IJSGS

ISSN: 2488-9229
FEDERAL UNIVERSITY
GUSAU-NIGERIA

INTERNATIONAL JOURNAL OF SCIENCE FOR GLOBAL SUSTAINABILITY

A Mutative GA-Based Regression Test Case prioritization for Object-Oriented Programs

[1]Samaila Musa and [2]Abdullahi Salihu
[1]Department of Computer Science, Federal University Gusau.
[2]Department of Computer Science, Nasarawa State University, Keffi.

Corresponding Author's Email: samailaagp@gmail.com

Received on: January, 2021 **Revised and Accepted on:** March, 2021 **Published on:** March 2021

ABSTRACT

Software testing is a set of activities in Software Development Life Cycle (SDLC) carried out with the intent of finding faults before delivery to the users. The software testing activity consumes almost half of the development effort, time and costs especially for large software systems. There are a lot of techniques proposed to prioritize test cases based on certain coverage criteria with the aim to reduce time and costs of testing. This research basically focused on test-case ordering and faults coverage by applying mutative genetic algorithm approach (MutGA). In this research, we used a sample test suite to demonstrate the workability of the proposed approach. MutGA approach is expected to gives us better results and save time and cost in term of Average percentage of rate of Fault Detection (APFC) as illustrated with sample example when compared with retest-all and randomly ordered test cases techniques.

Keywords: Test case; fault coverage; Genetic algorithm; Software Testing; Test Case prioritization

1.0 INTRODUCTION

1.1 Background of the study

Software maintenance is one of the software phase in software development life cycle and is an expensive phase for nearly 60% of the total cost of the software production (Roger, 2001). The cost of correcting a fault after coding is at least 10 times as costly as that before finishing coding and the cost of correcting a production fault is at least 100 times (Perry, 2006) as shown by IBM's studies and others. Grottke and Trivedi (2007) assert that the cost depending on the types and sizes of the software systems as observed in the literature that ranges from 5 times to 100 times.

One of the most important activities in software maintenance is regression testing (Rothermel and Harrold, 1994) that is performed in order to ensure that modifications due to debugging or improvement do not affect the existing functionalities and the initial requirement of the design. According to Zhang *et al*. (2009) regression testing comprises execution of an enormous amount of tests and is tedious, but it might not be feasible to repetitively re-execute all the test suite in order to tests the software during regression testing due to resource restraints. In this way, it is reasonable to select test case by selecting part of the test suite so that software is tested for early identification of faults. There are numerous regression tests selection strategies, for example, retest-all or randomly selects test cases, however using some pre-defined criteria for selection of test cases might result in more fault identification.

After the selection, when there is enormous number of the selected test cases, and when the testing resources are limited or insufficient to execute all test cases, a well-designed execution order of test cases seems especially significant, and may improve the likelihood of detecting faults earlier, which may be especially important when testing with limited test resources there is a need to arrange the tests based on some criteria in order to reduce regression testing time, i.e. prioritization of the tests.

Regression test case prioritization strategies is described as a process that allow the software testers to arrange their tests into certain model so that those with the most elevated need are executed before the lower need test case, and it can be utilized in conjunction with tests selection when tests disposing is satisfactory, also it might increase the utilization of testing time more beneficial than non-prioritize when the process of re-executing the test cases is terminated without prior notice. The mutative approach selects a test case with the largest

faults detection capability, by avoiding the crossover operation with the two chromosomes as initial population. The process is repeated until all the test cases ordering with highest number of faults detected is selected and run based on the number of fault detected. This research work focuses on reducing time of test case ordering and improving or achieving the same effectiveness in average percentage of faults coverage technique.

1.2 Related Works

The current strategies and identification of their relative properties would be necessary in order to provide a basis for comparison with the proposed strategy in this paper. This section presents fault severity and analyses of OOP regression test case prioritization strategies. Rothermel, *et al*. (2001) proposed *nonPrior* and *randPrior* to simply compare them with their regression test case prioritization strategy. In the earlier, the test cases are selected based on the affected statements. Test cases that execute at least one of the affected statements are selected. After the selection of the test cases, the order is maintain. The order produced after the selection might have ordered test cases with less number of affected statements. In the latter, the test cases are selected based on the affected statements. Test cases that execute at least one of the affected statements are selected. After the selection of the test cases, the selected test cases are ordered at random. The prioritization of the selected test cases is performed by generating a different randomly order of set of test cases. The best among the ordered test cases is selected as the best ordering by computing the total number of affected statements covered. Prioritizing test cases at random might order test cases with less number of affected statements first. To date, most weighted test suite prioritization strategies for genetic algorithm belong to the category of using fixed fault severity prioritization, because they only consider a fixed fault severity when prioritizing test case for regression testing.

