Genetic Algorithm Implementation and Variability for Code-Breaking

Zack Austin

ABSTRACT

Many scientists would agree that the usage of machine learning through neural networks and genetic algorithms have shaped the usage of modern AI. Codebreaking and other search-related problems are routinely solved from the Genetic algorithm process. In order to better this process, I'll look into the implementation and usage of Genetic variables and their effects on a string Code-breaking example along with possible alternatives to this operation.

I. INTRODUCTION

The focus of this research is not on whether a Genetic algorithm is fit for a code-breaking problem, but alternatively on how to make that Genetic algorithm a best fit for a solution. Time will be spent explaining the differences through a number of experiments explaining the process of selecting these variables. The process of writing a possible implementation for a Genetic algorithm will also be discussed.

A Genetic algorithm is an algorithm computed in such a way that would give analogy to the process of genetic evolution. Over the course of many iterations populations of individual genes form possible hypothesis to the solution of a possible problem [3]. Genetic algorithms are useful for search problems where the search space may be poorly observed, one such example being either a Sudoku puzzle [1] or fake code-breaking.

The following sections are organized such as: Section II will refer to the implementation of our GA. Section III will overview the experiments included in determining appropriate variables for finding the encoded string. Section IV will discusss each of the varying experiments given a single change to one particular input parameter. Ending with Section V, concluding the use of our GA on this code-breaking example and alternatives to this possible approach.

II. IMPLEMENTATION

A quick overview of my implementation of the Genetic algorithm involves setting up the default values for variables and functions used within along with sequentially applying a fitness function to each individual hypthesis and finally run through crossover and mutation methodologies until a threshold is met.

The fitness function is written to give the proportion of how many letters were off from the specified string, which is chosen randomly with a fixed variable length. The Boltzman Distribution is used with a fitness-proportionate selection to give a probability of choosing that individual string as a parent for crossover. If the probability is met, the string is added to the new list of parents and removed from the population.

After the selection process is complete pairs of parent strings are used to produce child strings in a process known as single-point crossover. Single-point crossover is where two parents are separated into two pieces and then one of each parents pieces are then used to create a new child hypothesis [2]. These child individuals are then appended back onto the population list.

Mutation is then run by randomly selecting a specified amount of the population and changed by randomizing one of it's characters. The fitness of each hypothesis in the population is reevaluated and the process continues until one of the individuals fitness' matches the needed threshold.

III. DEFAULT PARAMETERS

Following this section there are five experiments including a controlled experiment in testing the implementation of this Genetic algorithm. Experiments will run that will test the temperature for the Boltzman Distribution, the rate of population character mutation, the rate of single-point crossover, and the size of the encoded strings the GA is searching against. The fitness function can also be used and tested to either see the number of characters that are correct or the number of characters that are off by ASCII values.

Default parameters will be used to initialize each of these variables and each experiment except the controlled experiment. Experiments will change only one of the parameters that are used. Parameters can be both either a variable constant to be changed as well as potential function changes such as the use of differing crossover or fitness functions.

The size of the population will stayed fixed at 25 individuals for easier analysis, although in real scenarios this value may be several thousands. The length of the hypothesis and specified strings are defaulted to a fixed length of 5. It's possible to run a GA with variable lengthed hypothesis, but this alteration could possibly interfere with the experiment calculations towards the usefulness of certain factors. The number of characters that are off from the specified string will make up the default fitness function. The probability of a parent being chosen for crossover will be defined by fitness proportionate selection [2]. The mutaion rate is chosen to be 15% of the population.

The method of performing crossover will be single-point crossover of 2 parent hypothesis. The rate of crossover is defined to be a maximum of 60%. This is a maximum as there is potential for the implementation to choose a lower ratio of parents to be used in crossover. The reason for this is due to two factors: The selection probability is highly coupled to the temperature constant and there being a possibility of a lower number of parents being added to the crossover list if the temperature is too low. The default temperature chosen is initialized to 2 as well.

VI. EXPERIMENTS

A. Controlled Experiment

The controlled experiment will use the default values mentioned for the Genetic algorithms variables. It will serve as a basis to the generalized data and how the implementation runs. Other experiments will be run against this control to test individual variables usefulness in the code-breaking problem.

Table 1 is a table of the outputted information from the Genetic algorithm's default parameters. The experiment was run a total of 5 times, resulting in an average time of 0.83 seconds.

The fitness difference is 3.22. The fitness difference of an experiment is the difference between that experiment's total fitness value from its initial population to its final population. The fitness difference gives a generalized idea of how the hypothesis' have improved over the course of many generations. This is subject to how fit the initial population is, but nonetheless the GA implementation should generally improve the fitness of the hypothesis' over time.

