International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-9 Issue-4, February 2020

Fitness-Orientated Mutation Operators in Genetic Algorithms

Hoijin Yoon

Abstract*: Genetic Algorithms are the most popular metaheuristics used in search-based program solving. They search for a solution by repeating the following operators: selection, crossover, and mutation. By going through several generations of these operators, a solution is reached. The total number of generations depends on the reproduction of offspring. Of the reproduction operators, the mutation operator tends to be chosen with a random approach because the concept of mutation is from the natural evolution cycle. But the effectiveness of genetic algorithms should be monitored, especially in software testing. The effectiveness is represented by the total number of generations, which corresponds to the speed of solution acquisition. This work focuses on mutation as a factor in reducing the total number of generations and devises two contrasting ways to define mutation operators. One is fitness-positive, and the other is fitness-negative. The fitness-negative definition thus appears to fit more aptly. This work determines which of these two methods of mutation achieves the higher effectiveness through conducting a controlled experiment. The result shows that the fitness-positive method takes a smaller number of generations than the fitness-negative method.

***Keywords*: Mutation, Genetic Algorithms, Metaheuristic, Software Testing.**

I. INTRODUCTION

Genetic algorithms (GA) are inspired by Darwin's theory of natural evolution. The first GA was developed by Holland in 1975 to solve optimization problems and was based on biological genetic and evolutionary ideas [1]. It is a heuristic search approach that involves the process of natural selection where fitter individuals can survive better than those which are less fit. The fitter individuals generate offspring for reproduction, which form the next generation [1]. The phases of a GA include *mutation*, which plays the role of preventing the search from heading to a local solution.

The mutation concept is also used in other fields of computer science. *Mutation analysis* [2] is one of them, which measures how adequate a set of test cases is for a program code. It is also called *program mutation*. It generates *mutants* that have small changes from the program code.

Revised Manuscript Received on February 28, 2020.
***** Correspondence Author
Hoijin Yoon*, Department of Computer Engineering, Hyupsung University, Hwaseungsi, South Korea. E-mail: hjyoon@uhs.ac.kr

A test case is applied to the original program and its mutants at the same time to check if their results are the same or not. Receiving the same result means that the test case does not differentiate the original code and its mutant. The mutant is supposed to result in a different result than the original, since the mutant has been modified on purpose. Therefore a test case which cannot differentiate the original program and its mutant is not adequate for testing the code. On the other hand, if the results of the original program and its mutant are different for a certain test case it means that the test case works well enough to differentiate the mutant from the original program. In this situation, mutation analysis says that a test case *kills* the mutant, and the aim is to kill all mutants for a 100% adequacy of a set of test cases [3].

The performance of mutation analysis depends on the method of mutation performed on an original program, which are called mutation operators. For example, in C program code a mutation operator replaces a condition notation such as > to another one such as <. Many research works have done this for a diverse range of programming languages; C, Java, C++, and so on.

With this concept of mutation operators, this work figures out the relation the method of chromosome mutation and the performance of the GA when performing a search. Usually, a GA does not consider how to mutate as much as mutation analysis does. This may be because GA itself is a heuristic approach, which does not depend on a specific scenario to search for a solution. GA usually adopts a random value generation approach to mutation. We consider the result where more sophisticated mutation operators are adopted in GA. In this work, the fitness-related operation is considered and is designed in two contrasting ways: fitness-positive and fitness-negative. In terms of the mission of the mutation operation in GA, the fitness-negative operators would appear to result in a better performance of GA, because mutation operation in GA plays the role of preventing stagnation. This work deigns a controlled experiment and analyzes results to answer the research question—*"Does a fitness-negative mutation operator perform more effectively than a fitness-positive one"*

Section II explains the phases of GAs including mutation. In Section III a couple of mutation operators (rather than a random approach) are introduced and coded in a python program for the experiment. The controlled experiment is designed in Section IV and its results are analyzed in Section V. Finally, Section VI concludes this work and answers the research question.

II. GENETIC ALGORITHM

The GA process begins with selecting fit individuals from a population. To find the fit individuals, a function is formulated before starting the GA. It is called the *fitness function*, which is a kind of map to search solutions.

Once the selection is done by a fitness function, the GA produces offspring which inherit the genes of the parents' chromosomes through *crossover* operations. The offspring will be a member of the next generation. The phases of selection and crossover continues to iterate until the fittest individual is found by the fitness function [4]. The detail phases of a GA are as follows; *Initial population, Fitness function, Selection, Crossover, and Mutation*.

A. Initial Generation

At the beginning, the GA selects individuals as members of the initial population. Population means a set of individuals. An individual consists of parameters, which are called genes in GAs. A string of genes is called a chromosome, and each individual has its own chromosome. The chromosome is one of the solutions of the problem the GA is finding a solution to. In GAs, a chromosome is represented as a string. A string of alphabet characters is the standard.

