Matthew Daniel

CS 415

February 9, 2017

Sub Project 1a Write-Up

**Summary:**

To alter the genetic algorithm from class, I plan to use a different fitness function from the list provided. The Schwefel function that was briefly discussed in class seems promising. Additionally, I will vary mutation rates while holding the population size and number of iterations constant. I also want to increase the complexity of the crossover function. This may look something like picking random (or predetermined) sections of the DNA to exchange between the organisms passed into the crossover function.

An example graph of the data may look something like:

Matthew Daniel

CS 415

February 28, 2017

Final Project 1 Write-Up

**Abstract:**

I modified the genetic algorithm from class to measure a creature’s fitness using the Scwefel Function, to vary mutation rates, and to vary crossover. Using the same randomly generated population of size 20 and trait length of 10, I calculated the average fitness of the population after 40 iterations of the algorithm. I did this for every mutation rate/crossover combination, then compared how the combination ranked against the other mutation rates and crossover combinations. I ran 10 such trials, then took the average of the ranks for each of the variables as a measure of how well each mutation rate and crossover pattern created fit populations.

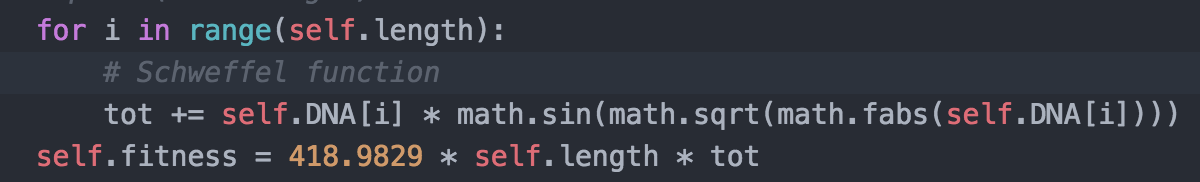
**Algorithm Descriptions:**

**Population initialization:**

I initialized the 10 trait values to a random float between (-10,10) for each of the 20 creatures in the population (same as class algorithm). I stored this initial population outside of the population class so that I could use the same randomly generated population values for each of the mutation rate/crossover combinations.

**Measure of fitness:**

I used the Scwefel Function from the list of provided fitness functions to measure the fitness of a creature. I calculated the fitness using the python math library:



**Mutation rates:**

I selected mutation rates that I thought were reasonable and covered a wide range. These rates were: 0.0, 0.005, 0.01, 0.02, 0.03, 0.05, 0.1, 0.15, 0.2, 0.3. Mutation occurs using the same function as the original in-class algorithm, varying a trait ±2 if a randomly generated number between 0 and 100 was less than the mutation rate times 100.

**Crossover functions:**

I experimented with various crossover functions. These are not intended to be biologically accurate. I have outlined my choices in the table below. This is intended to be used as a key when reading the crossover functions from the results.

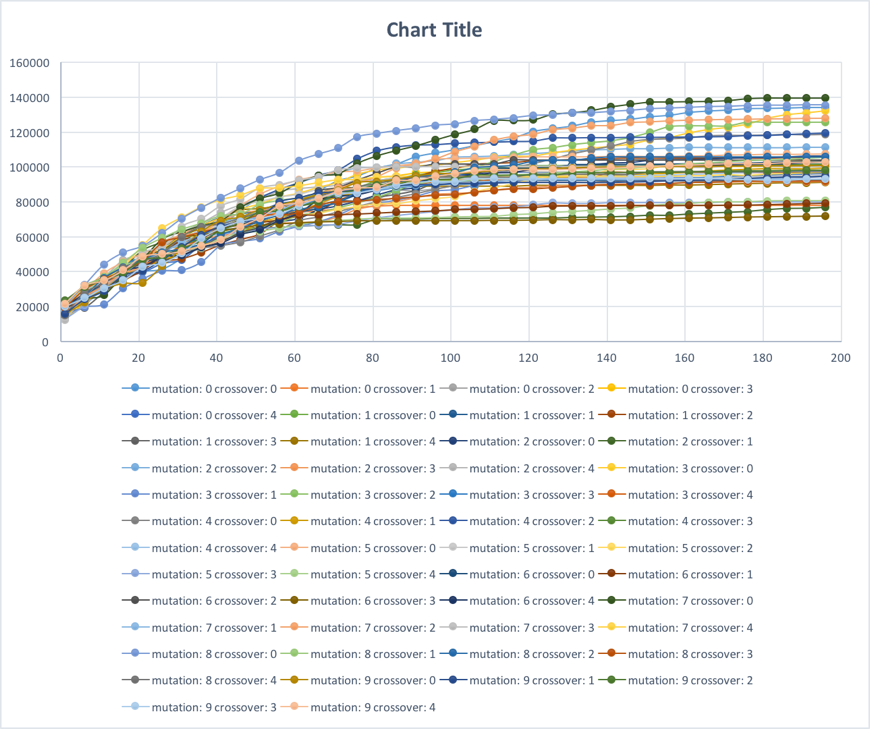
|  |  |
| --- | --- |
| Crossover function number | Description |
| 0 | The crossover function from class. It randomly exchanges values at index i between the two organisms with probability p = 0.5. |
| 1 | Crossover function that switches the first half of the first organism with the first half of the second organism. |
| 2 | Picks a random starting index less than the number of traits and a random ending index greater than the starting index and less than the number of traits. Swaps between these two indices. |
| 3 | Swaps every other (second) value. |
| 4 | Swaps every fifth value. |