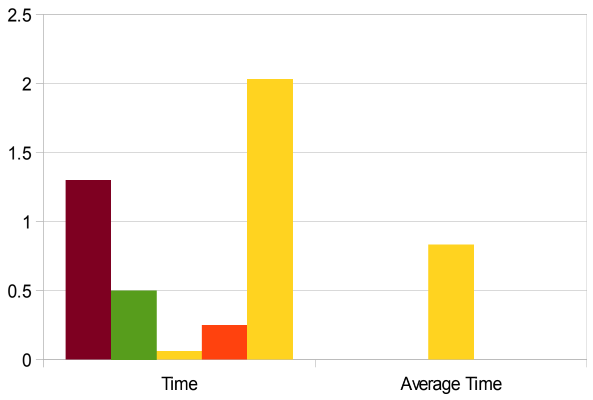

Illustration 1: Time - Controlled Experiment

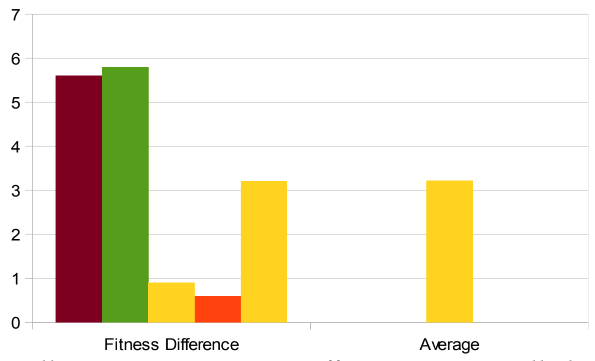

Illustration 2: Fitness Difference - Controlled Experiment

Controlled	Time	Fitness Difference	Population Generations	Expanded Hypothesis
1	1.3s	5.6	12229	144561
2	0.5s	5.8	4299	51626
3	0.06s	0.9	467	5611
4	0.25s	0.6	2167	25662
5	2.03s	3.2	19232	229497
	T_{avg}=0.83s	F_{diff}= 3.22		

Table 1: Output for the Controlled Experiment

Population generations refers to the number of iterations the fitness, selection, crossover, and mutation process were run. Expanded hypothesis refers to the number of children that were produced from crossover. Generally, both are highly coupled to the time aspect of an experiment.

B. Low Temperature = 0.1

In this experiment the temperature will be changed from it's default constant value of 2.0 to a low value of 0.1. The temperature of an experiment refers to a constant used in the Boltzman Distribution that uses that value to determine which individuals of a population are going to participate in crossover. A high temperature means this probability is likely more random while a low temperature will specifically try to select individuals with a better fitness value.

Temperature = 0.1	Time	Fitness Difference
1	1.45	-0.4
2	0.16	3.3
3	0.94	2
4	1.41	2.5
5	0.31	2
	T_{avg}=0.85s	F_{diff}= 1.88

Table 2: Output for Low Temperature Experiment

Table 4 gives the time and fitness values for this experiment for a low temperature value. As seen above, the average time taken to run is unaffected by the change in the temperature constant. The fitness difference, however, is nearly half of that of before. The reason for this being is that lower temperature values result in lower probabalistic values from the Boltzman distribution.

In the GA implementation, the population is sorted in ascending order based on the hypothesis of fitness values. The lower values are run through the fitness proportionate selection first and are rarely added for crossover, resulting in them staying at low fitness values. The individuals with high fitness values are thus given a better chance of being selected for single-point crossover. The crossover process is more likely to give worse child hypothesis' than their parents since they were close-fit. After this happens many generations the data is likely to get normalized along an average fitness value, resulting in a lower net difference in the fitness of the initial and final populations.

C. High Mutation Rate: 85%

The initial mutation rate is changed from 15% to 85% in this third experiment. The idea behind this experiment is to see if randomly selecting and changing characters of random individuals makes a considerable difference in the time or fitness aspects.

Mutation Rate 85%	Time	Fitness Difference
1	0.38	1.6
2	0.08	1
3	0.16	1.3
4	0.16	0.5
5	0.14	0.1
	T_{avg}=0.18s	F_{diff}= 0.9

Table 3: Output for High Mutation Experiment

The results of this experiment are somewhat surprising. Mutating the majority of the population at random lowered the amount of computation time considerably. Thinking through the process more slowly it starts to make more sense. Crossover helps to converge the fitness to higher values, but the usage of 2 parents with single-point crossover rarely would hit the threshold. Mutating a single character of a individual, although random, still had a higher chance of finalizing that fitness threshold. The less surprising output is the large change in the fitness difference from the controlled experiment. Randomly changing population members is bound to result in a failure to both normalize the data as well as converge it to higher fitness values.

