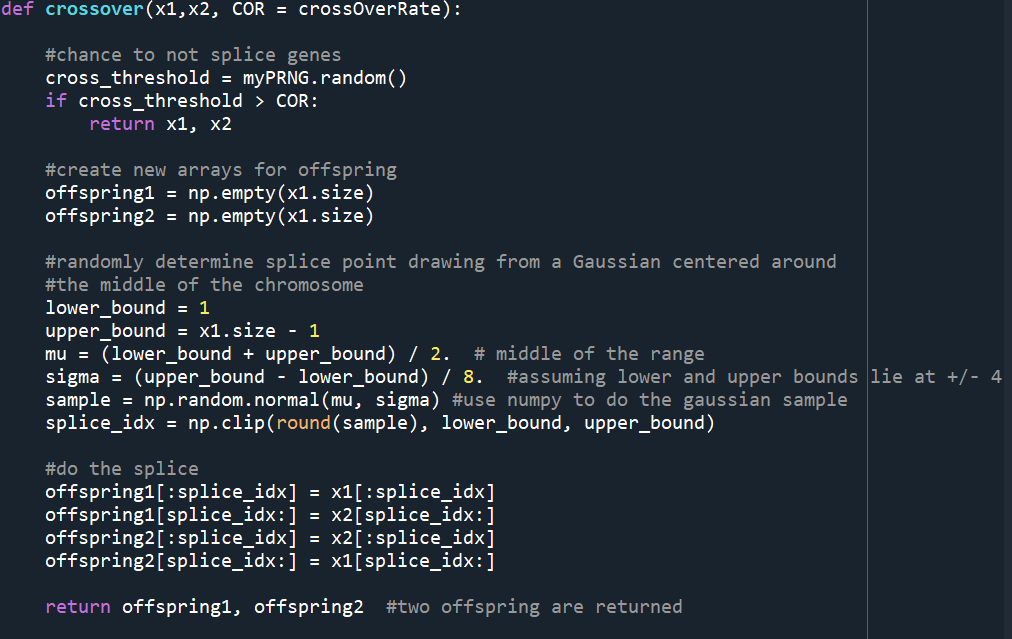
Given the population of solutions and their fitness scores, this function selects with replacement from population subject to a probability density function defined by the proportion of total fitness score explained by a given solution. That is:

This mating pool is equal in size to the original population just boostrapped using the probability selection above.

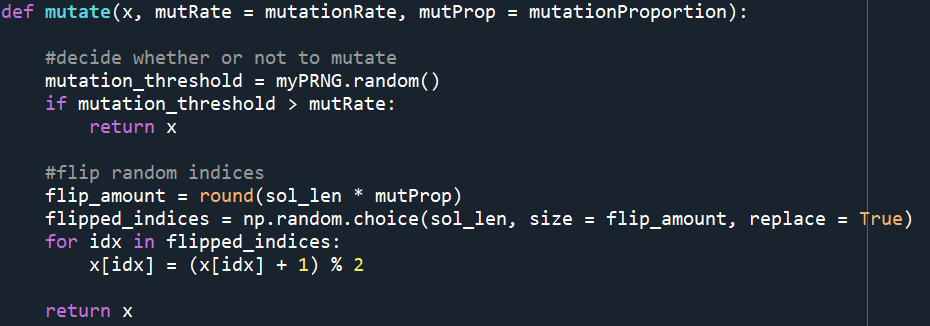
From this mating pool, the algorithm then breeds as follows:



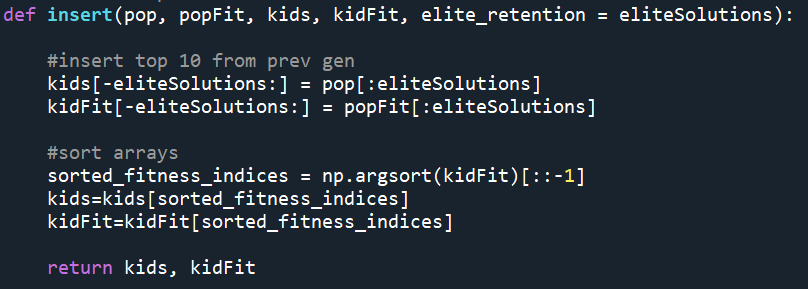
Each consecutive pair of solutions is bred. During the breeding, they are subjected to a “crossover” to produce two offspring. This is performed by:



