Title

Scott Smith and Brian Herbst

Abstract

The algorithm takes into account several factors, which include, but aren't limited to: pool size, population size, mutation rate, crossover rate, and selection type. We will be running tests on each of these to see which is the fastest. This takes into account the number of generations needed to find the solution, but we feel that the amount of time is a better judge of how good a program is. Each test will be recorded 3 times (cases), but they will be run between 10 and 20 times each, recording a random case, chosen prior to the run. This ensures that data is pure, and data is not chosen with bias.

Introduction

"A genetic algorithm (GA) is a method for solving both constrained and unconstrained optimization problems based on a natural selection process that mimics biological evolution. The algorithm repeatedly modifies a population of individual solutions."

- *Reference 4*
-

We created a genetic lab, where the computer would synthesize genetics to learn and figure out what our solution is. We had a string, which was a solution, and we also had several initial sets of sample data, which the computer would try to set some calculated rules, and follow those until it found the solution. This would also take into account "genetic mutation" which would randomly mutate one or two different letters in the string.

During each generation, all of the population is iterated through, so it will exponentially get slower with each additional generation.

Many times genetic algorithms and machine learning are used for problems, games, and solving other puzzles. One man even tried to use genetic algorithms to figure out the best seating arrangements for his wedding (see reference 2).

One of the most common games that are used to test or represent is the game called Mastermind (reference 3). In this game, the first player chooses 4 pins of any color, and places them in 4 slots. These are hidden from the second player, who then tries to guess the color and location of each pin. The first player only responds with the number of pins that are the correct color and in the wrong location, and also the number of pins which are colored correctly and in the correct position. The second player needs to figure out from the history of guesses the locations and colors of all 4 pins.

We decided to try a few different things to test our program. We wanted to test:

1. Solution length
2. Pool Size
3. Selection Type
4. Population Size
5. Mutation Rate
6. Crossover Rate

We wanted to see which variable caused the biggest change of speed, so we ran 3 cases of each test and grabbed the average speeds, and compared them with the other tests. We will be looking at the fastest times, slowest times, and tests that vary the least and those that vary the most.

We will test 6 different variables in our tests, they will all stay the same throughout all tests, except for the one in which we are changing for our tests. Our default values are as follows:

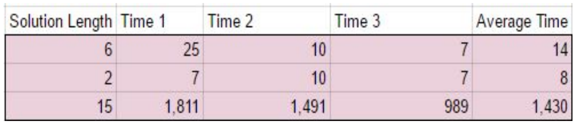
Variable	Default Value
Pool Size	7
Population Size	2
Mutation Rate	90
Crossover Rate	Random
Selection Type	Max-Selection
String Length	7

Our hypothesis was that the selection type would make the biggest change, because Most algorithms are dramatically faster than random selections.

Experiments

Solution Length

3 tests were made, all with the same variables except for the solution length. We were expecting that the longer the solution length was, the longer amount of time it would take in order to figure out the actual solution.

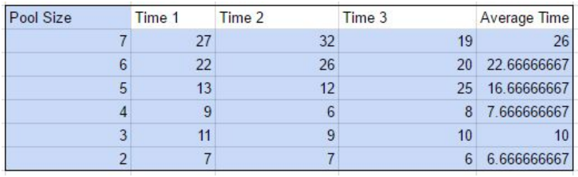


Solution Length	Time 1	Time 2	Time 3	Average Time
6	25	10	7	14
2	7	10	7	8
15	1,811	1,491	989	1,430

We ran the test with a solution length of 2, 6, and 15. We were expecting the one of length 15 to be the slowest, and have the most generations, and we were correct. With a length of 15 we had an average time of 1430 milliseconds (ms), while 6 and 2 had an average time of 14 ms and 8 ms, respectively. There was about a 175 times gap between the slowest average and the highest average, which is really adding about 109 ms per each character you add onto the solution length. We also noticed that this test was normally going about 8 generations into our strings.

Pool Size

Pool size is the number of letters each character could be in our solution. For example, we could have the answer only contain the letters 'a' and 'b', and this would be a pool size of 2. If the user wanted to have a larger pool, they could do all 26 letters of the alphabet, or they could add numbers, letters, uppercase, and symbols, which you could do 50 or 60 or more for your pool size. This can be a smaller or larger number. We decided to try a lot of these, and we were not disappointed. We did realized that it wasn't as big of a difference as the string length though.

