Genetic Algorithm Write-up
Tayler Tolman

I'll begin with a general overview of my algorithm followed by the experiments I did with the fitness function, initial population, crossover, and unique checks. For reference I chose the code breaking problem.

My generic algorithm is as follows: Step one establishes an initial population. This population will have random strings and fill a vector hereto referred to as popVector. Step two terminate all strings in the popVector if the fitness function on the string is a below average score.Step three, perform crossovers on random string pairs up to the original amount of strings in the vector. Step four mutates a percent of the children from the crossover pairs. Step five terminate again. Step six check for the correct answer then return to step three.
NOTE: Unless otherwise indicated the initial population has size of 1000 and code length is size of 11

Starting with my experiments of the fitness function. The first function I came up with was the number of characters that were in the correct position. This seems to be the best to me. However I also came up with a function that checks the amount of characters correct in the string without regards as to the position. Obviously this will lead to strings that contain all the right characters but improper order and only though even more mutations and crossover could the order even get better. Now this one could be improved greatly by checking if we have all the right amount of characters and then switching from mutation to only crossover.

	# Char correct	# Char in correct places
Generations taken	289	23
Time Taken	109.8 seconds	4.13 Seconds

Now I'll take a look at the initial distribution of the problem including size of the code and initial size of my population.

	Code size 5	Code size 11	Code size 25	Initial pop 100	Initial pop 1,000	Initial pop 10,000
Generations taken	11	25	51	129	26	16
Time Taken	1.25 seconds	2.88 Seconds	11.91	1.02	3.73	238.16

Looking at the size of the initial code, the longer the code the harder it is to crack. That makes perfect and logical sense. The size seems to take exponentially longer to solve the longer it is which also makes sense. The initial population though was quite confusing for me at first. Surely a larger population means a faster algorithm right? WRONG. Well wrong in this relatively simple problem I suppose. The data shows that smaller populations perform much better than larger ones. I suppose this is because smaller populations spend much more time mutating and crossing over than a larger population does. Whereas a larger initial will take less generations because it's randomness may include a more correct answer, the smaller population will take shorter time because it evolves much faster.

Now let's look at the crossover event. My initial crossover event simply took every other character from two strings and combined them into one. This was extremely simple and I suppose somewhat how nature works. My second crossover function nicknamed smart crossover utilizes the fitness function to help it decide if a crossover is good or not. These dramatically reduced the amount of crossovers. Data is as follows

	Old crossover	Smart crossover
Generations taken	21	22
Time Taken	10.82 seconds	2.78 Seconds

As you can see the generations needed is relatively the same but the smart crossover dramatically reduces the time. This is because crossovers that do not improve based on the fitness function are not added to the population. This is one of the biggest time saves I found that helped regardless of code size or initial population.

Next up we will take a look at the termination function which will trim up the population based on the fitness function. My initial thought is that we are looking for better subjects so of course we should only take subjects that are among the best The resulting data with survival percent I tweaked are as follows.

	50%< Of pop	55%< Of pop	65% < Of pop
Generations taken	21	ERROR	ERROR
Time Taken	3.89 seconds	ERROR	ERROR

Oh no! What happened? Well as it turns out because my fitness function returns an int and the comparison using a float we can have a situation where every string in the population has the same fitness score. If this happens then the algorithm can't find something better than the average, because they are all the average! But wait a second what if I simply focus on the highest scores instead of finding entities with an above average score? Data is as follows

	Old way averages	New way max
Generations taken	21	135
Time Taken	3.89 seconds	1.288 seconds

Holy guacamole look at that increase in speed! In fact I actually have a print statement giving me stats on each generation if I took that out and compared them...

	Old way averages	New way max
Generations taken	24	153
Time Taken	3.88 seconds	.6 seconds

Incredible! Now it does take way more generations but that's only because we are cutting out the strings which are not one of the best fits, leading to a smaller population and therefore less

randomness then on a larger population. Well looking for a max seems clearly better. At first I thought that was only because this was a hill-climbing problem and the solution path is somewhat linear but mutation actually allows some randomness which defeats the main obstacle to a pure hill-climbing algorithm.

Now let's take a look at the mutations. The first mutation I used an amount of random strings from the population and changes one character, adding the new string to the population. First I'll see what happens if two letters are changes instead. And then what happens if I modify the function to only mutate if it results in a better string. Results are as follows:

	One character mutation	Two character mutation	Smart mutation
Generations taken	150	291	124
Time Taken	.70 seconds	.89 seconds	.445 seconds

Mutating two characters instead of one increased the time and generation. No surprise there, increasing randomness will do that if the randomness is not controlled. The smart mutation however decreased the generations and time taken. This also makes sense since I'm only adding the mutation if it beats or is equal to the current maximum fitness score. Selecting mutations that only beat the current sounds good at first but the crossover is really the part that is optimal at finding that otherwise you dramatically increase the amount of mutations needed to find a correct answer. Example:

	Only accept better
Generations taken	1337
Time Taken	.60 seconds

Oddly enough this doesn't seem to take any more time. I suppose because many less strings are added to the population this way. Is this optimal on a larger problem? Probably not.

Conclusion:

The best way to utilize a genetic algorithm is to maximum the evolution potential of each entity in the population. Having an extremely large initial population will let you solve the problem in fewer generations but only because of the increased sample size and randomness. IT is much better to have a smaller population with more clues as to where to go than to try to brute force a solution. Furthermore progress is improved if the mutations and crossovers seek the best or at least equal solutions to the current best solution (mutations may need to be truly random however to escape hill-climbing plateau). A termination function is also necessary to ensure our population does not get too large to handle. All of these combine to form an algorithm that is both quick and frugal with its memory.