American Journal of Software Engineering and Applications
2016; 5(2): 7-14
http://www.sciencepublishinggroup.com/j/ajsea
doi: 10.11648/j.ajsea.20160502.11
ISSN: 2327-2473 (Print); ISSN: 2327-249X (Online)

An Improved Genetic Algorithm-Based Test Coverage Analysis for Graphical User Interface Software

Asade Mojeed Adeniyi, Akinola Solomon Olalekan

Department of Computer Science, University of Ibadan, Ibadan, Nigeria

Email address:
princedeniyiasade@hotmail.com (M. A. Asade), akinola.olalekan@dlc.ui.edu.ng (S. O. Akinola)

To cite this article:
Asade Mojeed Adeniyi, Akinola Solomon Olalekan. An Improved Genetic Algorithm-Based Test Coverage Analysis for Graphical User Interface Software. *American Journal of Software Engineering and Applications.* Vol. 5, No. 2, 2016, pp. 7-14.
doi: 10.11648/j.ajsea.20160502.11

Received: January 22, 2016; **Accepted**: February 3, 2016; **Published**: April 6, 2016

Abstract: Quality and reliability of software products can be determined through the amount of testing that is carried out on them. One of the metrics that are often employed in measuring the amount of testing is the coverage analysis or adequacy ratio. In the proposed optimized basic Genetic Algorithm (GA) approach, a concept of adaptive mutation was introduced into the basic GA in order for low-fitness chromosomes to have an increased probability of mutation, thereby enhancing their role in the search to produce more efficient search. The main purpose of this concept is to decrease the chance of disrupting a high-fitness chromosome and to have the best exploitation of the exploratory role of low-fitness chromosome. The study reveals that the optimized basic GA improves significantly the adequacy ratio or coverage analysis value for Graphical User Interface (GUI) software test over the existing non-adaptive mutation basic GA.

Keywords: Software Test Coverage Analysis, Graphical User Interface, Quality Software, Genetic Algorithm

1. Introduction

Graphical User Interface (GUI) is a means of interaction between an end user and a software system. Software systems have gained an unprecedented popularity for so long and the biggest factor behind this success is Graphical user interface. Software developing companies and developers have always shown a desire for fully assured high quality software. In order to ensure this desire is fulfilled, software must go through a comprehensive testing. It seems almost impossible to test GUI application manually because of the involved complexity thereby creating the need for test data automation [1].

The quality of software products is of paramount importance to users. According to Pfleeger [2], quality software is *"Software that satisfies the needs of the users and the programmers involved in it"*. Technological advancements have been responsible for the complex nature of computer and in particular the software that drives it. Based on this, the correctness of the software is of high importance which cannot even be guaranteed by the developer that designs the software.

Chayanika *et al*., [3] assert that the main purpose of software industry is to ensure that the software delivered to the end users is of high standard. Testing of software therefore cannot be overemphasized as this plays a major role in deciding the quality and reliability of the delivered software as well as ensuring the software meets the users' requirements.

Our everyday life dependence on computer; be it mobile, home appliances or office has rather placed an importance on software testing since we cannot afford to let the system fail us. Glenford [4] views testing as a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended. Software should be predictable and consistent, offering no surprises to users. Software testing is more properly viewed as the destructive process of trying to find the errors (whose presence is assumed) in software. Glenford [4] opined further that a successful test case is one that furthers progress in this direction by causing the software to fail.

In reality, planning for software testing should begin with the early stages of the software requirements process, and test plans and procedures should be systematically and

continuously developed (which could possibly be refined) as software development proceeds [5]. This is so because the test planning and designing activities serve as a clue to software designers/developers by assisting to highlight likely weaknesses, such as conflicts or oversights /contradictions, or ambiguities in the [5].

Coverage analysis can simply be defined as a measure of test case completeness. It can further be inferred as a means of determining how much test needs to be conducted in order to ascertain the quality of the developed software. Test coverage analysis can be done in order to determine test effectiveness, test suit improvement and software reliability estimation. Coverage is the extent to which a structure has been exercised as a percentage of the items being covered [6]. According to Muhammad *et al*. [6], test coverage is regarded as a key indicator of software quality and a crucial part of software maintenance which assists in carrying out the efficacy of testing through the provision of data on diverse coverage items. Test coverage is an indicator that gives insight to the test generators to focus on creating test cases that cover the areas that have not been tested.