Athar and Ahmad (2014) presented a test suite optimization technique using a Genetic Algorithm. The test suite formed gives 100% code coverage. As the chromosome size increases, the number of iterations required to get the optimized test suite is also increased. But the changes might not have affected all the test cases, therefore there is need to select affected test cases before prioritizing. A genetic algorithm to prioritize test suites was proposed by Purohit and Sherry (2014). The initial population was created at random where each chromosome represents the possible solution of the problem. The sequence of test casesis a chromosome and the collection of these chromosomes is the population. In their approach, test cases with the higher number of modified lines are more efficient and have higher priority. The number of modified lines determined how fit and efficient a test case sequence for execution during testing. They used one-point crossover operation after selecting two chromosomes to produce new chromosomes, and after the crossover, two genes are selected for mutation where the values are swapped. After repeating the crossover and mutation operations for a large number of generations, (as the termination criteria) the authors assumed that nearly optimal value will be generated. Precisely, the approach prioritizes test suites based on their fitness value, and the fitness values are computed based on the number faults detected. But the approach does not consider dependencies between the changed and other statements in order to find statements that are dependent on the changed statements. Also, the change s might not have affected all the test cases, there is a need to select affected test cases and then prioritize them. They also used the same severity for a fault even if the fault was executed by the preceding test case.
Konsaard and Ramingwong (2015) presented a test case prioritization based on total coverage using a modified genetic algorithm, whereby it takes advantage of simplicity and ability to vary the population to provide a variety of test cases for selection and prioritization processes. A randomized test case input is generated by the modified Genetic Algorithm based on one condition that each of them covers all independent paths of the program to assure the highest coverage. The authors evaluate the approach on the average percentage of condition covered and execution time compared with five other approaches. The results outperformed the other five approaches.

Currently, most of the optimization techniques are based on single objective and static in nature, but in Kiran *et al*. (2020), the authors proposed three variants of self-tunable Adaptive Neuro-fuzzy Inference System i.e. TLBO-ANFIS, FA-ANFIS, and HS-ANFIS, for multi-objective regression test suites optimization. The

two benchmark test suites are used for evaluating the proposed ANFIS variants. They measured the performance of the proposed ANFIS variants using Standard Deviation and Root Mean Square Error. The experimental results concluded that the proposed technique effectively reduces the size of regression test suite without a reduction in the rate of fault detection.

In Wang *et al*. (2020), the authors present a test case prioritization technique using fixed size candidate set adaptive random testing algorithm to reduce the effect of randomness and improve rate of fault detection effectiveness. The results of the experiment show that by a statistical analysis, indicate that the technique is more effective than random and the total greedy prioritization techniques in terms of rate of fault detection effectiveness.

2.0 RESEARCH METHODOLOGY

The overall structure of the research activities and steps that have been taken to achieve the research objectives is illustrated in Figure 2.1. The strategy consists of three major components; identification of affected statements, test case selection and prioritization as shown in Figure 2.1.