B. Fitness Function and Selection

The fitness function determines how good a fit an individual's chromosome is to the solution. The function returns a fitness score for each individual, where an individual achieving a higher score has more probability to be selected. According to the theory of natural selection, the fitter individual is guaranteed a higher probability of survival among the population. The function is dependent on the problem and its potential solution.

In the selection phase, the individuals evaluated highly by the fitness function are selected as parents of the next generation. The selected individuals are used in generating their children through *crossover* and *mutation*. The common selection method is the roulette wheel selection [4], where the probability of an individual being selected is dependent on the fitness value acquired before selection, with individuals with lower fitness values having a higher chance of being eliminated from the population.

C. Crossover and Mutation

Crossover is an operation for reproduction which combines the chromosomes of the selected individuals. There are many different variations; *single crossover point, single random crossover point, multiple crossover points, and multiple random crossover points* [5]. These variations define how one chromosome is divided into a unit for the crossover operation. For instance, single crossover point creates a single cutting point, which is normally the middle of chromosome.

Many studies states that mutation is necessary to reach the solution in GAs since the repetition of only crossover results in stagnation, where it would never reach the solution [6]. The random mutation selects the genes to be mutated randomly, and then replaces genes' encoding values, also in a random way.

III. MUTATION OPERATORS

Of the reproduction operators, the mutation operator tends to use a random approach since this matches the concept of mutation from natural evolution. However, special care must be taken over the effectiveness of genetic algorithms in software testing [7], where its mutation analysis technique adopts the concept of mutation to measure the quality of test cases. The effectiveness is represented by the time taken to find the solution, which can be measured as the total number of generations. This testing is based on the notion that the quality of a set of test cases can be a measure of the ability of that the set of test cases to differentiate the program under test from its different versions, which are called *mutants* [2]. It also assumes that a mutant is an incorrect version of program, and that it is supposed to be detected by test cases. Many studies of mutation analysis state that the analysis depends heavily on mutation operators, and many of these studies have developed mutation operators for work in specific programming environments such as object-oriented programming, aspect-oriented programming, and so on [8].

This work focuses on mutation as a factor to reduce the total number of generations. Beyond the random approach that GA normally adopts, we devise two contrasting ways to define mutation operators; one is fitness-positive, and the other is fitness-negative. We implement these two methods of mutation and a simple random approach in a program that will be executed in the controlled experiment in Section IV.

IV. CONTROLLED EXPERIMENT

The mission of Mutation was originally to inject diversity and prevent stagnation. In terms of this concept, the fitness-negative method seems to fit mutation better. In this section a controlled experiment is introduced to analyze which of two methods of mutation, fitness-positive or fitness-negative, achieves a higher effectiveness. The research question of this experiment is as follows;

"Does a fitness-negative mutation operator perform more effectively than a fitness-positive one?"

A. Subject

In the experiment, the GA searches a test set consisting of the least number of test cases that covers all pairs of input parameters. This implements a pairwise testing searching problem, which are popularly selected test cases in a large group of possible candidates. This is known as a NP-complete problem [9]. This problem is addressed as a proper GA application. We consider that a test case consists of three parameters and each parameter handles their own available values; the set of available values for the first one are {a, b, c}, the second one's are {0,1, 2}, and the third one's are {A,B,C}. A pair of four possible combinations within these three parameters are represented in one chromosome. Figure 1 shows what a chromosome encodes. We implement this program selecting pairwise test cases in python as demonstrated by Clinton Sheppard in [10].

	tc1	*tc2*	*tc3*	*tc4*
ch1	c 1 C	c 1 A	a 0 B	b 0 B
ch2	a 2 C	b 1 A	c 0 C	b 0 C
ch3	c 0 A	c 1 A	a 2 A	a 1 B

Fig. 1.Three sample chromosomes in the searching solution of pairwise testing

B. Fitness Function

Fitness represents how good an individual is in terms of the expected solution. Depending on the fitness of an individual, it is determined to be or not to be used in the reproduction process. Therefore the fitness measuring matrix, called fitness function, is totally dependent on what the search problem.

The goal of the pairwise testing that we have implemented for the experiment is to find a pair of four test cases that contains as many different combinations of test case as possible. To implement this fitness, we first assign an initial value and then decrease the value whenever finding same patters within one chromosome.

C. Mutation

As mentioned in Section III, the mutation operators in the experiment are *fitness-positive* and *fitness-negative*. When mutating a chromosome, each gene is changed in a positive way to fitness or in a negative way to fitness. The fitness-positive operator makes its final fitness value higher by replacing genes on purpose, and the fitness-negative operator does the opposite.