Genetic Algorithms (GAs) have been applied to a broad range of searching, optimization, and machine learning problems. GAs are iterative procedures implemented as a computerized search and optimization procedure that uses principles of natural selection. It performs a multi- directional search by maintaining a population of potential solutions (called individuals) and exchange information between these solutions through simulated evolution and forward relatively "good" information over generations until it finds a near optimal solution for specific problem [7]. More often than not, GA's converge rapidly to quality solutions. Although they do not guarantee convergence to the single best solution to the problem, the processing power associated with GA's makes them efficient search techniques [8].

There have been a number of studies that employ genetic algorithms for software testing. In this study, a concept of adaptive mutation was introduced into the basic genetic algorithm in order for low-fitness chromosomes to have an increased probability of mutation, thereby enhancing their role in the search to produce more efficient search. Section 2 of this paper highlights some related works while Section 3 gives the methodology for the study. In Section 4, results are presented and discussed and conclusion is drawn in Section 5.

2. Related Works

Memon, [9] in his PhD research focused on developing a testing framework for Graphical User interface (GUI) which covers the areas of testing environments, test coverage criteria development, test case generation, test oracles and regression testing. Research at that time on GUI was still at its infancy. So he had to adapt techniques from general software testing for GUI testing. The GUIs are differentiated from the traditional software with some characteristics like user events for input and graphical output and thus require different testing techniques.

Capture /Replay is a popular method that is being used for GUI software testing. One merit offered by this method is that it is able to determine test cases that are usable and unusable in a situation that the states of the GUI is modified, and furthermore determines which of the unusable test cases can be repaired and make it usable for the modified graphical user interface. This attribute made it usable for regression testing [10]. The observed laps with the Capture / Replay method is that it does not provide functionality to evaluate the coverage of a test suite mainly because it does not have a global view of the GUI.

Misurda, *et al*., [11] described a demand-driven framework for program testing having scalability and flexibility features. It makes use of test paths for the implementation of test coverage. This framework also has a means of ensuring performance and memory overheads are kept low through the use of dynamic instrumentation on the binary code which can be inserted and removed as at when required.

Matteo *et al.,* [12] in their own work proposed a framework called Covertures which is a virtualized execution platform meant for cross-compiled application on the host. This framework has the ability to carry out measurement of structural coverage of both object and source code without application instrumentation.

Sakamoto *et al*., [13] proposed a framework for consistent and flexible measurement of test coverage. This framework has support for multiple programming languages and also provides guidelines for the support of several test coverage criteria. The flexibility attribute of this framework is that it allows for the inclusion of user defined test coverage and new programming language. This framework is called Open Code Coverage Framework (OCCF).

Even though, models are costly to create and have a limited applicability, approaches based on modeling have been employed often in carrying out testing of software. In view of this fact, model based approaches are not being employed frequently for testing GUI software [14].

A Genetic Algorithm (GA) can be defined as a problem-solving approach that is developed as a programming technique by imitating the theory of natural evolution of species as proposed by Charles Darwin. In solving a specific problem using the genetic algorithm concept, it begins with a set of individuals (solution candidates) that forms a population which is generated in a random way. The genetic algorithm now selects parents from which to generate offspring by using reproductive operators that are analogous to biological processes mainly crossover and mutation [15]. The resulting chromosomes are then evaluated using a fitness function in order to determine how strong they are, the fitness values are then employed in taking a decision on which chromosome to be eliminated or retained [15].

In order to generate a new set of solution candidates (new population), the chromosomes having a high fitness value are retained while those that are not fit are discarded. Afterwards, a check is carried out to determine an individual in the population that connects the same two points as the

newly generated individual. If not exist, new individual is added to the population and if yes, the old individual is replaced by the new one if its fit value is higher. These variation and selection steps are repeated until a termination condition is met [16].

A genetic algorithm is especially appropriate to the solution of indefinite problems or non-linear complex problems [17]. Jones *et al*. [18, 19] proposed a technique to generate test-data for branch coverage using genetic algorithm. The technique revealed good results with number of small programs. Though Control Flow Graph (CFG) was employed in guiding the search; they based the fitness value on branch value and branching condition. In another study by Michael *et al*. [20], a tool for generating test data was developed using four different algorithms, two of which are genetic algorithm. This tool is called Gadget. With Gadget, they were able to obtain good condition / decision coverage of C/ C++ code.

Pargas *et al*. [21] also made use of genetic algorithm using the Control Dependence Graph (CDG) to search for test data that will give good coverage. They did a comparison of their system with random testing using six C programs of varying sizes. The outcome of their experiment revealed no difference with the smallest programs but the genetic algorithm based method gave a better performance for the three largest programs.