**Main program:**

The main program consists of nested for-loops, running a modified version of the in-class iteration steps over each (mutation number, crossover function) pair. The iterate step itself performs 40 iterations, as this seemed to be where the majority of the fitness measurements had plateaued.

**Writing/plotting results:**

I used the python csv library to export the mean of each generation’s fitness values to an array and eventually to a .csv file. The mean was found using the numpy.mean() function. I prepended a label with the current mutation rate and crossover value to the beginning of each population trial. Below, I show an example of a column of data and a plot created in Excel.

Graphs/csv%20example.pdf

The Excel chart was difficult to decipher any meaningful data from, as you had to manually set the series color, marker shape, etc. for each series. I imported matplotlib, and was able to display the results with more control. I experimented with two mappings, one which mapped line color to the crossover value and line transparency (alpha) to mutation function number (see figure 1-1, 1-2 below). The other mapping did the opposite (figure 2-1, 2-2). It was still difficult to draw conclusions from these graphs with so much visual information being presented, so I decided upon another metric by which to gauge the performance of each of the trials.

Back in Excel, I calculated the mean fitness of the population after each population had gone through its iterations. I then ranked these values, comparing them both to other mutation rates with the crossover function held constant, and other crossover functions with the mutation rate held constant. I did this for 10 trials, and took the mean of the ranks, so that the pairs would be tested for different random populations. This made it possible to draw some rough conclusions about how each crossover function and mutation rate performed, as far as maximizing population fitness.

