Code Breaking With Genetic Algorithms

written by Travis Smith and Chris Terry

Abstract

Our goal was to learn more about Genetic Algorithms. This paper will cover some details of how we implemented those algorithms, as well as the experiments we did with our different algorithms to see how they compared and how they could be made better.

Introduction

This project, along with its subsequent write up and report, were aimed at helping us gain an understanding of and appreciation for Genetic Algorithms(GA). While some of the details of how we implemented our algorithms are provided, the main focus of the write up is on our experimentation with these algorithms. We also cover some things that we wish we'd been able to do, but ran out of time to actually attempt.

For our program, we chose to simulate a code breaker. The general premise is that we had a target string which we wanted our GA produced strings to converge on. In a real world scenario we'd need some form of feedback about how close we were to the actual target code, such as a light when a certain character was entered correctly. For the ease of our implementation, the algorithm knew the original code and could compare each randomly generated string against it.

The following section will describe the GAs we wrote and used for the experiments. After that we report the details and results of our experiments. Including three similar experiments about the size of the population selected for reproduction as related to the three different selection criteria. The other two experiments compare the mutation rate, and the difference in the two GAs that have been developed.

The GAs were written by Chris Terry, and Chris and Travis each did experiments. The paper was co-written with Travis writing the first three experiments and Chris writing that last two.

Algorithm

There are two different algorithms used for these experiments. Both algorithms use a Single Point Crossover, as described in a slide deck provided by Todd Peterson[1]. Other crossover algorithms could have been used, but a Single Point Crossover seemed sufficient for the job without adding additional overhead to the development. One algorithm takes the target string and uses its length to generate a population of random strings of that length. This population is then measured against the target using the fitness function, the individuals that are selected are then mutated, with a small selection undergoing crossover. This corresponds to asexual and sexual reproduction. The mutated individual is kept in the population as well as its mutation, while the crossover selects a random crossover point and returns the four possible combinations of the two individuals. For example if the individuals are {ABC, abc} and the crossover point divides them after the first character, the results would be {Abc, aBC, bcA, BCa}. The mutation selects a random character from the string and changes it to either the character before or after it.

The other algorithm used differs in a few important ways. The population generated consists of various length binary strings. The binary string is then used to generate a character string. The character string generated is what the fitness test 'grades'. The fitness test grades the strings based on length and content. The closer they are to the target the better their grade is. A selection of the population is made which will breed. The breeding is random among the selection, and consists of a crossover similar to the other algorithm, except the crossover point is limited to the length of the shorter of the two strings as the length is not predetermined.

Experiments

We ran experiments on the first algorithm with three types of selectors, Tournament Selection, Ranked Selection, and Fitness Proportionate. We tested each selector with varying amounts of selections before the crossover and mutates after the crossover. These mutations/selections were kept the same for each run, and were limited to 1, 3, and 5 mutations/selections per selector. So for example you could have 3 selections, 1 crossover, and then 3 more mutations, but the selections and mutations would always happen the same amount of times. The default parameters for the experiments were population size of 1000 and a crossover rate of 0.05%. To account for random fluctuation in each run, we ran each experiment ten times.

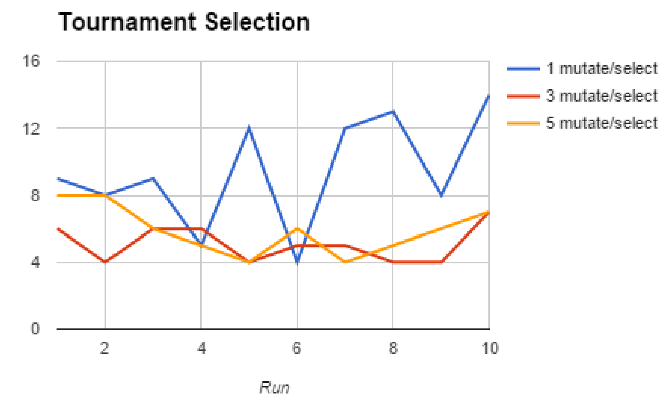
The other experiments we ran isolated the second GA. The default values used were population size of 1000 and mutation rate of 0.05. The first experiment was conducted by changing the mutation rate and then running the simulation 50 times and averaging the result. The last experiment varied the crossover strategy.

Tournament Selection

This was the first selector that we coded, and the first one to be run through the gauntlet. It created a sort of baseline for us to judge the rest of our selections.