Shunkun *et al*, [22] capitalized on the pitfalls in the traditional Ant Colony Optimization algorithm for test cases generation in software testing engineering. Some of the flaws are relative scarcity of early search pheromone, low search efficacy and simplicity of the search model. They came up with three improved Ant colony algorithm which are now integrated to form A Comprehensive Improved Ant Colony Optimization (ACIACO), and they were able to generate higher coverage results.

Abdul Rauf *et al*., [14] proposed a system for GUI testing and coverage analysis based on traditional genetic algorithm. Their method is subdivided into three major blocks; Test data generation, path coverage analysis and optimization of test paths. The proposed system made use of traditional genetic algorithm for the optimization of test paths.

The present study considers an improvement over the GUI testing coverage analysis by Abdul Rauf *et al*., [14]. They made use of the basic Genetic Algorithm to optimize the test paths, but here, we modify the basic Genetic Algorithm to optimize the test paths with the hope of achieving a higher coverage analysis than what they obtained.

3. Research Methodology

3.1. Experimental Approach

A GUI is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. A GUI contains graphical objects; each object has a fixed set of properties [14].

To test GUI and analyse the coverage, the proposed methodology was divided into three major blocks listed below as earlier suggested by Abdul Rauf *et al*., [14].

i. Test data generation.
ii. Path Coverage Analysis.
iii. Optimization of Test Paths.

The test data generation is a set of events that were generated from the application that was used for the experiment. This was generated manually by carrying out several test cases on the application to be used for the GUI test while keeping the event identities (ids) being generated in a text file. These event ids were then arranged to determine the path coverage analysis, which is the second block in the methodology being employed.

This study employed the event flow graph (EFG) technique of the GUI test. A User defined calculator was built in C# programming language. This user defined calculator had an in-built instrumentation code that logs parameters like the event_id (widget id), button_name etc. as the application is being interacted with. Another application used in testing our methodology was a user defined Notepad that was developed with Java programming language. It also had an in-built instrumentation code that logs parameters like the event_id (widget id), button_name etc. as the application is being interacted with. Thereafter, we extended our testing to Microsoft (MS) Notepad application.

3.2. Fitness Function Evaluation

Given an input to a program, the fitness function returns a number that indicates the acceptability of the program. The selection algorithm uses the fitness function to determine which variants survive to the next iteration, and this is employed as a termination criterion for the search. In this work, our fitness function was based on how much test cases were successfully validated in line with Abdul *et al*., [14].

Fitness function is hereby defined as Test paths covered by chromosome divided by the total number of test paths i.e.

$$\text{Fitness} = \frac{\text{Test paths covered by chromosome}}{\text{Total number of test paths}}$$ (Abdul et al., [14]).

3.3. The Modified Reproduction Operation

There are basically two reproduction operators in genetic algorithm: Crossover and Mutation. In this work, the reproduction operators were employed in order to increase the coverage efficiency. However, this work is capitalizing on the pitfalls of the basic genetic algorithm in the area of reproduction operator known as Mutation. In the basic genetic algorithm, there is equal application of mutation operator which can as well be referred to as total randomness of mutation irrespective of their fitness. The implication of this action is that a very good chromosome (chromosome of high fitness) is equally likely to be disrupted by mutation as a bad one. Though we know that bad chromosome are less likely to produce good ones through crossover due to their lack of building blocks.

After the crossover operation has been performed we

introduced the evaluation of the mean fitness of the chromosome; thereafter making a comparison of each chromosome to the mean fitness value. The chromosomes having fitness greater than or equal to the mean fitness were made to join the new population without passing through mutation exercise while those with fitness value below the mean fitness value were made to pass through the mutation exercise so that they can benefit most from the operation. This process continues until the termination criterion is met and the whole process comes to a halt and the result is displayed. Figure 1 shows the design and execution flow of basic GA [8]; while figure 2 shows the proposed modified GA Algorithm for optimization of paths.

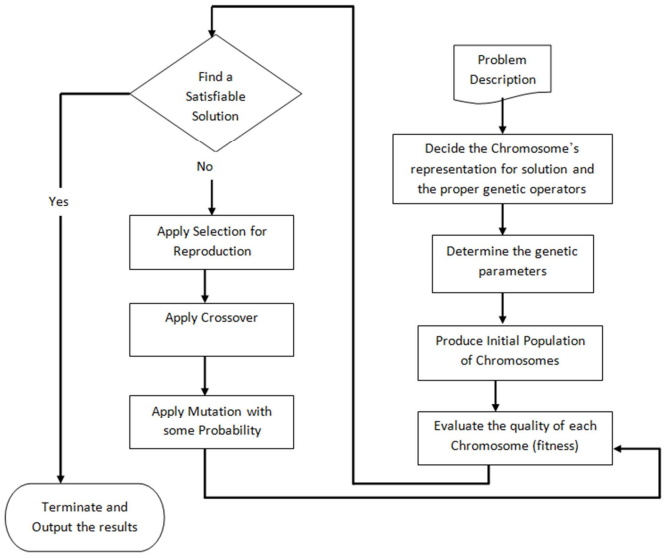

Figure 1. *Basic Genetic Algorithms - Design and Execution Flow (Samarah, 2006).*