**Results:**

**Matplotlib:**

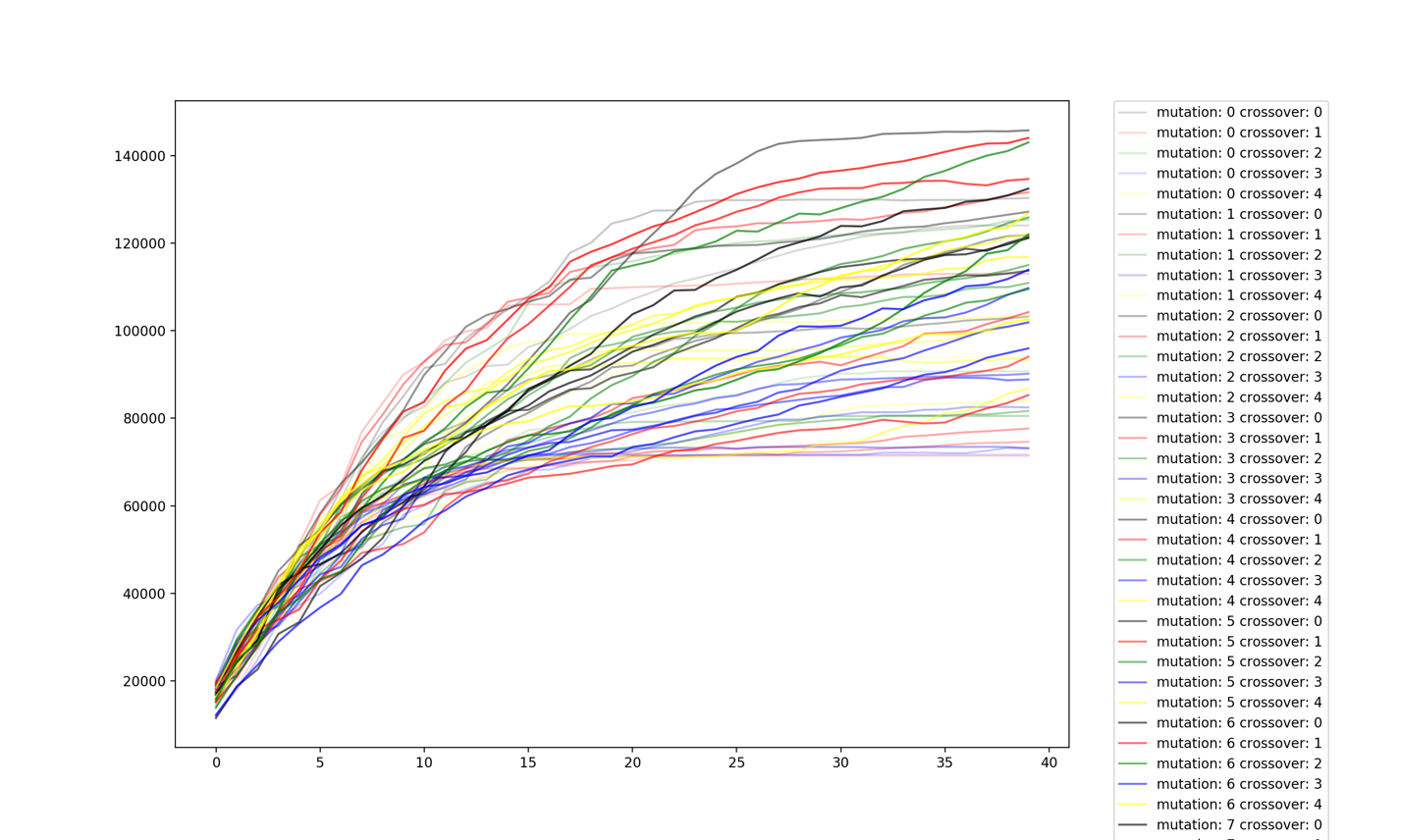
****

Figure 1-1: Map color to crossover, alpha (lightness) to mutation number

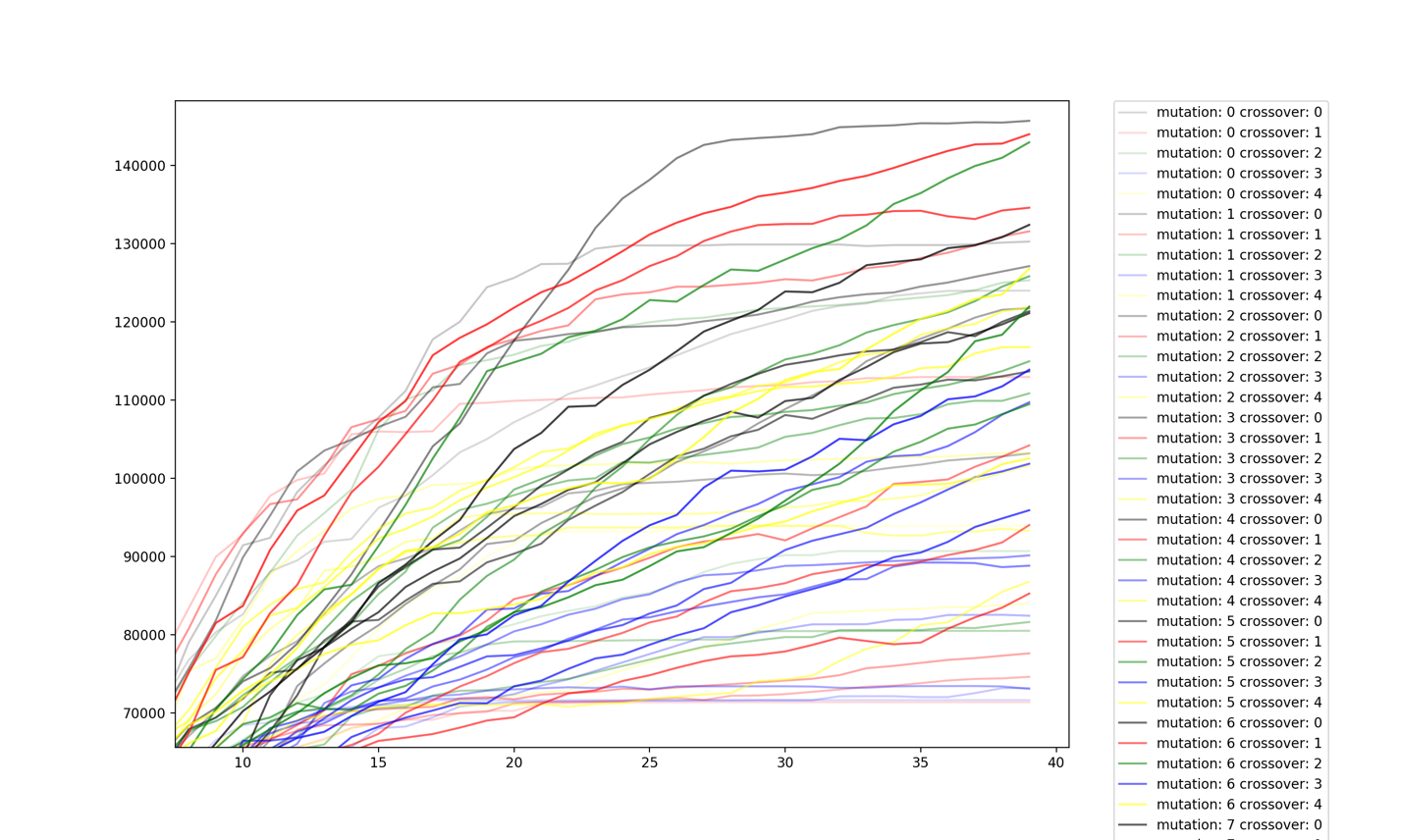


Figure 1-2: Same as previous with window focused on higher (x,y) values

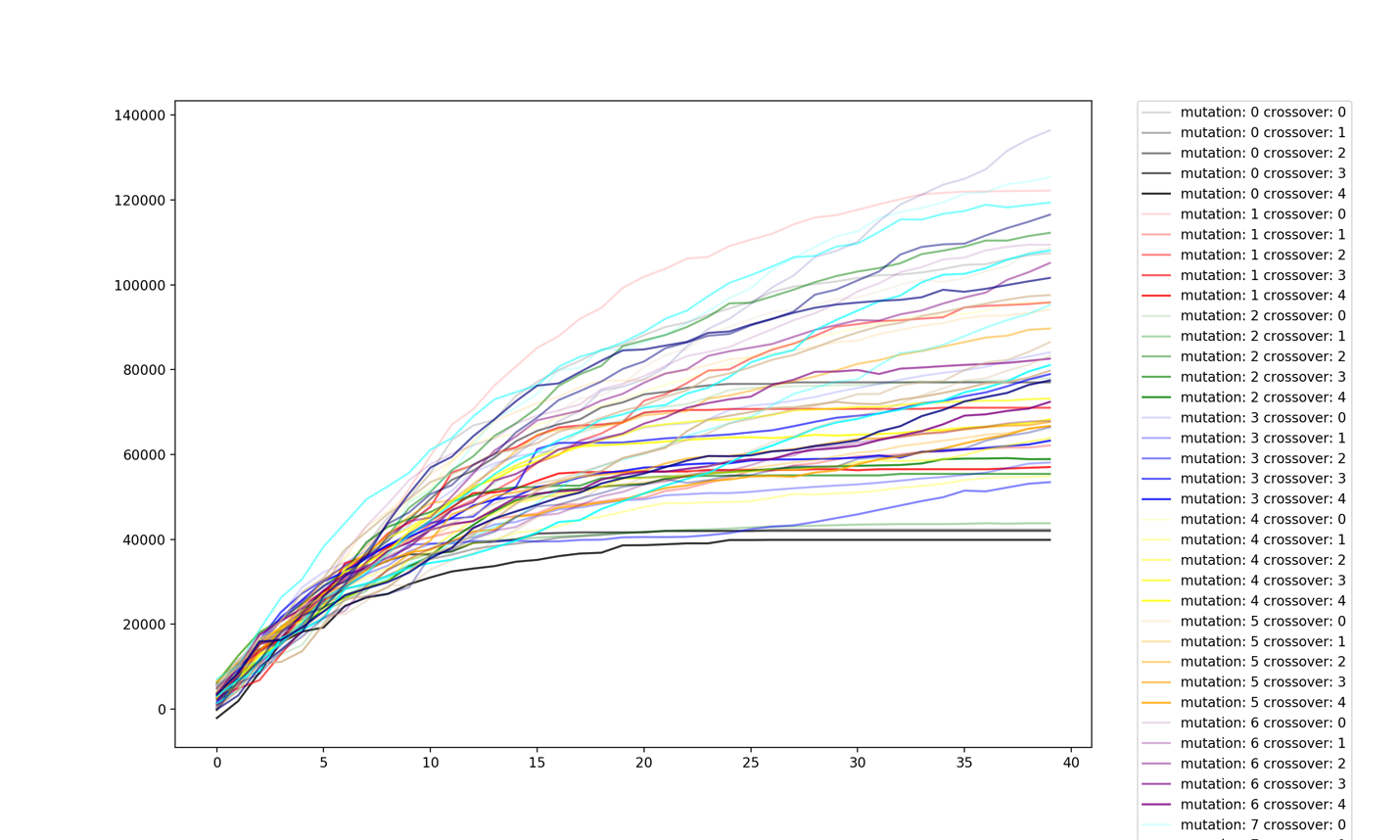


Figure 2-1: Map color to mutation number, alpha (lightness) to crossover

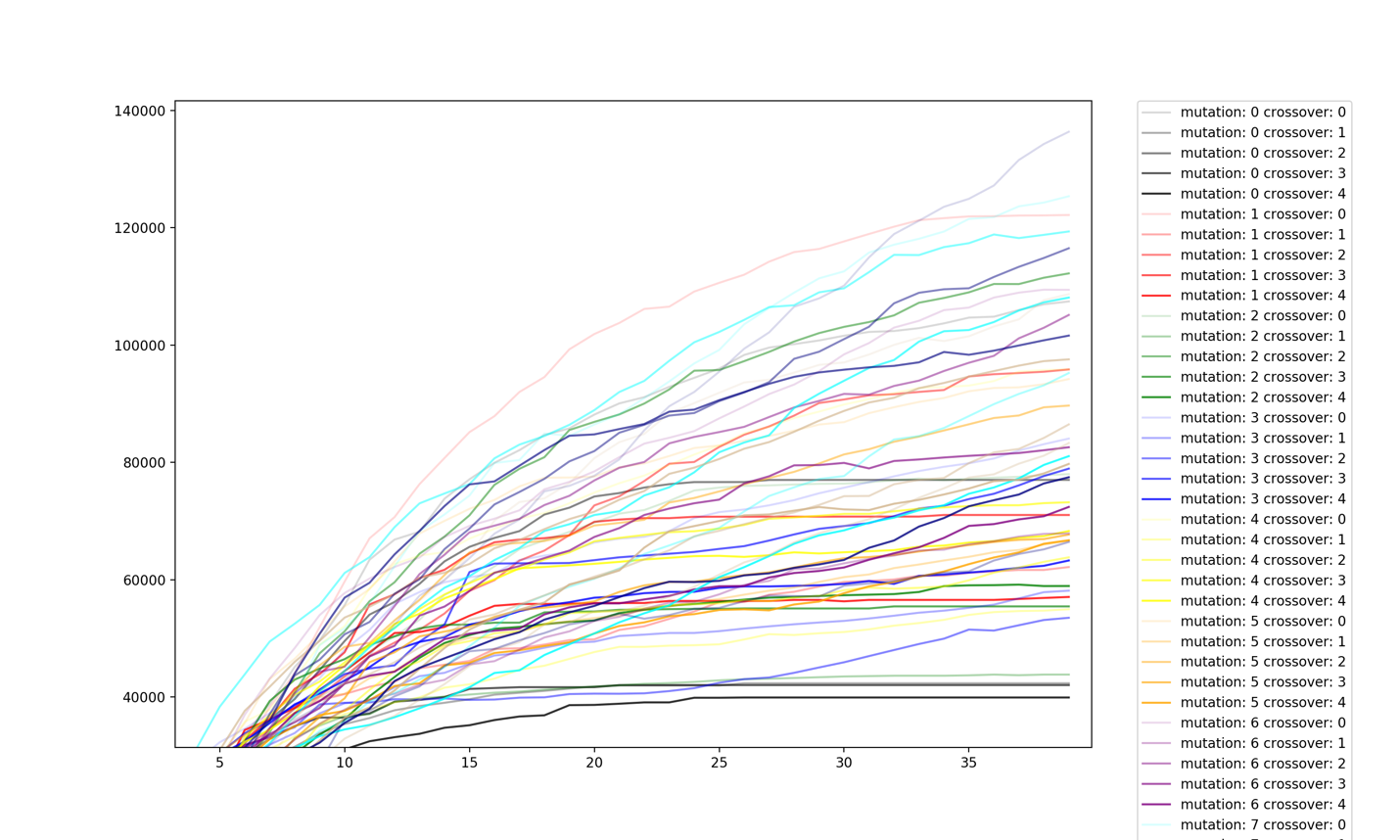


Figure 2-2: Same as previous with window focused on higher (x,y) values

**Excel:**

**Final%20Results%20-%20crossover.pdf**

Figure 3: Comparison of fitness based upon crossover function after final population. Rank aggregated after 10 trials

**Final%20Results%20-%20MutRate.pdf**

Figure 4: Comparison of fitness based upon mutation rate after final population. Rank aggregated after 10 trials

Final%20Results%20-%20Trial%20Example.pdf

Figure 5: Calculations for one trial (trial 10), comparing mutation rates, crossover functions against each other

**Conclusions:**

Though there is a lot of data to visualize here, we can draw some conclusions from the matplotlib graphs, and some more in depth conclusions from the Excel charts.

