**Efficiency of the Genetic Approach when Guessing a Known Word**

Mike Cluck & Jonathan Holt

**Abstract.** We investigate how adjusting the population size, mutation rate, and target size, as well using different selection and evaluation methods, affects the number of generations required to discover a known target word when using a genetic algorithm. Genetic algorithms simulate biology by using a selection process to find a pair which will mate, thereby crossing over their chromosomes to produce new offspring. Random mutation is simulated by randomly mutating one gene in an organism in a percentage of the population. We found that EXPAND ON THIS AS DETAILS COME OUT.

**1 Introduction**

Genetic algorithms simulate biology to produce a population of organisms that represent the domain of the problem and by directed selection, random mutation, and effective fitness evaluation, find a solution to that problem domain.

Our problem domain is that of random codes and seeking to generate a known target code. In this paper we discuss how altering different parameters within our genetic algorithm can produce different rates of convergence. Performance is measured in number of generations before the target is generated meaning a few number of generations is a more efficient method. We also look at the amount of variance across multiple runs to verify how reliable a given method is.

The rest of the paper is structured as follows. In section 2 we discuss the algorithm itself, how it improves the population, and what variables can be changed. Section 3 outlines each of our five experiments and shows what our control parameters are. This is then further broken in to sub-sections discussing the details of each experiment. In section 4 we discuss what conclusions can be made from these experiments. Section 5 discusses future work that could be done and how that additional work may increase or decrease the rate of convergence. Finally, all referenced material will be found at the end.

**2 The Algorithm**

The algorithm we implemented on which to run our tests accepts five arguments that allow us to test different set ups with minimal effort. From the command line we can specify the population size for each generation, the type of selection used between generations, the crossover (or how the survivors reproduce), the rate of mutation, and how each element will be evaluated for fitness.

Taking inspiration from Mr. Ben Coppin [1] we developed the following algorithm for generating and evolving a population. At the beginning of the process, an array is filled with pseudo-random strings the same length as the target string. Each of these initial strings is given a fitness score by the evaluation function specified. These scores will be integers less than zero, with zero being the fitness threshold (meaning the guessed string is correct). The fitness scores are then used in the selection process. After the survivors are selected each round, they are paired up and their genes crossed over to repopulate the new generation back to the specified generation size. A given percentage of the population, as specified by the mutation rate, is selected at random and mutated by randomly changing a single character of the string. From here the process repeats itself until the most fit string of a generation reaches the fitness threshold of zero.

**3 Experiments**

Experiments will be run with a set of default parameters. A change will be made to one parameter for each experiment and the resulting change in performance will be measured. The control group parameters can be seen in Table 1-1.

|  |  |
| --- | --- |
| Target | HelloWorld |
| Selection | Fitness Proportionate |
| Crossover | Single Point |
| Mutation Rate | 5% |
| Generation Size | 100 |
| Fitness Evaluation | ASCII Sum |
| Tournament Size | N/A |

*Table 1*

The “ASCII Sum” method we use for measuring fitness can be represented with the following formula.

Σ -|ASCII(desired character) – ASCII(actual character)|

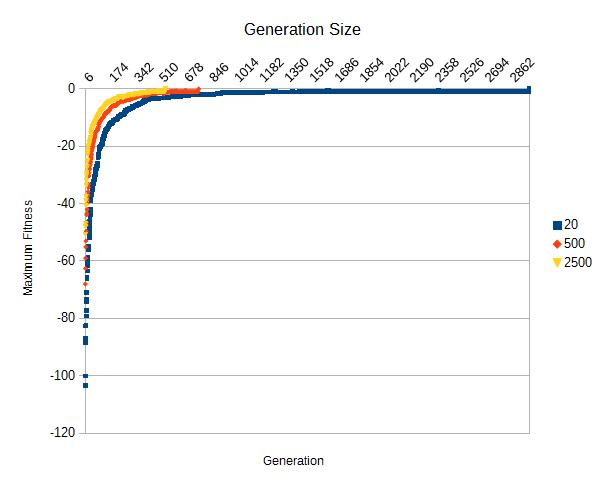
Meaning we take the summation of the absolute value of the difference between the ASCII code of the desired character at each position with the actual ASCII code of the character at each position. We will compare the results of each experiment against the results of the control to see how changing each parameter effected the outcome.