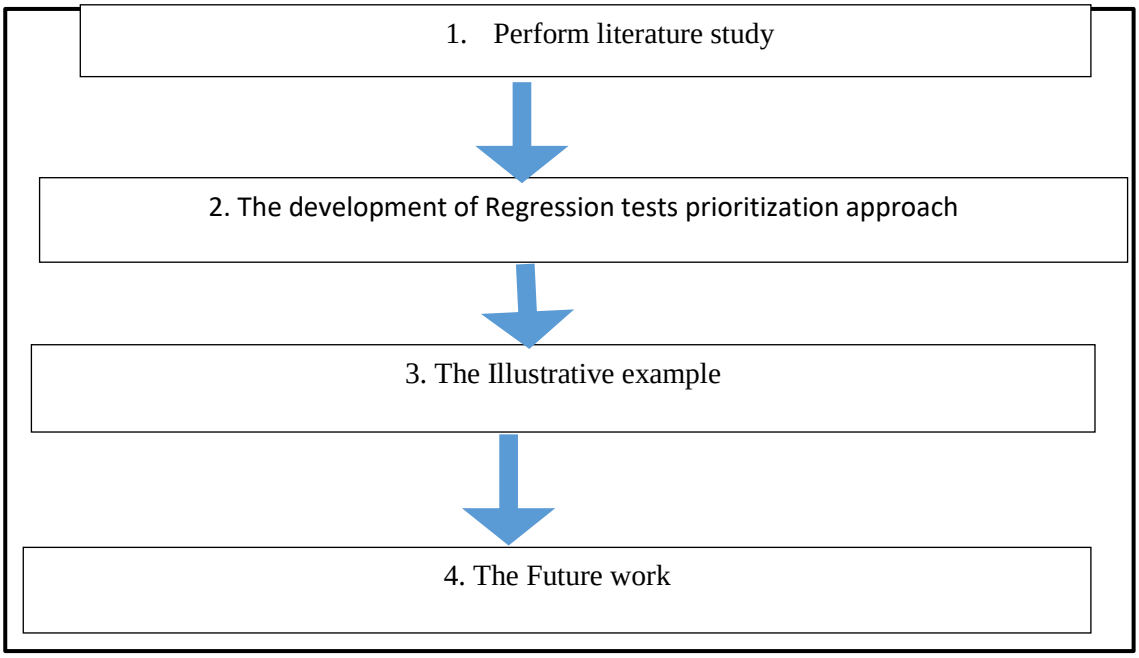

Figure 2. 1: Research Methodology Overview

Figure 2.1 shows that *mutGA* starts with the construction of ESDG of the original OO program to identify the affected statements as described in our previous works(Musa *et al*. 2014a, 2014b, 2015). The second part is for test case selection using the affected statements (Musa *et al*. 2014a), and the third part is for test case prioritization from the selected test cases. We modified our prioritization technique to design our new approach and it was given a name as Mutative Genetic algorithm (MutGA) as shown in Figure 2.2.

The major difference between our approach and previous techniques Musa *et al*. (2014a, 2014b, 2015)are:

i. Random Population Generator Component
ii. Mutation performer instead of crossover performer

2.1 Random Population Generator Component

Given the encoded selected test cases, *T',* a random population of two chromosomes is generated as the initial population unlike in the initial work that population of n as the size of the selected test cases by using Random-population-generator. Figure 2.3 shows the algorithm for random population generator component.

The algorithm takes encoded selected test cases as input, and then it produces an initial population of a set of chromosomes as output

