AUTOMATED MUTATION-BASED TEST DATA GENERATION: GENETIC ALGORITHM GAME-LIKE APPROACH

BY

HAYATULLAHI BOLAJI ADEYEMO

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

SOFTWARE ENGINEERING

MARCH 2018

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **HAYATULLAHI BOLAJI ADEYEMO** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SOFTWARE ENGINEERING**.

[illegible]

Dr. Moataz A. Ahmed
(Advisor)

[illegible]

Dr. Khalid Al-Jasser
Department Chairman

[illegible]

Dr. Jameleddine Hassine
(Member)

[illegible]

Dr. Salam A. Zummo
Dean of Graduate Studies

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

[illegible]

Dr. Sajjad Mahmood
(Member)

١/١١/١٨

Date

I dedicate this thesis to my parents and my late maternal grandmother

ACKNOWLEDGMENTS

All praise which is abundant, excellent and blessed is due to Allah (SWT), The Most Compassionate, The Most Merciful. To Him belongs all praise and thanks in its entirety. I thank Allah for everything.

Then I thank my parents for their love, moral, emotional and financial support.

I also thoughtfully thank my advisor, Dr. Moataz Ahmed, whom under his guidance, constructive criticisms, and gentle pushes I am able to complete this research work. I thank him for his inspiration, advice and encouragement. His jokes on concepts like ‘No pain, no gain’, ‘No free lunch’ had really kept me striving to proceed in this research, when solving some problems in this research seems like unachievable. I also show my sincere recognition to my thesis committee members Dr. Jameleddine Hassine and Dr. Sajjad Mahmood who taught me some of the foundational courses upon which this research was built.

I will not forget the effort of my wonderful life-companion, soul mate and consort for her affection, understanding, encouragement, support and prayers. She is indeed a friend in need.

My acknowledgment would be incomplete if I fail to recognize the effort of Dr. Muhammad Sulayman of Graduate Studies, King Fahd University of Petroleum and Minerals, Kingdom of Saudi Arabia. I am also indebted to the Nigerian Community at KFUPM for making this environment a home abroad for me. I thank my friends and those who have assisted in one way or the other. To you all, I say Jazaakumullaahu khayran.

Lastly, all praise and perfection is due to Allah, The Most Gracious, The Most Merciful.

TABLE OF CONTENTS

ACKNOWLEDGMENTS v
TABLE OF CONTENTS vi
LIST OF TABLES viii
LIST OF FIGURES ix
LIST OF ABBREVIATIONS xi
ABSTRACT xii
ملخص الرسالة xiv
CHAPTER 1 INTRODUCTION 1
1.1 Problem Statement 2
1.2 Major Contributions 3
1.3 Organization of the Thesis 4
CHAPTER 2 BACKGROUND 5
2.1 Software Testing 5
2.2 Mutation Testing 8
2.2.1 Mutation Operators 15
2.2.2 Problems of Mutation Testing 18
2.3 Genetic Algorithms 19
2.3.1 Biological Background of Genetic Algorithms 19
2.3.2 Details of Genetic Algorithms 21
2.4 Test Data Generation 28
2.4.1 Test Case Design for Black Box Testing 30
2.4.2 Test Case Design for White Box Testing 31
2.4.3 Test Data Generation Using Genetic Algorithms 31
CHAPTER 3 LITERATURE REVIEW 34
3.1 Genetic Algorithms Based Test Case Generation 34
3.2 Mutation-Based Test Case Generation 37
3.3 Research Questions 44
CHAPTER 4 PROPOSED APPROACH 46
4.1 Methodology/Approach 46
4.2 Mutant Fitness Function 51
4.3 Test Fitness Function 56
4.4 Mutants Generation 60