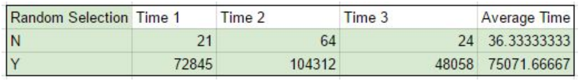


Pool Size	Time 1	Time 2	Time 3	Average Time
7	27	32	19	26
6	22	26	20	22.66666667
5	13	12	25	16.66666667
4	9	6	8	7.666666667
3	11	9	10	10
2	7	7	6	6.666666667

We had the smallest average time of 7 ms while the largest had an average time of 26 ms. This really is only a 19 ms difference, so given that the pool size went from 2 to 7, it really only adds about 2-3 ms per item added in the pool. This didn't seem like it was a very big change for what we thought it was going to do. We were very wrong on this one. We were going between 4 and 10 generations on this one.

Selection Type

Selection type is the way that the computer chooses two children to “breed” together, which produces one offspring. This helps create some randomness throughout. We decided that we could create a second test, which instead of picking the children off of a weighted random scale, we would pick the two children that had the most correct solution. For both of these types of selection a value was given to all the strings in the population, and for random selection, a higher valued string had a higher chance of being chosen as a new parent, while the max selection algorithm grabbed the two which were closest to the solutions.

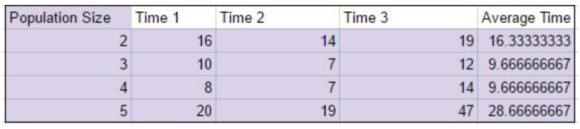


Random Selection	Time 1	Time 2	Time 3	Average Time
N	21	64	24	36.33333333
Y	72845	104312	48058	75071.66667

Selecting these in a non-random way reduced times dramatically, which we expected. Most things are generally faster than random if there is a certain selection weight given. We had our random going well over an averaged 60 seconds while our non-random max selection algorithm was running in at about 36 seconds on average. In our max selection we were normally at 3 or 4 generations, whereas the random selection was normally in generation 12 or 13.

Population Size

The population size is a set of rules that help the program reach a conclusion faster. The more that is in the initial population, the less the program has to create, which in turn makes the program faster. We assumed that the higher the population was, that the faster the program would find the answer. Here are our results:



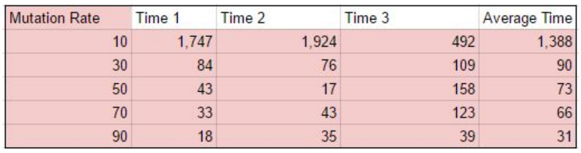
Population Size	Time 1	Time 2	Time 3	Average Time
2	16	14	19	16.33333333
3	10	7	12	9.666666667
4	8	7	14	9.666666667
5	20	19	47	28.66666667

As you can see there wasn’t really a difference in having 3 or 4 in our population, whereas having 2 and 5 in our population actually slowed down the process. It seems that the population probably doesn’t make any significant difference, since the test goes through thousands of strings, having only 2 or 5 would be a very very small percentage of the number that are needed to actually make a significant difference. We ran a couple tests that had closer to 100 in our population and it still didn’t make a significant change. Our generation count was normally at 4 on all the scenarios we ran for this test.

Mutation Rate

The next test was to test the mutation rate. The mutation rate is the chance that a string will have a random mutation, just like genetics. If our mutation rate was 20%, and we had a string “aaaaa”, we would have a 20% chance that our string could randomly mutate one character to a different letter in our pool. If our pool was 2 characters, then if our string mutated then it would become something like “aabaa” or “abaaa” or “aaaab”.

The assumption was made that the fewer times a string was mutated, the faster it would reach a conclusion. This was based on our previous selection test, where adding the random selection slowed down the program significantly. Our times came in and we noticed they didn’t really make a difference on smaller pool sizes, but with larger ones it did make a difference. Our table below illustrates a pool size of only 2, and changing our mutation rate between 10 and 90 percent.



Mutation Rate	Time 1	Time 2	Time 3	Average Time
10	1,747	1,924	492	1,388
30	84	76	109	90
50	43	17	158	73
70	33	43	123	66
90	18	35	39	31

The results show that with only a 10% chance of mutation, the program was sometimes 10 times slower than that which mutated between

30 and 90% of the time. The more mutations, the faster it came to the conclusion.