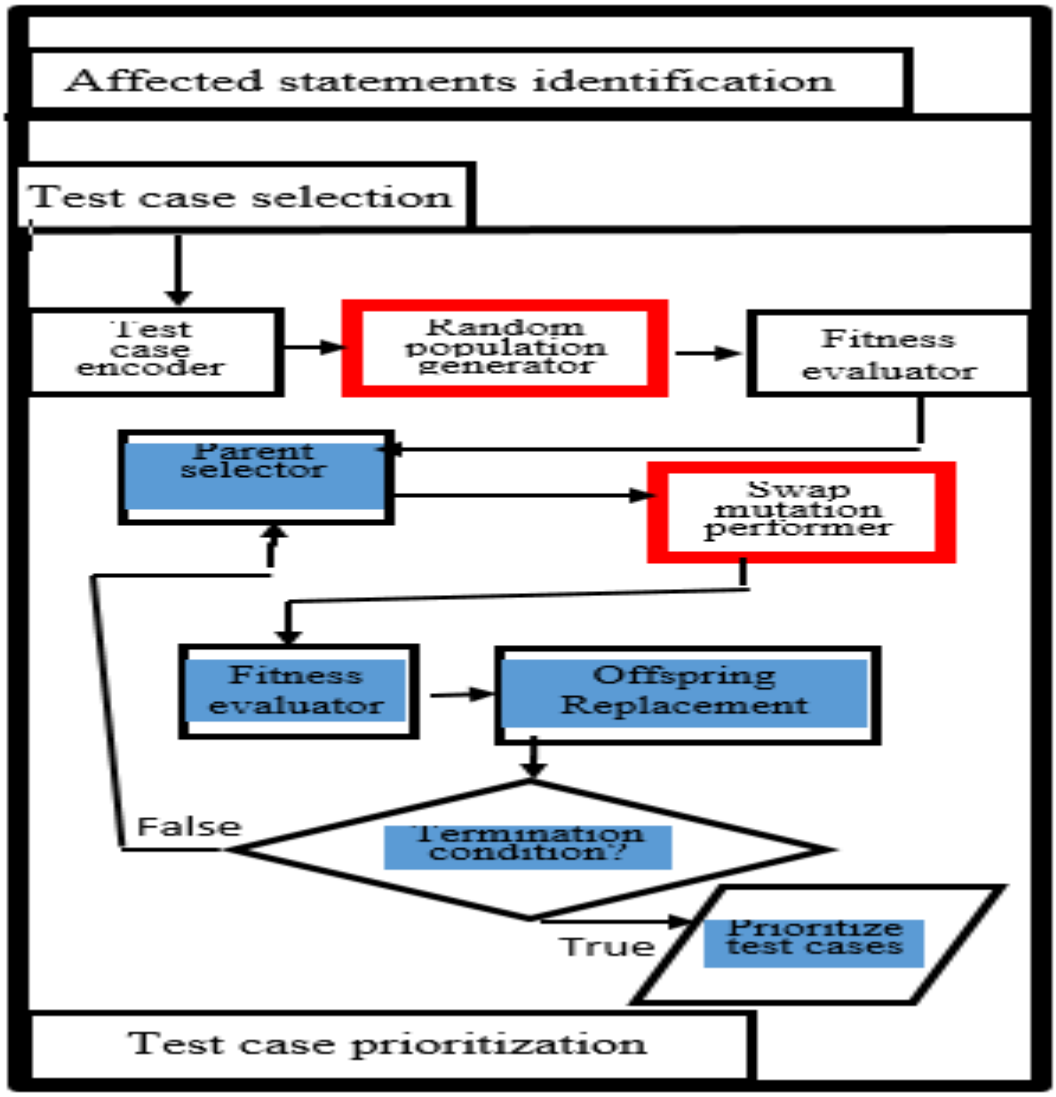

Figure 2.2.Conceptual Design of *mutGA*

Algorithm Random-population-generator
Input = *T*
Output = *P*
begin
1. *P* = { }; *// initial population*
2. Identify all the encoded selected test cases, *T*
3. **for** the two chromosomes**do**
4. **begin**
5. generate an order of *T'* as chromosome, *C*;
6. *P = P ∪ {C};*
7. **end for**
8. **return** *P = {C₁, C₂,};*

End

Figure 2.3. Algorithm for Random Population Generator Component

.

2.2 Swap Mutation Performer Component

Given two child generated from the crossover, this component performs the mutation operation in order to produce two new chromosomes. A chromosome is normally altered in one or two genes by mutation from its underlying state that can bring about totally new gene qualities being added to the gene group. Two random integer numbers are generated, between 1 and *numberOfGenes*, i.e. r_1 and r_2. The genes of these positions will simply swap their genes for the first child, and the same process is repeated for the second child.The genetic algorithm might have the capacity to touch base at a superior arrangement than was already conceivable as the result of new gene values added. The main reason for change is to keep the search falling into a local optimum of the state space. Figure

2.4 shows the algorithm for swap mutation performer component.

The algorithm takes two children as input, and then it produces two new chromosomes as output.

For example, given $child_1$= {*2, 5, 7, 1, 6, 8*} and $child_2$= {*7, 1, 2, 5, 6, 8*}, the Swap-mutation-performeralgorithm generates two random integer numbers between 1 and 6 (*numberOfGenes*), i.e. r_1 and r_2. The genes of these positions will simply swap their genes for the first child (c$hild_1$), and the same process is repeated for the second child ($child_2$).