4.5 Mutant Program Generation 62
4.6 Test Case Generation 64
4.7 Selecting The GA Parameters 65
CHAPTER 5 EXPERIMENTS, RESULTS AND DISCUSSION 71
5.1 Experiment Design 71
5.2 Description of Programs Under Test 72
5.3 Results and Discussion 74
5.4 Parameter Selections for the GAs 74
5.5 Discussion of the results of Experiment 78
5.5.1. QuadraticSolver 78
5.5.2. TriangleType 85
5.5.3. MID 89
5.5.4. LineRectangleClassifier 91
5.5.5. PointCircleClassifier 94
5.6 Confidence Interval 97
5.7 Answering Research Questions 99
5.8 Hard to Kill Mutants 101
CHAPTER 6 CONCLUSION AND FUTURE WORK 103
6.1 Main Contributions of the Study 103
6.2 Limitations of the Study 103
6.3 Threats to Validity 104
6.4 Future Work 105
REFERENCES 107
Appendix A_Experimental Data 118
Appendix B Codes of Programs under Test 120
Appendix C Theses/Dissertations on Mutation Testing 122
Vitae 127

LIST OF TABLES

Table 1: Java Class-level mutation operators 17
Table 2: Chromosome representation and interpretation 24
Table 3: Summary of Mutation-based Test Case Generation 42
Table 4: Initial Probability of operators 52
Table 5: Updated Probability of operators 52
Table 6: Probability of program line number 55
Table 7: Test Execution Matrix 56
Table 8: Updated test execution matrix 57
Table 9: Score of test cases 58
Table 10: Sorted test cases with respect to their killing power 58
Table 11: Valid mutants for QuadraticSolver program 61
Table 12: Tester GA Parameters and Levels 67
Table 13: Mutation GA Parameters and Levels 67
Table 14: Experimental Design for Tester GA Parameter Selection 68
Table 15: Experimental Design for Mutator GA Parameter Selection 69
Table 16: Description of Programs under Test 72
Table 17: Selected parameters for tester GA 76
Table 18: Results of experiment to select the best parameter set for mGA 76
Table 19: Selected parameters for mutant GA 78
Table 20: Fitness of tGA of each subject program over 32 runs 98
Table 21: Results of RQ3 101
Table 22: Summary of Master's and PhD theses on mutation testing 122

LIST OF FIGURES

Figure 1: Mutation Testing Process Flow chart (from [10]) ... 9
Figure 2: Examples of mutants ... 11
Figure 3: Basic Genetic Algorithm ... 23
Figure 4: The Genetic Algorithm with Elitism (from [28]) ... 25
Figure 5: Flowchart of mutant generation and analysis ... 48
Figure 6: Flowchart for mutants conversion ... 49
Figure 7: Representation of mutant chromosome ... 60
Figure 8: Representation of test chromosome ... 65
Figure 9: Plot of Tester GA Confidence Interval for parameters selection ... 75
Figure 10: Plot of mGA Confidence Interval for parameters selection ... 77
Figure 11: Total and Killed mutants for QuadraticSolver using 100 generations ... 79
Figure 12:Killed mutants and unique mutants for QuadraticSolver (100 Generations) ... 79
Figure 13: Killed mutants and unique mutants for QuadraticSolver (150 Generations) ... 80
Figure 14: Number of unique tested mutants for QuadraticSolver (150 Generations) ... 81
Figure 15: Killed mutants and unique mutants for QuadraticSolver (200 Generations) ... 81
Figure 16: Number of unique tested mutants for QuadraticSolver (200 Generations) ... 82
Figure 17: Killed mutants and unique mutants for QuadraticSolver (250 Generations tGA-mGA) .. 82
Figure 18: Killed mutants and unique mutants with randomly generated mutants for QuadraticSolver (250 Generations) ... 83
Figure 19: Number of unique tested mutants (randomly generated) for QuadraticSolver (250 Generations) ... 84
Figure 20: Killed mutants and unique mutants with randomly generated tests for QuadraticSolver (250 Generations) ... 84
Figure 21: Killed mutants and unique mutants with random mutants and tests for QuadraticSolver (250 Generations) ... 85
Figure 22: Killed mutants and unique mutants for TriangleType (250 Generations tGA-mGA) ... 86
Figure 23: Number of unique tested mutants (randomly generated) for Triangle (300 Generations) 87
Figure 24: Number of unique tested mutants with randomly generated tests for Triangle (250 Generations) ... 88
Figure 25: Killed mutants and unique mutants with random mutants and tests for TriangleType (250 Generations) ... 88
Figure 26: Killed mutants and unique mutants for MID (250 Generations tGA-mGA) ... 89
Figure 27: Number of unique tested mutants (randomly generated) for MID (250 Generations) ... 90
Figure 28: Killed mutants and unique mutants with randomly generated tests for MID (250 Generations) ... 90
Figure 29: Killed mutants and unique mutants with random mutants and tests for MID (250 Generations) ... 91
Figure 30: Killed mutants and unique mutants for LineRectangleClassifier (250 Generations tGA-mGA) ... 92
Figure 31: Number of unique tested mutants (randomly generated) for LineRectangleClassifier (250 Generations) ... 92
Figure 32: Killed mutants and unique mutants with randomly generated tests for LineRectangleClassifier (250 Generations) ... 93