D. Low Crossover Rate: 10%

The next experiment sets out to see how changing the amount of individuals chosen for crossover affects finding the fitness threshold. All previous experiments used a 60% crossover rate. Earlier it was mentioned that the crossover rate potentially normalized future hypothesis' over time. The purpose of this experiment is to test and see whether that assumption is true.

Crossover Rate 10%	Time	Fitness Difference
1	1.0s	1.8
2	0.75s	-0.3
3	0.92	0.5
4	1.1s	-0.2
5	1.25s	-0.3
	T_{avg}=1.0s	F_{diff}= 0.3

Table 4: Output for Low Crossover Rate 10% Experiment

The results are interesting on this one, the assumptions from earlier are partially correct. The data normalizes as suspected. An unseen problem occurs where the population converges to values that are similar to the very initial population. Even the previous example with high mutation rates had a higher fitness towards the end. The average time of 1.0s is also slightly higher than that of the controlled experiment's 0.83s. Essentially, this variation of using two parents and single-point crossover is both worse at coming up with fast solutions as well as giving a population of well shaped similar hypothesis. Although, if one were to have a use in keeping the difference the same this parameter change would be useful.

E. Longer Code Length: 6 Characters

All of the previous experiments were attempts at molding a better code breaking algorithm. This final test will run against the controlled test and see how much slower the algorithm will potentially run when the encoded string is 1 character larger. The fitness difference will again be looked at to see if it changes from the increased number of iterations.

Code Size: 6	Time	Fitness Difference
1	4.47s	2.7
2	3.14s	2.6
3	2.91s	0.9
4	7.6s	1.3
5	3.31s	2.1
	T_{avg}=4.286s	F_{diff}= 1.92

Table 5: Output for Longer Code Size: 6 Characters

As seen in Table 5, going up a single character increases all aspects by as much as a magnitude of 10. The fitness difference was also affected, moving away from the average of 3.22 down to an average of 1.92. This kind of dropoff is rather large considering how many characters a GA realistically would have to search for code breaking problems. Illustration 3 contrasts this experiment with the control to better gaze larger increase in time ratio for a string that's one character larger.

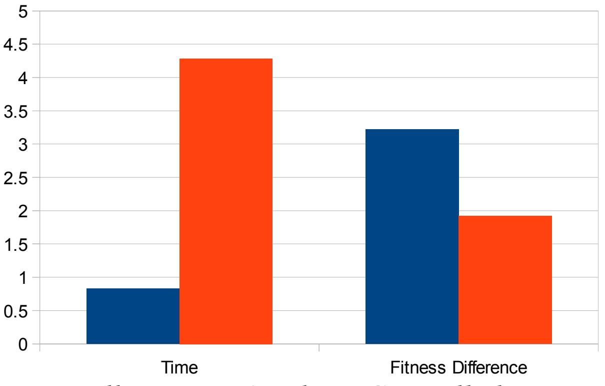

Illustration 3: Blue - Controlled Orange - Longer Code Size (+1)

V. Conclusion

Various methods and paremeters have been proposed to better solve code breaking problems. Experiments were run on breaking larger codes that showed a need for a strong Genetic algorithm implementation. One of them is to change the temperature of selectivity, another includes changing the rates of which mutation and crossover occur, and finally others are included in changing the proposed methods of implementation on those functions.

Concluding from the experiments, single-point 2 parent crossover tends to be too random to be overly useful and makes large changes to the population which aren't always positive. Mutation is a good addition to a GA implementation for it's minute changes, even if they tend to lower the fitness distribution from the initial and final populations. Finally, all changes to the controlled experiment related in the fitness differences to be lowered.

In the future, it would be interesting to look into other methods of crossover such as two-point and uniform. Selection techniques such as ranked proportionate and tournament could also provide potential benefits that are worth studying.

References

[1] Kazemi, S. M., & Fatemi, B. A Retrievable Genetic Algorithm for Efficient Solving of Sudoku Puzzles.
[2] Mitchell, T., M., Machine learning. Chapter 9. p. 250-252. Published by McGraw-Hill, Maidenhead, U.K., International Student Edition, 1997.
[3] Prebys, E. K. (2007). The genetic algorithm in computer science. In *MIT Undergrad. J. Math.*