Five experiments will be run testing the effects of each parameter on the results. In the first experiment we adjusted the population size from 100 to 20 then 500 and finally 2500. The second experiment adjusts the length of the target code making it both smaller and larger. In our third experiment, we apply a tournament based selection method instead of a fitness proportionate method. The fourth experiment increases the mutation rate. Lastly, the fifth experiment evaluates each string based on how many characters are correct, rather than the difference of their ASCII values. Each experiment was run 10 times and an average of the results was taken.

**3.1 Generation Size**

For this experiment, we tried decreasing and increasing the population size to see it's affects on solution convergence. A larger population introduces greater genetic diversity. It is thought that this added diversity will increase the likelihood that the target is found sooner. Likewise, a smaller population requires more mutation to generate the correct code.

As shown in Figure 1, an increased population size improves the rate at which a solution is found. With a population size of 20 a solution is found after approximately 2800 generations where as with a population of 2500, a solution is found within approximately 500 generations. When running the control, we found that approximately 1000 generations we needed to find the solution.

*Figure 1*

If the population is below 100 we begin to see a nearly exponential increase in required number of generations.

**3.2 Code Length**

In this experiment, we tried decreasing the code length by 50% (“Hello”) then increasing it by 50% (“HelloWorldHello”) and 100% (“HelloWorldHelloWorld”). We expected to see a significant decrease in the number of generations as we reduced the size of the target code and a similarly significant increase in generations as we increased the size of the target code.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Minimum | Maximum | Variance |
| Control | 606 | 1529 | 27.84% |
| -50% | 24 | 604 | 44.42% |
| +50% | 1351 | 2745 | 25.42% |
| +100% | 1690 | 3395 | 21.27% |

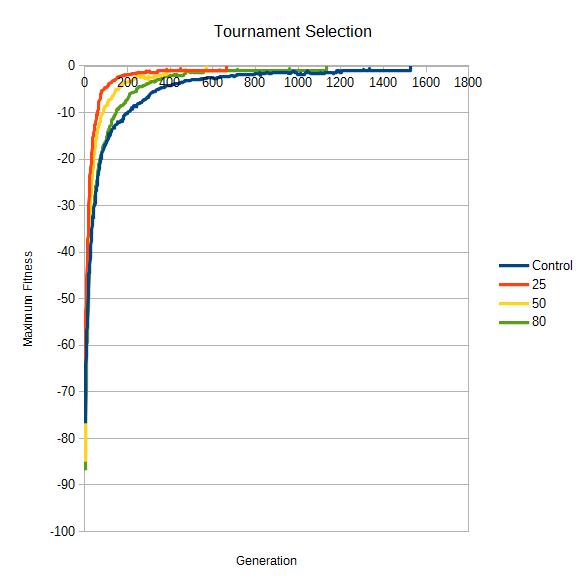
*Table 2*

Initially, the results showed what we expected. As you can see in Table 2, decreasing the size of the target code by 50% dramatically decreased the number of generations per solution. Increasing by 50% and 100% increased the average generation count by approximately 190% and 230%, respectively. What wasn't anticipated was the amount of variance across solutions. A very clear pattern emerges showing that the greater the length of the input, the less amount of variance between amount of required generations.

The variance phenomena can be accounted for when you consider that as the code length decreases, the likelihood that the required chromosomes already exist within the population increases. However, it's still entirely possible for none of the chromosomes in the population to match with the target code meaning a solution could be found very quickly but in many cases it will still require a decent amount of work to find the answer.

**3.3 Tournament Selection**

So far, all of the experiments have been using a fitness proportionate selection method. We experimented with using a tournament selection method. We believe that using a tournament will ensure that stronger organisms survive and weaker ones are removed more quickly, thus resulting in faster convergence. To test this, we started with the same population size as the control, 100 organisms, and tried holding tournaments with 25, 50, and 80 organisms. We expected that the efficiency will increase up until 50, as that would be half of the population participating and anymore would allow too many organisms with lower fitness into the tournament.