$Child_1$ = *swapMutation* ($child_1$)

$Child_2$ = swapMutation ($child_2$)

Algorithm Swap-mutation-performer
Input : {$child_1$,$child_2$} // the two children produced by crossover
Output : {$child_1$,$child_2$} // the two newly children produced by mutation
begin
1. $child_1$ = {g_1,g_2,..,g_n}; // *n>=1 is the first child*
2. $child_2$ = {g_1, g_2,…,g_m}; // *m>=1 is second child*
3. generates two random integer numbers r_1 and r_2
4. **for** each child, $child_i$**do**
5. *// swap the genes of the two random positions*
6. $child_i$ = gene($child_i$) + gene($child_i$) of the second position r_2 + gene($child_i$) + gene($child_i$) of the first position r_1 +…+gene($child_i$);
7. **return** {$child_1$, $child_2$};

end

Figure 2.4. Algorithm for Swap Mutation Performer Component

When the two random numbers are 3 and 5 for the first child, then the first child will be:

$Child_1$ = *{2, 5, 6, 1, 7, 8}*

When the two random numbers for the second child are 3 and 4, then second child will be:

$Child_2$ = {7, 1, 5, 2, 6, 8}.

The two offsprings, $child_1$, and $child_2$ are added to the population and any two worst chromosomes are removing from the new population. The iterations are repeated until a certain number of values is reached and return the optimal ordering.

2.3 Illustrative Example

Keeping in mind the end goal of guaranteeing that the mutGA approach works legitimately, we give an illustrative example in this section prior to the useful usage and observational/empirical assessment. The proof has been prepared by brief prioritizing tests for a triangle program written in java program.The mutants executed from the result of mutating is_Right(), is_Isosceles() and Is_Equilateral () methods of Triangle class using *µJava* are 100. The changed statements are S_{25}, S_{25a}, S_{25b}, S_{45}, S_{45a}, S_{45b}, S_{45c}, S_{45d}, S_{45e}, S_{55}, $S_{55a,}$ and S_{55b}. Based on Slicing-performer algorithm, the affected statements for S_{25}, S_{25a} and S_{25b} are S_{26}, $S_{27,}$ and S_{28}, while for S_{45}, S_{45a}, S_{45b}, S_{45c}, S_{45d}, S_{45e} mutants are S_{46}, $S_{47,}$ and S_{48}, and for S_{55}, S_{55a} and S_{55b} are S_{56}, S_{57}, S_{58} and S_{59} statements that control dependent on the changed statements.

The statements S_{90}, S_{93} and S_{95} are the method call statements that called the methods is_Right (), is_Isosceles () and Is_Equilateral() respectively, are also affected statements as the result of the method call. The affected statements are S_{25}, S_{25a}, S_{25b}, S_{26}, S_{27}, S_{29}, $S_{90,}$ S_{45}, S_{45a}, S_{45b}, S_{45c}, S_{45d}, S_{45e}, S_{46}, S_{47}, S_{49}, S_{92}, S_{55}, S_{55a}, S_{55b}, S_{56}, S_{57}, S_{59} and S_{94}.After identifying the affected statements, with the coverage information, Test-case-selector algorithm selects the following test cases:

T’ = {t_1 t_2 t_5 t_6 t_7 t_8} and their affected nodes are:

- T_1 = { S_{55}, S_{55a}, S_{55b}, S_{56}, S_{57}, S_{94}}
- T_2 = {S_{55}, S_{55a}, S_{55b}, S_{59}, S_{94}}
- T_5 = {S_{45}, S_{45a}, S_{45b}, S_{45c}, S_{45d}, S_{45e}, S_{46}, S_{47}, S_{92}}
- T_6 = {S_{45}, S_{45a},S_{45b}, S_{45c}, S_{45d}, S_{45e}, S_{49}, S_{92}}
- T_7 = {S_{25}, S_{25a}, S_{25b}, S_{26}, S_{27}, S_{90}}
- T_8 = {S_{25}, S_{25a}, S_{25b},S_{29}, S_{90}}

The details of the prioritization are:

Encode the Solution. The Test-case-encoder algorithm encodes the selected test cases as the initial solution. The encoded solution of T` = {$t_1$, $t_2$, $t_5$, $t_6$, $t_7$, $t_8$}, is T` = {*1, 2, 5, 6, 7, 8*}.

Initial Population Generation. The Random-population-generator algorithm randomly generates the initial population. The algorithm generates the initial population as the size of the selected test cases as shown in Table 2.1.