Tournament Selection			
Run	1 mutate/select	3 mutate/select	5 mutate/select
1	9	6	8
2	8	4	8
3	9	6	6
4	5	6	5
5	12	4	4
6	4	5	6
7	12	5	4
8	13	4	5
9	8	4	6
10	14	7	7
avg.	9.4	5.1	5.9

The first thing that stood out upon looking at the data was how much worse the selection process was when it was only run one time per crossover. It averaged almost twice as many iterations to complete as the runs with multiple selections before the crossover.



It's worth pointing out that increasing the number of selections/mutations to five per crossover was on average almost an entire iteration longer. This could be due to relatively small sample size, or it could be a sign that the algorithm starts to fall apart when too many members of the population get removed before a crossover. In either case, the difference is far more visible on the graph than it is on the table.

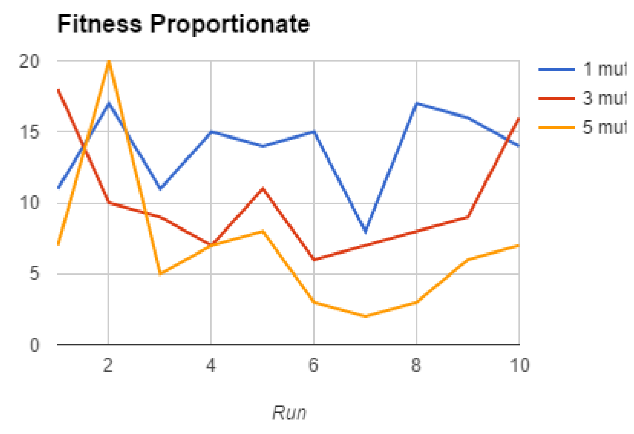
Fitness Proportionate

Since a fitness function was needed to construct the Tournament Selection, it made sense to build the Fitness Proportionate next. This selector relies on a weighted random selection, based on the fitness value of each candidate. The method we implemented created several thousands of copies of candidates in a list as part of the weighted random process, which resulted in the algorithm taking longer to run than it probably should have. None the less, we were able to gather data from it.

	Fitness Proportionate		
Run	1 mutate/select	3 mutate/select	5 mutate/select
1	11	18	7
2	17	10	20
3	11	9	5
4	15	7	7
5	14	11	8
6	15	6	3

7	8	7	2
8	17	8	3
9	16	9	6
10	14	16	7
avg.	13.8	10.1	6.8

The first thing we noticed is how many more iterations it took compared to the Tournament selection, averaging 13.8 iterations with a single selector and mutator per crossover compared to the 9.4 of Tournament. Combined with the longer run time of those iterations, the Fitness Proportionate did not seem to be a good fit for our problem.



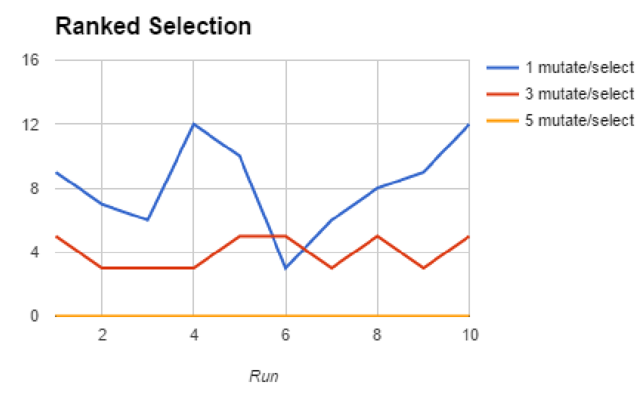
One thing that was interesting about Fitness Proportionate though was that it continued to get better as more selects were done before the crossover, unlike Tournament Selection. While there was the anomalous run 2 that took 20 iterations to complete, the rest of the runs took dramatically fewer iterations. With run 2 included, the average was 6.8 iterations, but removing it as an anomaly results in the average dropping down to 4.8 iterations to find the target, faster than anything Tournament Selection was able to do. If the implementation itself were to be cleaned up more, it could be a very viable solution.

Ranked Selection

Ranked Selection was the most interesting of the selectors to test. It had many features similar to Fitness Proportionate in its implementation, but was dramatically smaller in the size of its weighted list. This lead to it running faster, but the algorithm itself also seemed pretty good.