**Matplotlib:**

We will refer to figures 1-2 and 2-2, because they paint a slightly clearer picture of the fitness values we care about (at the end of the iterations).

From figure 1-2, we can see that the red, black, and green lines seem to gather at higher y-values. Red corresponds to crossover function 1, black to crossover function 0, and green to crossover function 2. We can see that these might be the best-performing for this particular trial, while yellow (crossover 4) and blue (crossover 3) seem to generally fall further down the y-axis. The alpha-values of the lines seem to be mixed generally, so it is hard to view much information about the mutation’s effects.

In figure 2-2 color tells us very little. Instead, we can make out a slight pattern of lines increasing in darkness moving down the y-axis. The lighter values correspond to lower crossover values, which would seem to confirm speculations made in looking at figure 1-2. It is difficult to tell anything about the mutation rates from these plots though.

**Excel:**

In figure 3, we can hope to learn a bit more about which crossover functions yielded the most fit final populations. Green colors/low numbers represent the best performers, and red colors/high numbers the worst. The original crossover function (function 0) from class ranked first in each trial, making it the clear top-performer. Swapping a random sub-section of traits (function 2) performed second-best. The other three functions had relatively similar average rank. This generally mirrors the results from matplotlib, with the exception of function 1 underperforming a bit perhaps.

From figure 4, we are finally able to tell something about how mutation rates affected the final population fitness. The 0% mutation rate ranks the worst on average. We get better and better results as we increase the mutation rate, seemingly peaking in the 15%-20% range. It would have been interesting to go beyond 30% to ensure that the maximum fitness values were indeed around 15%-20%.

Figure 5 simply shows the ranking for each individual trial (i.e. a new random population that the algorithm ran on). The shaded results were copied over to form one trial for the final results.

**Potential problems with method:**

Visualizing a variable on a color/alpha scale would intuitively work better for mutation rate, which is qualitative, rather than crossover, which does not fall on a numeric scale. To my surprise, it was easier to tell which crossover functions performed better than mutation rates from the matplotlib graphs. This may be due to there being too many mutation rates to visualize easily from the graph.

There are a few potential problems with the Excel visualization. The first may be that final population average fitness may not be a good measure of algorithm performance. I thought about measuring something else, such as the logarithmic function generated by the scatterplots of each of the series, but thought that the final average fitness value made the most sense. Secondly, assigning a simple rank to the crossover functions/mutation rates may not be a good measure of how the algorithm actually does. This tells you nothing about how close together the compared final population finesses were. Averaging these ranks also ignores the concept that, for some random populations some mutation rate or crossover function might do really well, but for others it may do poorly. Third, my results don’t say anything about which mutation rate/crossover function is the best general purpose way to go with this fitness function. Finally, these results do not find the best combination of crossover function and mutation rate. The data for the best pair is potentially lost as all of the ranking and averaging is occurring.