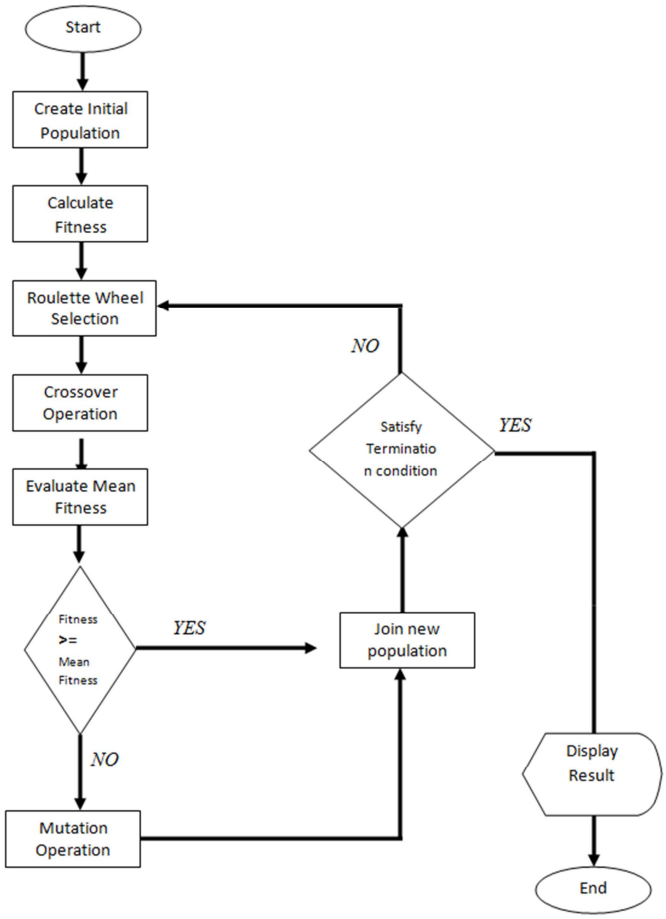

Figure 2. *Execution flow of the proposed Optimized Genetic Algorithm Method for Optimization of Paths.*

Figure 3 represents sample test cases for the calculator that was used to test the project while Figure 4 represents the widgets on the calculator with corresponding labels for each of the widgets. For instance, if we pick (the last entry in Figure 3) 4, 5, 8, 13, 5, 17 it implies 458 divide (13) equals (17).