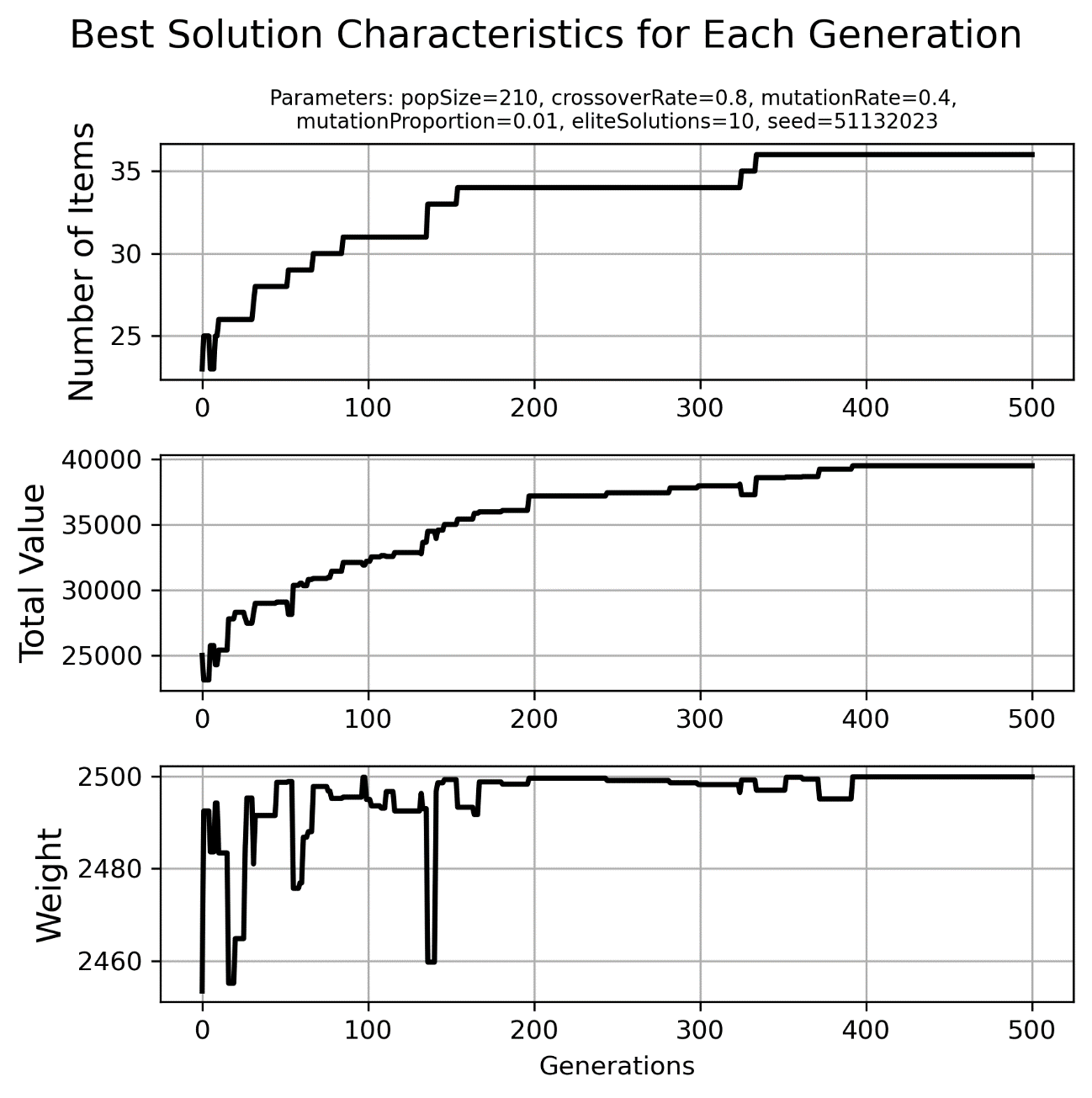
The crossover rate determines the likelihood that the two parents will be spliced. Otherwise, the two parents are themselves returned as the offspring. The splice is done by selecting a pivot index. Each parent array is partitioned into two parts on either side of the pivot index, and the first partition of each array are swapped to make the two offspring. These offspring are then given the chance to mutate:



These mutations happen with a likelihood given by the mutation rate. If mutation doesn’t happen, the solution is simply returned unperturbed. Otherwise, *k* bits are flipped in the binary solution where *k* is given by the rounded value of the mutation proportion multiplied by the solution length.

Once a new generation of child solutions is created, they are evaluated for fitness and sorted. Then an elitism policy is implemented to preserve a set amount of the best of the previous generation. The amount kept is given by the elite\_retention argument below. 

This new population is then sorted to maintain the descending order of fitness, completing the new generation. Each generation should be a group of perturbations on the strongest members from the previous generation.



As we can see from the above example, the genetic algorithm is able to backtrack, removing items from the knapsack. The weight is the most stable parameter, with the knapsack staying almost full the entire time. The rate of improvement drops off as diversity of the population diminishes. More generally, we can see that the genetic algorithm produces much better results than a hill-climbing algorithm as seen in the table below.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Population Size** | **Generations** | **Crossover Rate** | **Mutation Rate** | **Mutation Proportion** | **Eliteism** | **Weight** | **Value** | **# Items** |
| 120 | 500 | 0.8 | 0.4 | 0.01 | 10 | 2492.6 | 38242.7 | 37 |
| 150 | 500 | 0.1 | 0.4 | 0.01 | 10 | 2497.9 | 37713 | 36 |
| 150 | 500 | 0.25 | 0.4 | 0.01 | 10 | 2497.4 | 39141.8 | 37 |
| 150 | 500 | 0.2 | 0.4 | 0.01 | 10 | 2499.9 | 39520.7 | 36 |
| 150 | 500 | 0.5 | 0.1 | 0.01 | 10 | 2498.9 | 37188.6 | 32 |
| 150 | 500 | 0.5 | 0.4 | 0.01 | 100 | 2492 | 34645 | 30 |
| 150 | 500 | 0.5 | 0.4 | 0.01 | 10 | 2492.8 | 38482.9 | 38 |
| 150 | 500 | 0.5 | 0.4 | 0.01 | 50 | 2499.7 | 38221 | 36 |
| 150 | 500 | 0.5 | 0.4 | 0.05 | 10 | 2498.6 | 35163.7 | 33 |
| 150 | 500 | 0.5 | 0.4 | 0.1 | 10 | 2481.3 | 26242.3 | 27 |
| 150 | 500 | 0.5 | 0.4 | 0.2 | 10 | 2494 | 26672 | 24 |
| 150 | 500 | 0.5 | 0.4 | 0.3 | 10 | 2491.3 | 24758.7 | 24 |
| 150 | 500 | 0.5 | 0.4 | 0.4 | 10 | 2497.3 | 26523.3 | 24 |
| 150 | 500 | 0.7 | 0.4 | 0.01 | 10 | 2496.8 | 38083.2 | 37 |
| 150 | 500 | 0.8 | 0.4 | 0.01 | 10 | 2499.1 | 38983.2 | 38 |
| 150 | 500 | 0.9 | 0.4 | 0.01 | 10 | 2498.2 | 38715.6 | 37 |
| 200 | 500 | 0.8 | 0.4 | 0.01 | 10 | 2497.6 | 37898.7 | 37 |
| 210 | 500 | 0.8 | 0.4 | 0.01 | 10 | 2497.8 | 38199.6 | 37 |

**Table 2**: Results for various hyperparameter turnings for genetic algorithm.