The mutation rate is the frequency of mutating genes over time, also treated as an important factor affecting how fast the GA finds a solution. We set the rate to two kinds of values, 0.2 or 0.1, in the experiment to perform a simple check on if the positive or negative mutation operators would depend on the rate.

D. Independent variables vs. Dependent variables

The experiment handles six settings with three different population sizes and two different mutation rates, and the size of selection keeps around 40% of the population size. The experiment uses population and mutation rate (*MR*) as independent variables. It defines the total number of generations as a dependent variable. This number varies in both fitness-positive operations and fitness-negative operations. For convenience the number of positive cases is called *Gen_P* and the number of negative cases is called *Gen_N*. *Gen_P* is an abbreviation of the total number of generations in the case of applying a fitness-positive mutation operator, and *Gen_N* is an abbreviation of the total of generations in the case of applying a fitness-negative mutation operator.

V. RESULTS

A. Round 1 : Comparison under basic conditions

The independent variables for each of the six cases in the experiment are shown in Table-I. In the first round of the experiment, the program finding a set of pairwise sets of test cases was executed with a population of and a mutation rate of 0.1, and it found a solution after 10 generations when applying a fitness-positive operator when mutating chromosomes. However, in the cases applying the fitness-negative operator the same stop condition was not reached, and the program was never finished. That is why there is no applicable number in the *Gen_N* column in Table-I.

Table-I. Generations in PO vs. NO

	Independent Variables		*Depended Variables*	
case#	Population	*MR*	*Gen_P*	*Gen_N*
1	25	0.1	10	n/a
2	25	0.2	8	n/a
3	50	0.1	9	n/a
4	50	0.2	7	n/a
5	100	0.1	15	n/a
6	100	0.2	12	n/a

B. Round 2 : Comparison under reachable condition

After round 1, we changed the stop condition to be slightly looser until the program running with a negative mutation operator stopped. Finally, we found the strongest but reachable stop condition, and under that feasible condition round 2 achieved results for *Gen_N* as well as *Gen_P*. These results are shown in Fig. 2. Interestingly, *Gen_P* with a *MR* of 0.2 and *Gen_N* with a *MR* of 0.1 are the same in all the cases of populations. The bold line represents both at the same time in Fig. 2. There are two findings from the chart. First, the gap between *Gen_P* and *Gen_N* under the same *MR* is bigger in case of *MR* 0.2 than in the case of *MR* 0.1. Second, in the case MR=0.2 there is a larger gap between the total number of generations than in the case MR=0.1.

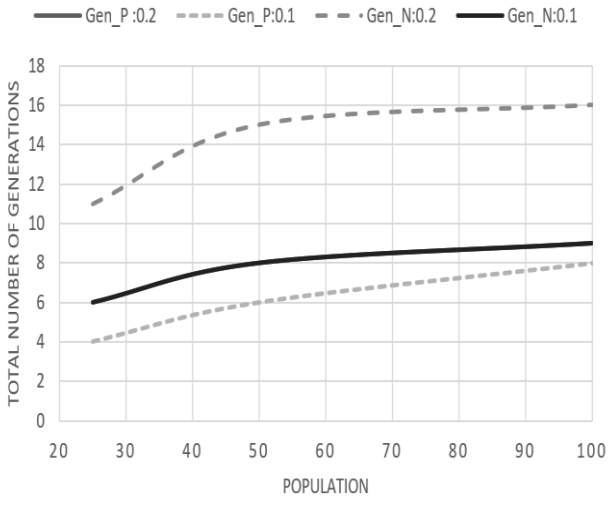

Fig. 2. Comparison of *Gen_P* and *Gen_N* in the cases *MR* equals 0.1 or 0.2

C. Round 3 : Variation of *Gen_P* and *Gen_N*

While doing Round 1 and Round 2, we have recognized that *Gen_P* and *Gen_N* would be different from time to time due to random behavior, such as defining the initial generation, still being present in the GA.

Table-II. Gen_P and Gen_N in 30 times repetitions

Case	*Gen_P*				Gen_N			
	Mean	Median	Max	Min	Mean	Median	Max	Min
1	4.73	5.00	8	3	8.30	8.00	13	4
2	5.17	5.00	9	3	5.67	7.50	9	4
3	6.03	6.00	9	4	12.20	12.00	19	6
4	6.60	6.50	10	4	7.80	7.50	13	5
5	7.37	7.00	11	4	17.73	17.00	36	9
6	7.60	7.00	11	5	8.80	9.00	13	6

Round 3 repeats the same experiment 30 times in each case from case 1 to case 6. The result of 30 repetitions is shown as the mean, median, max, and min values in Table-II. We then drew box-and-whisker diagrams to check how much the randomization affects the final number of generations. These are shown in Fig. 3. The title of each diagram is "population / selection / MR." For instance, "100/40/0.1" refers to a population of 100, a selection of 40, and a mutation rate of 0.1.