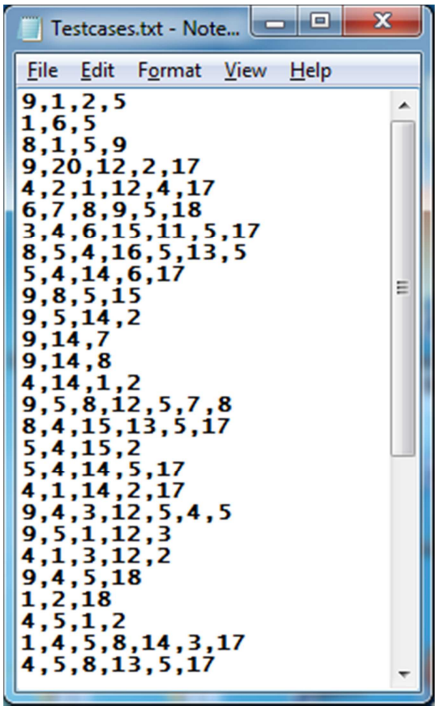

Figure 3. *Sample Test Cases for Calculator.*

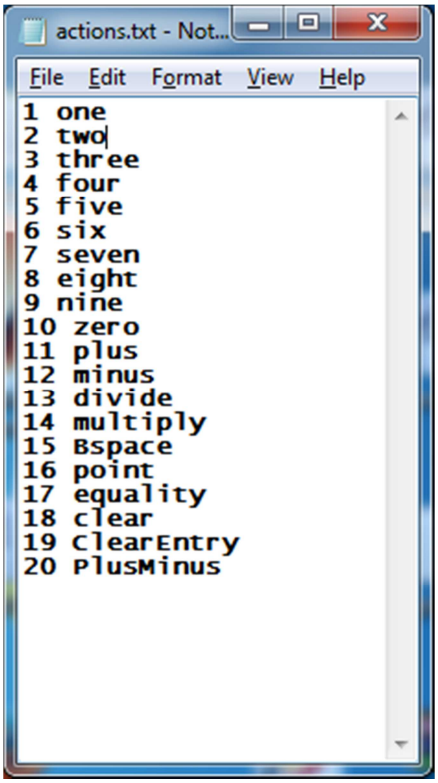

Figure 4. *Representations of the Calculator Widgets.*

3.4. Formation of Chromosomes

An Actions File was used to denote what each of the paths in the test cases file represents for ease of transformation from numerical value to the widget name. The test case having the longest length determines the length of the chromosomes to be generated. For instance, if the longest test case is having a length 10, then the length of the chromosomes will be 10. From Figure 3, the longest test case is having length 7, if that test case file is used for the experiment, the chromosomes to be formed will be of length 7 i.e. consists of 7 genomes.

3.5. Software Tools

Earlier works in GUI software testing have explored several software packages for the execution of their experiments based on their proposed approaches. Some of the available packages that have been used are GUITAR (Graphical User Interface Testing fRamework) GUI Ripper (This is meant for reverse engineering), PATHS (Planning Assisted Tester for grapHical user interface Systems), C++, Java, C# (C-Sharp) and MATLAB (MATrix LABoratory). Software interfaces to be tested are sometimes written in C++, Java or C-Sharp programming languages. The tool for the optimization of test paths in our proposed approach was developed using the Java Programming Language because of its comprehensive and powerful exploration capabilities.

3.6. Test Data Generation

The use of events to produce data for the testing of GUI software has become a common practice since the software is characterized by states. The technique for the test data generation is based on events. We made use of user-defined calculator, user-defined notepad application and we extended it to an off –the-shelf MS Notepad application. The interface of our Calculator and user defined Notepad is shown in Figures 5 and 6 respectively. As event takes place, the event ids as well as the widget get stored into a notepad from where they will be picked up for further analysis. This approach made the path coverage analysis to be carried out easily.

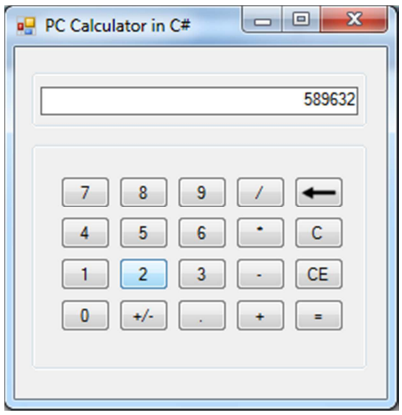

Figure 5. *Interface of calculator application.*

Figure 7 represents the internal labels that were used to represent each of the calculator widgets within the program for the calculator for ease of logging in order to know the particular calculator button that is pressed.

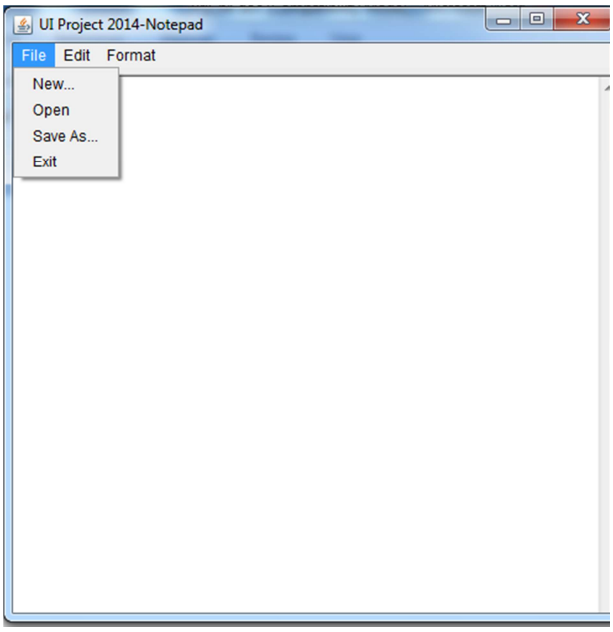

Figure 6. *Interface of User Defined Notepad.*

+/- button: 3005	**Backspace button:** 1003	**0:**2000	/:3004	**1:**2001
CE button: 1000	**2:**2002	**+:**3001	**3:**2003	-:3002
*:3003	**C button:** 1001	**4:**2004	=:3000	**5:**2005
6:2006	**Point button:** 2010	7:2007	**8:**2008	**9:**2009