Table 2.1. Initial population

S/No.	Chromosome
1	2, 5, 7, 6, 8, 1
2	7, 1, 5, 6, 2, 8

Evaluate the Fitness. The fitness value for each chromosome in the population was calculated using *calcFitness* algorithm. The fitness value of each chromosome in the population was shown in Table 2.2 at column 3 (Fitness value) as computed below:

$Chromosome_1$ = {2, 5, 7, 6, 8, 1}

Compute the fitness of each gene (i.e. {2, 5, 7, 6, 8, 1}) by calculating the fault severity by reducing the severity to 5%, 10%, 50%, and 75% of the initial severity if the fault is already visited by the preceding test case and multiply it by the position of the gene in the chromosome.

Here, we demonstrate using 5%.

$Gene_2$ = {S_{55}, S_{55a}, S_{59}, S_{94}} = 20+20+20+20 = 80*6 = 480 //None of the statements was visited

$Gene_5$ = {S_{45}, S_{45a}, S_{45b}, S_{45c}, S_{45d}, S_{45e}, S_{46}, S_{47}, S_{92}} = 20+20+20+20+20+20+20+20+20 = 180*5 = 900 //None of the statements was visited

$Gene_7$ = {S_{25}, S_{25a}, S_{25b}, S_{26}, S_{27}, S_{90}} = 20+20+20+20+20 +20 = 120*4 = 480 // None of the statements was visited

$Gene_6$ = {S_{45}, S_{45a}, S_{45b}, S_{45c}, S_{45d}, S_{45e}, S_{49}, S_{92}} = 1+1+1+1+1+1+20+1 = 27*3 = 81 // All the statements were visited by the preceding test cases (t_5) except S_{49}

$Gene_8$ = {S_{25}, S_{25a}, S_{25b}, S_{29}, S_{90}} = 1+1+1+20+1 = 24*2 = 48 // All the statements were visited by the preceding test cases (t7) except s29

$Gene_1$ = {S_{55}, S_{55a}, S_{56}, S_{57}, S_{94}} = 1+1+20+20+1 = 43*1 = 43 // All the statements were visited by the preceding test cases (t_2) except S_{56} and S_{57}

The fitness of the chromosome is the sum of the fitness of all the genes in the chromosome. Fitness = 480+900+480+81+48+43 = 2032.

$Chromosome_2$ = {7, 1, 5, 6, 2, 8}

$Gene_7$ = {S_{25}, S_{25a}, S_{25b}, S_{26}, S_{27}, S_{90}} = 20+20+20+20+20 +20 = 120*6 = 720 // None of the statements was visited

$Gene_1$ = {S_{55}, S_{55a}, S_{56}, S_{57}, S_{94}} = 20+20+20+20+20 = 100*5 = 500 //None of the statements was visited

$Gene_5$ = {S_{45}, S_{45a}, S_{45b}, S_{45c}, S_{45d}, S_{45e}, S_{46},S_{47}, S_{92}} = 20+20+20+20+20+20+20+20+20 = 180*4 = 720 //None of the statements was visited

$Gene_6$ = {S_{45}, S_{45a}, S_{45b}, S_{45c}, S_{45d}, S_{45e}, S_{49}, S_{92}} = 1+1+1+1+1+1+20+1 = 27*3 = 81 // All the statements were visited by the preceding test cases (t_5) except S_{49}

$Gene_2$ = {S_{55}, S_{55a}, S_{59}, S_{94}} = 1+1+20+1 = 23*2 = 46 // All the statements were visited by the preceding test cases (t_1) except S_{59}

$Gene_8$ = {S_{25}, S_{25a}, S_{25b}, S_{29}, S_{90}} = 1+1+1+20+1 = 24*1 = 24 // All the statements were visited by the preceding test cases (t_7) except S_{29}

The fitness = 720+500+720+81+46+24 = 2091.

Table 2. 1. Initial population with their fitness values

S/No.	Chromosome	Fitness value (Reduced to 5%)
1	2, 5, 7, 6, 8, 1	480+900+480+81+48+43 = 2032
2	7, 1, 5, 6, 2, 8	720+500+720+81+46+24 = 2091

Selection. The two chromosomes are selected in the population for mutation.