Figure 33: Killed mutants and unique mutants with random mutants and tests for LineRectangleClassifier (250 Generations) 94
Figure 34: Killed mutants and unique mutants for PointCircleClassifier (250 Generations tGA-mGA) 94
Figure 35: Number of unique tested mutants (randomly generated) for PointCircleClassifier (250 Generations) 95
Figure 36: Killed mutants and unique mutants with randomly generated tests for PointCircleClassifier (250 Generations) 96
Figure 37: Killed mutants and unique mutants with random mutants and tests for PointCircleClassifier (250 Generations) 97
Figure 38: Confidence Intervals for 32 runs of the experiment on the subject programs 99
Figure 39: Roots of quadratic equation 118
Figure 40: Results of experiment to select the best parameter set for tGA 126

LIST OF ABBREVIATIONS

GA	Genetic Algorithm
mGA	mutator Genetic Algorithm
tGA	tester Genetic Algorithm
AOR	Arithmetic Operator Replacement
ROR	Relational Operator Replacement
GUI	Graphical User Interface
RQs	Research Questions
MATLAB	MATrix LABoratory
IEEE	Institute of Electrical and Electronics Engineers
API	Application Programing Interface
RIP	Reachability Infection Propagation
SDL	Statement DeLetion
MST	Mutation Sensitivity Testing
DNA	Deoxyribonucleic Acid
ACGT	Adenine Cytosine Guanine Thymine
CFG	Control Flow Graph
NEHD	Normalized Extended Hamming Distance
GADGET	Genetic Algorithm Data Generation Tool
ABS	Absolute Value Insertion
LCR	Logical Connector Replacement
COBOL	Common Business Oriented Language
FORTRAN	Formular Translator
CPU	Central Processing System
PUT	Program Under Test

ABSTRACT

Full Name	:	Hayatullahi Bolaji Adeyemo
Thesis Title	:	Automated Mutation-based Test Data Generation: Genetic Algorithm Game-Like Approach
Major Field	:	[Software Engineering]
Date of Degree	:	[May 2018]

Testing is a crucial phase of software development life cycle. It is meant to improve the confidence in the quality of the software. One of the essences of testing is to uncover faults using test cases. Test cases that satisfy a given criterion are created to uncover faults. Various criteria have been proposed in the literature to ascertain adequate coverage of the different software behavior. Mutation coverage criterion is one of such criteria where analysis is performed to find tests that distinguish a program from its mutants. The criterion has only one requirement; that is to kill a mutant.

After three decades of research, mutation testing is still yet to be fully adopted by industries due to its high cost. The cost is due to the high number of mutants to be considered as well as the equivalent mutants generated unknowingly. Many researches have focused on solving one or more problems associated with the hesitation of adopting mutation testing by industries. Apart from developing effective tests, we also ensure non-trivial mutants are generated to excellently produce quality test cases.

The major contribution of this thesis is the development of a mutation-based novel game-like testing technique using Genetic Algorithms to allow development of meaningful program mutants on one side and generate tests cases that kill such mutants on the other side. In this research, we developed an approach to generating both mutants and test cases by two competing players. The approach was modelled as a non-cooperative game between a mutant generation player and a test generation player where both players use Genetic Algorithms in playing the game. The two players – mutantGA (mGA) and testGA (tGA) respectively generate hard-to-kill mutants and effective test cases to kill those mutants. The technique is validated experimentally by considering five case-study MATLAB programs. Results show that the technique is promising in, on one hand, generating strong and hard-to-kill mutants; and on the other hand, generating effective test data generated to kill most of those mutants. We also compared the performance of the GA-based players to the performance of random players; the GA-based players' performance was shown to outperform that of the random players.