Figure 7. *Event ID's of Calculator application.*

Path coverage (%) = ((no. of paths Covered) / (total no. of independent paths)) × 100

2009	3004	2003	2009	3003	2003	2005	3001
2008	3002	2003	3000	2007	1001	2005	1000

Figure 8. *Sample of Sequence of Generated Events.*

Figure 8 is a set of sequence of events generated while using the calculator. The numbers displayed are the internal labels that were used to represent each of the widgets in the calculator. For instance, looking at Figure 7 that displays 2009, 3004, 2003, 2009, it implies the following widgets on the calculator 9, /, 3, 9 were pressed. The formula adopted for calculating the Coverage is as follows:

4. Experimental Results

Table 1 gives a summary of the details of the parameters that were used during experimental run. The initial population was set to 100 while the number of generations ranges between 300 and 500 at a step of 25. The crossover probability was set to 0.88 while the mutation probability was set to 0.03. The termination criteria was used to halt each run of the experiment either when the coverage achieved is 88% or the number of generation reached the set threshold.

Table 1. *Parameters Used.*

Parameters	Values
Population	100
Number of generations	300-500
Mutation Probability	0.03
Crossover Probability	0.88
Termination criteria	Coverage >88% or Generation = 500

Table 2 shows the results of the coverage achieved for each of the three applications that were used with the Basic Genetic Algorithm as well as the average coverage per generation with generation ranging between 300 and 500 at a step of 25. The highest coverage obtained for Ms-Notepad, User Defined Notepad and User Defined Calculator at 500 generations were 85%, 87.67% and 71.43% respectively which are also in line with what Abdul *et al.*, (2010) obtained using the basic Genetic Algorithm except for calculator that was slightly higher. The average coverage achieved with the basic genetic algorithm after 500 generations was 81.37%.

Table 2. *Coverage with respect to number of generations using Basic Genetic Algorithm.*

Number of Generations	MS Notepad	User Defined Notepad	Calculator Coverage	Av. Coverage
300	68.00%	73.33%	50.00%	63.78%
325	72.00%	73.33%	57.14%	67.49%
350	76.00%	76.33%	57.14%	69.82%
375	76.00%	76.67%	64.29%	72.32%
400	76.00%	83.33%	64.29%	74.54%
425	76.00%	84.67%	71.43%	77.37%
450	80.00%	86.67%	71.43%	79.37%
475	84.00%	87.67%	71.43%	81.03%
500	85.00%	87.67%	71.43%	81.37%

Table 3. *Coverage with respect to number of generations using Optimized Basic Genetic Algorithm.*

Number of Generations	MS Notepad	User Defined Notepad	Calculator Coverage	Average Coverage
300	72.00%	80.00%	57.14%	69.71%
325	76.00%	80.00%	57.14%	71.05%
350	84.00%	83.33%	64.29%	77.21%
375	84.00%	86.67%	64.29%	78.32%
400	84.00%	86.67%	64.29%	78.32%
425	84.00%	86.67%	64.29%	78.32%
450	88.00%	88.67%	72.43%	83.03%
475	88.00%	90.00%	72.43%	83.48%
500	92.00%	90.00%	73.43%	85.14%

Table 3 displays the results of the coverage achieved using the Modified Basic Genetic Algorithm on the same data set as the basic genetic algorithm for each of the three applications that were used for the experiment. The same numbers of generations were used and the average coverage achieved was 85.14% at 500 generations starting from 300 at a step of 25. This result shows a significant improvement over that of basic genetic algorithm that gave us an average of 81.37%. However, looking at the obtained coverage for

MS-Notepad application at 500 generations, the obtained coverage of 92% is higher than what the Abdul *et al*., (2010) obtained with the basic genetic algorithm. This is an indication of better performance with our proposed methodology.

Table 4 highlights a comparison of the average coverage of the basic and the proposed (modified) Gas.

The results obtained for the proposed methodology from our experiment reveals some significant improvements over the results obtained from the benchmarked methodology. The comparisons of both methodologies are shown in Figures 9, 10 and 11 via graphical charts for each application with an average coverage of 81.37% and 85.14% for Basic Genetic Algorithm and Optimized Genetic Algorithm respectively.