	Ranked Select		
Run	1 mutate/select	3 mutate/select	5 mutate/select
1	9	5	0
2	7	3	0
3	6	3	0
4	12	3	0
5	10	5	0
6	3	5	0
7	6	3	0
8	8	5	0
9	9	3	0
10	12	5	0
avg.	8.2	4	0

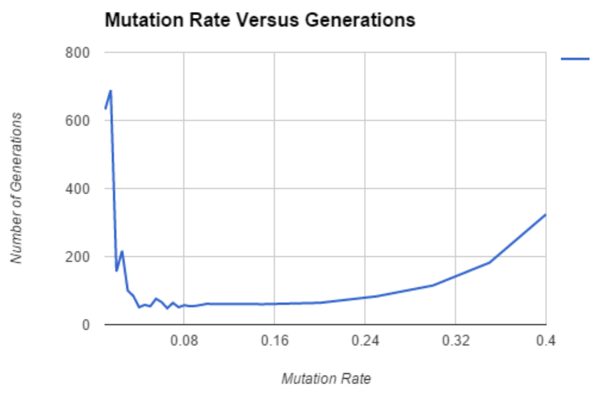
The first thing that we noticed was that the single selection/mutation per crossover was the most efficient out of all of the implementations. The rest of the results were a bit different than what meets the eye. With 3 mutations/selections per iteration, the algorithm either completed in 5 iterations, 3 iterations, or it hung. There was no middle ground. Needless to say, hanging's not ideal for an algorithm. The problem only got worse when we bumped it up to 5 mutations/selections per iteration. It hung every single time, resulting in no data for that experiment.



In talking over what could have caused this odd behavior, we came to the conclusion that the hanging was probably caused by the dramatically decreased population from each selection. Each time you select without making a crossover, you're more likely to end up with a list of candidates that all have the same rank. In our case, the rank was determined by the number of correct letters, as well as the number of correct letters in the correct position. If a population was constructed where every member had the same weight, then the effectiveness of the algorithm drops off dramatically.

Mutation Rate and the Number of Generations to Completion

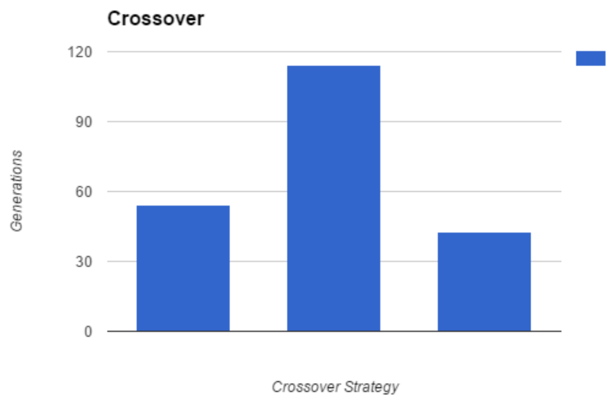
For this experiment the mutation rate was varied while all other parameters were held constant. The mutation rate was started at 1% and increased at 0.5% intervals to 10% then at 5% intervals to 0.4%. The following graph shows the results of the experiment.



As the graph shows there is a rather large sweet spot between about 0.04% and 0.2%. This indicates that for this GA and the current fitness test and crossover this mutation rate can vary without significant impact on the number of generations required to find the solution.

Crossover Strategies

In this experiment the variable was the crossover strategy used. In the base case the crossover strategy is a single point crossover at a random point. The second strategy uses a two point crossover again with the points chosen at random. The third crossover strategy is a random selection of bits from each parent in a 9:1 ratio per parent. The following table displays the results.



As shown in the table the first strategy works relatively well averaging less than 60 generations to find the solution. The two point crossover was less effective requiring twice as many generations on average than the one point crossover. The optimal solution however, appears to be the multipoint random selection crossover.

Conclusion and Future work

The biggest takeaway that we got from this project and the experiments is how much of a difference fine tuning the algorithm can make. In some cases, minor changes make a dramatic improvement. In other cases, minor changes can lead to a great loss in performance. In still others it's a trade off, where it might be faster most of the time but hang the rest.

In the future it'd be nice to re-visit the Fitness Proportionate to see what went wrong with tying the weight to the fitness value. There are several likely spots where things could be improved, but it's in a state where it currently works well enough for demonstration purposes so it's being left as-is.

The Genetic Algorithm is an interesting approach to solving a problem. I have yet to see its great strengths using the problems that we have used so far. But it can be an easy algorithm to implement, and with modern computers its possible to make the billions of calculation necessary in a relatively short time. Other future work to look at includes trying different fitness functions, and changing the population size we didn't really experiment with that much.

References

[1] ML CH 9 Genetic Algorithms.ppt, by Todd S. Peterson