$Chromosome_1$ = C_1= {*2, 5, 7, 6, 8, 1*} and $Chromosome_2$ = C_2 = {*7, 1, 5, 6, 2, 8*}.

Mutation. The Swap-mutation-performeralgorithm generates two random integer numbers between 1 and 6 (*numberOfGenes*), i.e. r_1 and r_2. The genes of these positions will simply swap their genes for the first Chromosome($Chromosome_1$), and the same process is repeated for the second Chromosome ($Chromosome_2$).

$Child_1$ = *swapMutation (*$Chromosome_1$*)*

$Child_2$ = swapMutation ($Chromosome_2$)

Given

$Chromosome_1$= *{2, 5, 7, 6, 8, 1}* and

$Chromosome_2$ = *{7, 1, 5, 6, 2, 8}*

When the two random numbers are 3 and 5 for the first Chromosome, then the first child will be:

$Child_1$ = *{2, 5, 8, 6, 7, 1}*

When the two random numbers for the second Chromosome are 1 and 3, then second child will be:

$Child_2$ = *{5, 1, 7, 6, 2, 8}*

Evaluate the Fitness. Evaluate the fitness of the two offspring $child_1$ and $child_2$ using *calcFitness* algorithm.

Fitness = 480+900+108+129+240+24 = 1881.

The fitness of $child_2$is the sum of the fitness of all the genes in the child:

Fitness = 720+500+720+69+54+24 = 2087.

The replacement criteria. Add the two offspring, $child_1$, and $child_2$ to the population and remove chromosomes 3 and 4 as the two worst chromosomes from the new population.

The termination criteria. Check if the termination condition is not true, then, repeat the stages from selection to termination steps, else, end the process and return the optimal ordering.

After 20 iterations, the prioritize test case is {5, 7, 1, 6, 2, 8} with Fitness value = 2231.

3. Results

The expected results from the proposed prototype will be effective and efficient. As shown illustrative example subsection (Section 2.3), after twenty (20) iterations, the prioritize test case is {5, 7, 1, 6, 2, 8} with Fitness value = 2231, that shows a significant improvement in the fitness value compared to the initial values.

The performance of the proposed approach was evaluated and compared with retest-all and randomly ordered test cases techniques using Average Percentage of rate of Fault Detection (APFD) as in the previous researches (Musa *et al*. 2014a, 2014b, 2015), given as

$$APFD = 1 - \frac{Tf1+Tf2+\cdots+Tfm}{\#N * \#killed\ mutants} + \frac{1}{2*\#N} \quad (1)$$

Tf_1 = Number of faults detected by T_1 * its position

#N = Number of affected test cases

#killed mutants = Total number of mutants killed

The ordering are:

Retest-all = {t_1, t_2, t_5, t_6, t_7, t_8}

Random = {t_2, t_5, t_7, t_6, t_8, t_1}

mutGA = {t_5, t_7, t_1, t_6, t_2, t_8}

Using Equation 1,

APFD for Retest-all =

$$1 - \frac{6*1+1*2+9*3+1*4+6*5+1*6}{6 * 24} + \frac{1}{2*6}$$

= 1 – 75/144 + 0.083

= 1 – 0.521 + 0.083 = 0.479 +0.083 = 0.562

= 56.2%

APFD for Random =

$$1 - \frac{5*1+9*2+6*3+1*4+1*5+2*6}{6 * 24} + \frac{1}{2*6}$$

= 1- 62/144 + 0.083 = 1 – 0.431 + 0.083

= 0.652 = 65.2%

APFD for *mutGA*

$$1 - \frac{9*1+6*2+6*3+1*4+1*5+1*6}{6 * 24} + \frac{1}{2*6}$$

= 1 – 54/144 + 0.083 = 1 – 0.375 + 0.083

= 0.708 = 70.8%

The values of APFD for retest-all and randomly ordered test cases techniques (56.2% and 65.2% respectively) show that the performance of our proposed approach, *mutGA (*70.8%) higher percentage of faults detection. This shows that, if after execution of the first test case, the process halt, our approach will detect nine (9) faults, while retest-all and randomly ordered test cases techniques will detect six (6) and five (5) respectively.