ملخص الرسالة

الاسم الكامل: حياة اللهي بولاجي أدييمو

عنوان الرسالة: الانشاء الآلي لبيانات الاختبار باستخدام التبديل : طريقة الألعاب باستخدام الخوارزمية الجينية

التخصص: هندسة البرمجيات

تاريخ الدرجة العلمية: مايو ٢٠١٨

الاختبار هو مرحلة حاسمة من دورة حياة تطوير البرمجيات. يهدف إلى تحسين الثقة في جودة البرنامج. واحد من أساسيات الاختبار هو كشف الأخطاء باستخدام حالات الاختبار. يتم إنشاء حالات الاختبار التي تفي بمعيار معين للكشف عن الأعطال. تم اقتراح معايير مختلفة في الأدبيات للتأكد من التغطية الكافية لمختلف سلوكيات البرامج. معيار التغطية للطفرات هو أحد هذه المعايير حيث يتم إجراء التحليل للعثور على الاختبارات التي تميز البرنامج عن المسوخ. المعيار له شرط واحد فقط ؛ هذا هو قتل متحولة.

بعد ثلاثة عقود من البحث ، لا يزال اختبار الطفرات لا يزال يعتمد بالكامل من قبل الصناعات بسبب تكلفته المرتفعة. ترجع التكلفة إلى العدد الكبير من المسوخ الذي يجب أن يُنظر إليه بالإضافة إلى المتحولات المكافئة المولدة بدون علم. وقد ركزت العديد من الأبحاث على حل مشكلة واحدة أو أكثر من المشاكل المرتبطة بتردد اعتماد اختبار الطفرات من قبل الصناعات. وبصرف النظر عن تطوير اختبارات فعالة ، فإننا نضمن أيضًا توليد الطفرات غير الطفيفة لإنتاج حالات اختبار الجودة بشكل ممتاز.

وتتمثل المساهمة الرئيسية لهذه الأطروحة في تطوير تقنية اختبار تشبه الألعاب المبنية على الطفرات باستخدام الخوارزميات الجينية للسماح بتطوير طفرات برنامجية ذات مغزى على جانب واحد وتوليد حالات اختبارات تقتل مثل هذه المسوخات على الجانب الآخر. في هذا البحث ، قمنا بتطوير نهج لتوليد كل من المسوخ وحالات الاختبار من قبل لاعبين متنافسين. تم تصميم هذا النموذج على أنه لعبة غير متعاونة بين لاعب جيل متحور ومولد توليد اختبار حيث يستخدم كلا اللاعبين الخوارزميات الجينية في لعب اللعبة. يقوم اللاعبان - mutantGA (mGA) و testGA (tGA) على التوالي بتوليد مسوخ يصعب قتله وحالات اختبار فعالة لقتل تلك المسوخات. يتم التحقق من صحة هذه التقنية تجريبيا من خلال النظر في خمس برامج MATLAB دراسة حالة. تظهر النتائج أن التقنية واعدة في ، من ناحية ، توليد المسوخ قوية ويصعب قتل. ومن ناحية أخرى ، توليد بيانات اختبار فعالة ولدت لقتل معظم هذه المسوخ. نحن أيضا مقارنة أداء اللاعبين المستندة إلى GA لأداء لاعبين عشوائية. تم عرض أداء اللاعبين المعتمدين على GA بتفوق أداء اللاعبين العشوائيين.

CHAPTER 1

INTRODUCTION

Achieving user satisfaction is a major concern in software development. If the software cannot satisfy its intended users, the aim of developing it, in the first place, is defeated. Therefore, a high quality software is the one that does what the customers want it to do. Software quality, in this regard, is the conformance to explicitly stated requirements and standards. Software testing is an instrument to ascertain the software quality. Testing is the process of evaluating a component of a system or the whole system by manually or automatically verifying whether the system satisfies the specified requirements. The process is meant to uncover discrepancies between actual results and expected ones. Testing can be classified according to its level of granularity (e.g., unit, module, integration, and system), its characteristic (e.g., white-box and black-box), and its objective (e.g., reliability, robustness, security, performance, and user-friendliness) [1].