Table 4. *Comparison of Basic GA and Optimized GA on Average Test Path Coverage.*

Number of Generations	Basic G. A	Optimized G. A
300	63.78%	69.71%
325	67.49%	71.05%
350	69.82%	77.21%
375	72.32%	78.32%
400	74.54%	78.32%
425	77.37%	78.32%
450	79.37%	83.03%
475	81.03%	83.48%
500	81.37%	85.14%

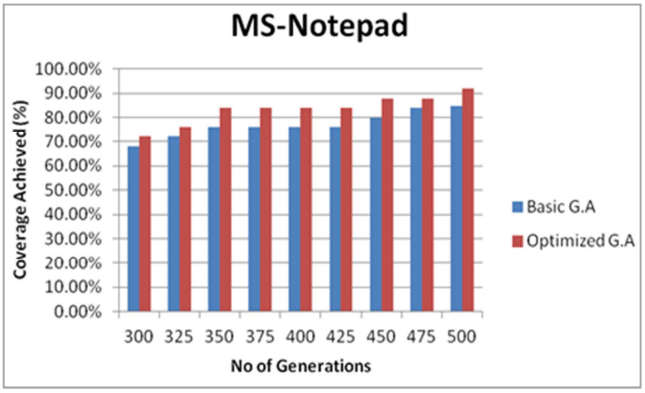

Figure 9. *Comparison of basic and optimized GAs on test path coverage for MS-Notepad.*

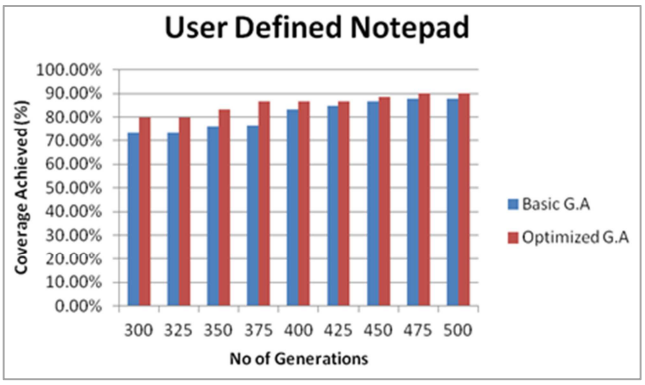

Figure 10. *Comparison of basic and optimized GAs on test path coverage for User-defined Notepad.*

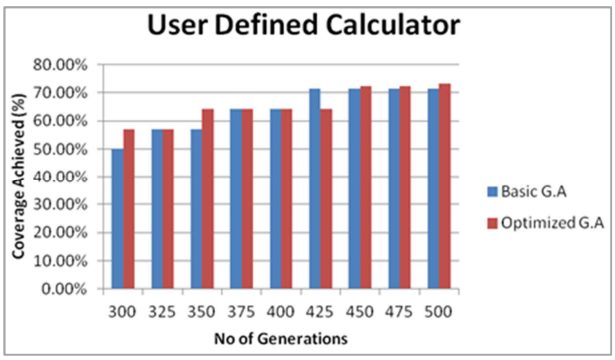

Figure 11. *Comparison of basic and optimized GAs on test path coverage for user-defined Calculator.*

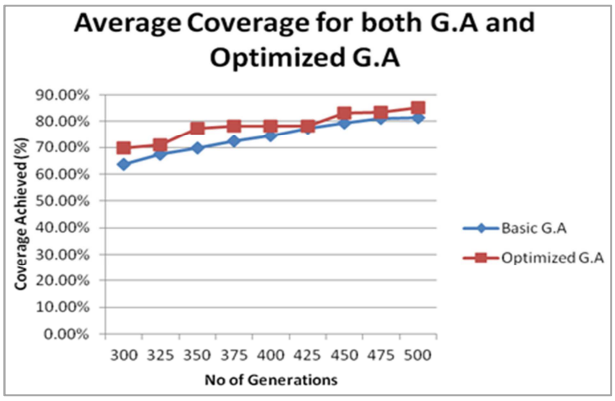

Figure 12. *Comparison of basic and optimized GAs based on Average Path Coverage for all the applications.*

The results reveal that an increase in the number of generations is directly proportional to an increase in percent coverage. The study reveals that the Optimized Basic Genetic Algorithm produces better results than the Basic Genetic Algorithm. The overall average coverage achieved for both methodologies is shown graphically in Figure 12.

The implication of the achieved coverage is that there is still need for more testing to be carried out, which is an indication for the test case generators to focus on the area that have not been tested and generate more test cases from there. By this, we can be rest assured of the quality and reliability of the software to be delivered.

5. Conclusion

From the obtained result in this study, it is hereby concluded that the optimized Genetic Algorithm improves significantly the Adequacy Ratio or Coverage Analysis value for GUI software test over the existing non-adaptive mutation basic Genetic Algorithm.