The generation count on this one was normally at 5, but with the mutation rate of 10% we were getting into generation 7 and once into generation 8, while the third time was generation 4.

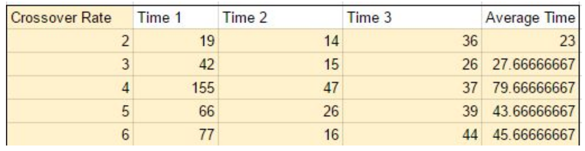
Crossover Rate

"Crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next."

- *Reference 5*

In simple terms: the crossover rate is actually the number of characters that would get split and combine with another string, to make two completely different strings. For example, if we had the strings "aaaaaa" and "bbbbbb" and our crossover rate was 50%, than if crossover happened, it would take 50% of the first string and swap it with the second, making "aaabbb" and "bbbaaa". This is known as crossover and makes two completely new string.

Our hypothesis on crossover rate is that the more crossover that happens, the faster that the program will run. We based this off of the mutation rate results, since the more it happened, the faster the program reached the answer.



Crossover Rate	Time 1	Time 2	Time 3	Average Time
2	19	14	36	23
3	42	15	26	27.66666667
4	155	47	37	79.66666667
5	66	26	39	43.66666667
6	77	16	44	45.66666667

As you can see, the results actually didn't vary much, especially with the exclusion of the one outlier, since the average without the outlier was over 2 times faster than the outlier itself. It seems like crossover rate doesn't really make a difference in how fast our program reaches the answer.

The crossover rate test normally went into the 4th and 5th generations, and it didn't vary as much as we thought it would.

Conclusion and Future Work

Our program was pretty fast at finding the solutions, although we were able to figure out which ones were the fastest, slowest, and which ones made bigger and smaller changes.

Out of the 25 recorded tests, only one was more than 2000 ms in time, and that was our random selection test. All the other tests were under 2 seconds in time and most were between the generations of 4 and 11 for their tests.

The fastest test we had was one of our pool size tests. With a string length of 7 and pool size of 2, we had times of 7 ms, 7ms, and 6 ms to solve the problem. This almost took longer to output the result than figure the result itself.

Our slowest test, as well as the test that made the biggest difference, was our selection test. Anytime you switch from random to a special algorithm, things tend to speed up. Out of our three runs with random selection, we had a slowest time of 104,312 ms and an average of 75,072 m. When we switched to our max value selection algorithm our times leaped to a slowest time of 64 ms, a fastest time of 21 ms, and an average of 36 ms. This means our algorithm was on average about 2,085 times faster than the random selection.

The changes that made the smallest difference was a tie between the crossover rate and the pool size. As we changed the crossover rate, we only saw a change between our fastest and slowest averages of 19ms, and that was the same as the pool size. Again, this could have been because pool size might not make a difference unless you have a larger proportion given. This would be a great test to have in the future.

Another good test would be to combine some of these and see if we can get a really fast solution. We could go through each test and get the fastest scenario and put them together, to make sure we had the best mutation rate, and crossover rate. Some things you can't necessarily use like pool size and string length and population size, but the end user could use these, and we could recommend the best ones.

The last test would be to create another selection type that would not only see if any letters in the string were correct and in the correct spot, but also see how many were correct, but not in the right spot. This would make our selection more like the Mastermind game talked about in the introduction. It would be able to tell which ones were the right “color” and also which ones were the right “color” and in the right spot.

Most of these tests ran quick, all except for the random selection. We feel that any of these quicker scenarios would still be good to use regardless of the application, since most ran under 1 second.

References

1. https://docs.google.com/spreadsheets/d/1j02IrGhmPlitGScQLGE8JAGbUtbs-IRz1RcfDScYvN4/edit?usp=sharing
2. http://stackoverflow.com/questions/1538235/what-are-good-examples-of-genetic-algorithms-genetic-programming-solutions
3. https://en.wikipedia.org/wiki/Mastermind_(board_game)
4. https://www.mathworks.com/discovery/genetic-algorithm.html?requestedDomain=www.mathworks.com
5. https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)