*Figure 2*

What we actually saw was very interesting. In every case, tournament selection out-performed the fitness proportionate method as expected. However, rather than seeing an increase in efficiency up to 50, we actually saw a decrease. A lower tournament size resulted in a more efficient route to the solution.

This result can be explained when you consider that as you reduce the tournament size, you're only allowing the very strongest of the organisms through. At a size of 25 that still represents one-fourth of the population size, which gives enough genetic diversity to allow efficient evolution.

**3.4 Mutation Rate**

What truly separates a genetic algorithm from other similar algorithms is the use of mutation. Mutation adds an organic mechanism for introducing new traits to a population. In our control experiments, we randomly mutated one gene in 5% of the population every generation. We tried changing that to 15%, 20%, and 25%, expecting to see an increase in efficiency in the early sets but a decrease in the latter set.

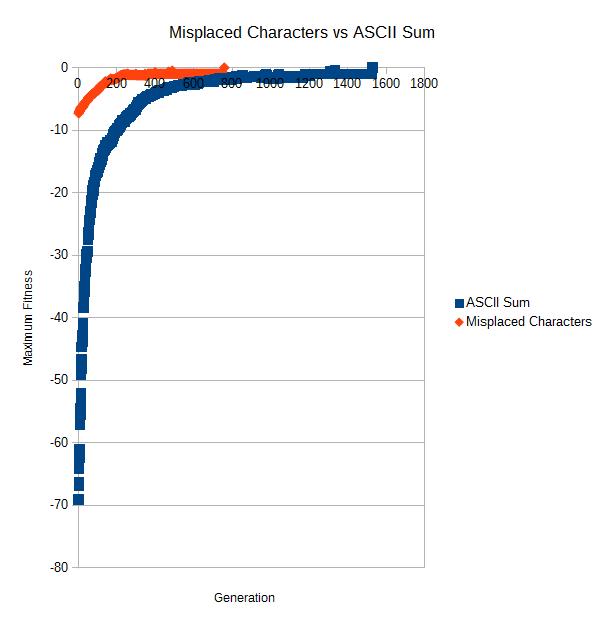
|  |  |  |
| --- | --- | --- |
|  | Minimum Generations | Maximum Generations |
| Control (5%) | 606 | 1529 |
| 15% | 529 | 1808 |
| 20% | 536 | 1592 |
| 25% | 535 | 2715 |

*Table 3*

What we found was that increasing the mutation rate above 5% consistently allowed the system to find a solution in shorter time. However, we saw a far greater variance between the maximum and minimum number of generations. This is too be expected due to the incredibly random nature of mutation. It's just as possible to create a correct gene as it is to create an incorrect gene.

**3.5 Misplaced Characters**

For our final experiment, we decided to try a new way of evaluating the fitness of a given organism. Previously, we were using the ASCII Sum method to calculate fitness but here we used a simpler approach. For every character at a given index that does not match the target character at the same index, reduce the fitness by 1 point. We expected this to take longer as it does not differentiate between “zello” and “aello” as related to the word “Hello.”



*Figure 3*

Instead, we can see that the misplaced characters fitness method converges at approximately twice the rate of the ASCII sum method. In addition to that, even when accounting for the difference in scale, we don't see the initial spike required for the ASCII sum method before it begins to find a solution. There is an initial growth followed by a period of trying to find that last character but that phase is entered into at a much faster rate.

When you consider the fact that mutation is purely random, this result makes sense. The ASCII Sum makes the assumption that when the individual genes are closer to the desired gene it's getting closer but the reality is that there does not exist any mechanism which would push genes closer to the target as it approaches the solution.

**4 Conclusion**

**5 Future Work**

**References**

[1] Coppin, B.: Artificial Intelligence Illuminated. Jones and Barlett Publishers (2004)