References

[1] Abdul R., Aleisa E. and Bakhsh I. (2013). GUI Test Coverage Analysis using NSGA II, *The Proceeding Of International Conference on Soft Computing and Software Engineering* [SCSE'13], San Francisco State University, CA, U. S. A., March 2013.

[2] Pfleeger S. L. (2001). Software Engineering Theory and Practice, Prentice Hall.

[3] Chayanika S., Sangeeta S. and Ritu S. (2013). A Survey on Software Testing Techniques using Genetic Algorithm, *International Journal of Computer Science Issues*, Vol. 10, Issue 1, No 1, January 2013.

[4] Glenford J. M. (2004). The Art of software Testing, Second Edition, Revised and Updated by Tom Badgett and Todd M. Thomas with Corey Sandler**,** John Wiley & Sons, Inc. 2004.

[5] Pierre B. and Richard E. F. (2014). Guide to the Software Engineering Body of Knowledge V3.0, A project of IEEE computer Society 2014.

[6] Muhammad S., Suhaimi I. and Mohd N. M. (2011). A Study on Test Coverage in Software Testing, *International conference on Telecommunication Technology and Applications*, Proc of CSIT Vol. 5 IACSIT Press, Singapore, 2011.

[7] Michalewics Z. (1992). Genetic Algorithm + Data Structures = Evolution Programs, Springer, 1992.

[8] Samarah Amer (2006). Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm (An Unpublished publication).

[9] Memon A. M. (2001). A Comprehensive Framework for Testing Graphical User Interfaces, Ph. D. Thesis, University of Pittsburgh, Pittsburg, PA.

[10] Memon A. M. and Soffa M. (2003). Regression Testing of GUI's, *Proceeding of European Software Engineering Conference* /FSE'03. Sep. 2003.

[11] Misurda J., Clause J. A., *et al*. (2005). Demand-driven structural testing with Dynamic instrumentation, In *Proceedings of 27th International Conference on Software Engineering,* 2005 ICSE 2005, pp. 165-175.

[12] Matteo B, Cyrille C., Tristan G., Jerome G., Thomas Q. and Olivier H. *et al*., (2009). Covertures: an Innovative Open Framework for code coverage analysis of safety critical applications, Covertures Open Repository at Open-DO.org, http://forge.open-do.org/projects/couverture.

[13] Sakamoto K., Washizaki H., et al. (2010). Open Code Coverage Framework: A Consistent and Flexible Framework for Measuring Test Coverage Supporting Multiple Programming Languages, *10th International Conference on Quality Software, QSIC*, 2010, pp. 262-269.

[14] Abdul R., Arfan, J. and Arshad A. S (2010). Fully Automated GUI test coverage analysis using GA, *Seventh International conference on information technology.* IEEE 2010, pp. 1057-1063.

[15] Wen-Yang L., Wen-Yuan L. and Tzung-Pei H (2003). Adapting Crossover and Mutation in Genetic Algorithms" *Journal of Information Science and Engineering,* 19, 889-903 2003.

[16] Benjamin D., Edda H. and Christian K. (2008). Crossover Can Provably be Useful in Evolutionary Computation.

[17] Wu, Y., Ji P. and Wang T. (2008). An empirical study of a pure genetic algorithm to solve the capacitated vehicle routing problem, *ICIC Express Letters*, Vol. 2, No. 1, pp. 41-45, 2008.

[18] Jones B. F., Sthamer H. H. and D. E. Eyres (1996). Automatic structural testing using Genetic algorithms, *The Software Engineering Journal*, Vol. 11, No. 5, pp. 299-306, 1996.

[19] Jones, B. F., Eyres D. E. and Sthamer H. H., (1998). A Strategy for using genetic algorithms to automate branch and fault-based testing, *The Computer Journal,* Vol. 41, No. 2, pp. 98-107, 1998.

[20] Michael C. C., McGraw G. and Schatz M. A. (2001). Generating software test data by evolution, *IEEE Transactions on Software Engineering*, Vol. 27, No. 12, pp.1 085-1110.

[21] Pargas R., Harrold M. J. and Peck R. (2008). Test-data generation using genetic algorithms, Journal of Software Testing, Verification and reliability, *Science and Software Engineering*, Vol. 9, No. 4.

[22] Shunkun Y., Tianlong M. and Jiaqi X. (2014). Improved Ant Algorithms for Software Testing Cases Generation, the *Scientific World Journal* Vol. 2014, Article ID 392309, 9 pages Hindawi Publishing Corporation.