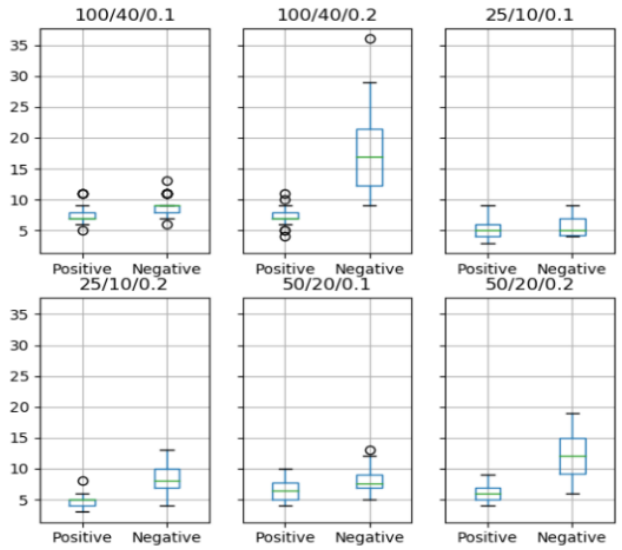

Fig. 3.Box-and-whisker diagrams of Gen_P and Gen_N for 30 repetitions

From Fig. 3 it can be seen that *Gen_N* has a wider variation than *Gen_P*, and that the wider variations tend to happen in cases where *MR* is 0.2 rather than *MR* is 0.1. On the whiskers of the diagram, the case of 100/40/0.2 results in a *Gen_N* from 9 to 36. Overall, the fitness-positive operator results in more predictable and expedient searching solutions as the lower the number of generations leads to faster solutions.

VI. CONCLUSIONS

In GAs, the mutation operation plays the role of preventing the searching process from stagnating in subsequent generations. In these terms, the operation makes individuals' fitness lower and take the population out of the possible stagnation. That is why we suggested fitness-negative mutation operators. Moreover, we brought a research question, *"Does a fitness-negative mutation operator perform more effectively than a fitness-positive one?"* and designed a controlled experiment to answer the question. Through the experiment's three rounds, a fitness-positive operator resulted in a better performance. From this examination, we learned that the fitness is still an important guideline even in a tool preventing stagnation by injecting unexpected individuals into the population.

ACKNOWLEDGMENT

This work was supported by the Hyupsung University Research Grant of 2019.

This research was also supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education (NRF-2017R1D1A1B03034557).

REFERENCES

1. Holland, J.H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*; MIT Press: Cambridge, MA, USA, 1975.
2. R.A. DeMillo ; R.J. Lipton ; F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", *IEEE Computer*, 11(4), IEEE, 1978
3. Roland Untch, Mary Jean Harrold, and Jeff Offutt, "Mutation Analysis Using Program Schemata", in *Proceedings on International Symposium on Software Testing and Analysis*, pages 139-148, Cambridge, Massachusetts, June 1993
4. M. Mitchell, *An Introduction to Genetic Algorithms*. The MIT Press, 1999
5. Alan T Piszcz, Terence Soule, "A Survey of Mutation Techniques in Genetic Programming", in *Proceedings on GECCO'06*, July 8–12, 2006
6. D. M. Tate and A. E. Smith, "Expected allele coverage and the role of mutation in genetic algorithms," in *Proceedings of the 5th International Conference on Genetic Algorithms*. 1993
7. Ziming Zhu and Li Jiao, "Improving search-based software testing by constraint-based genetic operators," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*, pp.1435-1442, 2019
8. K. N. King and Jeff Offutt. "A Fortran Language System for Mutation-Based Software Testing", *Software Practice and Experience*, 21(7):686-718, July 1991
9. P. Flores and Y. Cheon, "PWiseGen: Generating test cases for pairwise testing using genetic algorithms," in *Proceedings on IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, vol. 2, pp. 747–752, 2011
10. Clinton Sheppard, *Genetic Algorithms with Python*, CreateSpace Independent Publishing Platform, 2018

AUTHORS PROFILE

Hoijin Yoon is currently an associate professor in the Department of Computer Science and Engineering at Hyupsung University since 2007. She received a B.S. degree in Computer Science from Ewha Woman's University in Korea and received M.S. and Ph.D. degrees in Computer Science and Engineering from Ewha in 2004. Her research interests are in software engineering with particular emphasis on testing. Her current research project is on the verification and validation of interactions between intelligent things and humans.