4.0 Conclusion and Future Work

A mutated regression test case prioritization technique that ordered selected test cases T` using GA was proposed. The technique used mutation operation to mutate the test cases, and a reduction in fault severity when a statement was executed by the preceding test cases, named *mutGA*. This technique can be used for regression testing of OOP. A sample example was demonstrated for the *mutGA* as described in Section 2.3. We prioritized a sample Java program with the proposed prototype. The implementation showed the evidence that *mutGA* was feasible to be used in practice.

Although this research has considerably achieved the intended goals as proposed by the technique, there are many possible extensions that can be enhanced in the future. This section presents these possible extensions as future work.

4.1 Fully Implemented and Integrated

*mutGA*used existing tools in some of the phases while others are manually done. The prototype can be expanded to make it fully automated in the future so that all phases of the approach can be integrated as a complete tool. It can be implemented using large codes to test its workability.

4.2 Tool Support for a Multi-language Tool

The *mutGA* tool can be developed using Java, and it can be extended so that it can identify other OOP languages whose syntax is similar to that of Java, e.g. C++, C#. The newly tool extended can be applied to the tool by using existing concepts, which will help in having a broader opportunity and be more beneficial to a widespread of users.

From the illustrated example, it can be concluded that the proposed approach can achieved a better result even without the application of crossover operation in GA-based regression test case prioritization.

References

Athar, M. and Ahmad, I. (2014). Maximize the Code Coverage for Test Suit by Genetic Algorithm. International Journal of Computer Science and Information Technologies, 5: 431-435.

Grottke M., and Trivedi, K. (2007). Fighting Bugs: Remove, Retry, Replicate, IEEE Transactions on Software Engineering. 2: 107-109.

Kiran A, Butt WH, Shaukat A, Farooq MU, Fatima U, Azam F, et al. (2020) Multi-

objective regression test suite optimization using three variants of adaptive neuro fuzzy inference system. PLoS ONE 15(12): e0242708.https://doi.org/10.1371/journal.pone.0242708

Konsaard P., and Ramingwong, L. (2015). Total Coverage Based Regression Test Case Prioritization using Genetic Algorithm. 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, June 2015. IEEE, pp: 1-6.

Musa, S., Sultan, A. B. M., Abd-Ghani, A. A. B., and Baharom, S. (2015). Regression Test Cases selection for Object-Oriented Program based on affected statements. International Journal of Software Engineering and Its Applications. 10: 91-108.

Musa, S., Sultan, A. B. M., Abd-Ghani, A. A. B., and Baharom, S. (2014a). Regression Test Framework based on Extended System Dependence Graph for Object-Oriented Programs. International Journal of Advances in Software Engineering & Research Methodology– IJSERM, 2:28-33.

Musa, S., Sultan, A. B. M., Abd-Ghani, A. A. B., and Baharom, S. (2014b). A Regression Test Case Selection and Prioritization for Object-Oriented Programs using Dependency Graph and Genetic Algorithm,Research Inventy: International Journal of Engineering And Science, 7: 54-64.

Perry, W. 2006. Effective Methods for Software Testing, John Wiley Publication

Purohit, G.N., and Sherry, A.M. (2014). Test suites prioritization for regression testing using genetic algorithm. International Journal of Emerging Technologies in Computational and Applied Sciences. 150: 255-259.

Roger, S.P. (2001). Software Engineering: A practitioner's Strategy, 5th Edn, New York, America, McGraw-Hill, Inc.

Wang, R., Li, Z., Jiang, S. and Tao, C. Regression Test Case Prioritization Based on Fixed Size Candidate Set ART Algorithm. International Journal of Software Engineering and Knowledge EngineeringVol. 30, No. 03, pp. 291-320 (2020).https://doi.org/10.1142/S0218194020500138

Rothermel, G., and Harrold, M.J.(1994).. Selecting regression tests for object-oriented software. International Conference on in Software Maintenance, IEEE, September 1994. Pp: 14-25.

Rothermel, G., Roland, H.U., Chu, C., and Harrold, M.J. (2001). Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, 10: 929-948

Zhang, L., Hou, S. S., Guo, C., Xie, T., and Mei, H. (2009). Time-aware test-case prioritization using integer linear programming. In Proceedings of the eighteenth international symposium on Software testing and analysis. ACM 2009, pp: 213-224.