Functional software testing is a technical term used to refer to the process of validating software system in order to guarantee technical and requirement needs. Software testing is believed to be an expensive and time consuming task as it consumes roughly 50% of the development assets [1]. Taking into consideration the cost of carrying out testing, it is desirable to give it adequate attention so that the cost is reduced as well as the effort to be expended. One of the ways to reduce cost is to identify bugs in the early stage because any bugs identified later can cost more to fix as it may affect other earlier stages (e.g., design, implementation, etc.) of the development. As software complexity keeps increasing, there is an urgent need to generate effective test data to carry out the testing process in a cost-effective manner [2]. Software testing therefore helps in providing stakeholders with valid empirical reports about the quality of a software system, product, or service.

Mutation testing involves imitating competent programmer's mistake by injecting an error into a program to produce a mutant and investigate if the test cases can detect the injected error by observing if the outputs of the original program and the outputs of mutant are the same. If they are different, the error is detected otherwise it is not detected. Mutation testing is recognized as an effective type of testing software system [3]. However, it is not adopted in industry. The failure to adopt it is nothing but because of its cost [4]. This cost is incurred from creating mutants and executing them. It is prohibitively expensive to decide to execute all the possible mutants of a program even for an averagely big-sized program. The efforts expended in identifying equivalent mutants also contributes to the high cost of mutation testing. It can be concluded that mutation testing has two major problems: the problem of detecting equivalent mutants and the problem of the large number of mutants to be produced and executed. Mutation score of a test suite is the ratio of the mutants killed by the test suite to the total number of non-equivalent mutants involved in the execution. A mutation score of 1.00 signifies that all the mutants are killed and the test cases are mutation-adequate [4].

1.1 Problem Statement

It is generally established that software testing is one of the most integral parts of software development and costs up to half of the total budget [1]. High mutation score means the test cases are effective. But the value can at times be misleading if the mutants are trivial. A trivial mutant can be killed by any test case. In order to make mutation score more reliable, there is need to ensure non-trivial mutants are involved and this will give confidence in the test cases that will kill the mutants. We intend to target the problem of developing non-trivial mutants and developing effective test cases to kill them. This will help in demonstrating and mimicking the mutant creation by a competent programmer so as to have confidence on the test cases that can kill those non-trivial mutants. Also, measuring the effectiveness of whole test suite can be improved if the contribution of each test case in obtaining the score can be known. Also, as far as our knowledge of the researches in mutation-based test generation is concerned, no work has focused on generating both test cases and mutants automatically. Most of the researches show automatic generation of one of test cases and mutants and manual generation of the other.

This research would automate the generation of high quality mutants and effective test cases using GAs to address the aforementioned problems above.

1.2 Major Contributions

The major contributions of this thesis can be enumerated as follows:

1. Developing a framework and features to compare the existing GA-based test data generation techniques;
2. Proposing a GA-based test data generation technique;
3. Incorporating and implementing the test generation technique with mutants' generator establishing a non-cooperative game between them so that effective test cases and non-trivial mutants are generated competitively;
4. Validating the effectiveness of the implemented approach using 5 different MATLAB subject programs.

1.3 Organization of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 introduces the concept and detailed background of software testing, genetic algorithms and test data generation mentioning and explaining the different categories and types of software testing. In Chapter 3, related literature on genetic algorithms and mutation-based test data generation were identified. Also, research questions and research hypotheses are discussed. Chapter 4 introduces the proposed approach used in this research work. Chapter 5 presents and discusses the experiment and the results obtained. The research questions were also answered and the hypotheses were tested. Chapter 6 concludes the report of the thesis. It also presents some limitations to the study, threats to validity and future work.

CHAPTER 2

BACKGROUND

This chapter introduces software testing explaining different testing techniques used in verifying and validating software artifact against some quality attributes. It explains the testing theory and its importance in developing quality products. It also gives background on mutation testing as an important testing technique explaining its pros and cons. It also describes the rubrics of Genetic Algorithm (GA) and presents background knowledge on test data generation.

2.1 Software Testing

Software testing is an important phase in general software development. It comprises of test input generation, test execution and test output inspection. It involves running a program with the aim of uncovering errors in its source code. An estimate shows that more than 50% of the software development effort goes to testing [1]. The use of automated testing techniques would assist in curbing this cost significantly.

A programmer, as a fallible being, can make slight/small programming mistakes that can have negative impact on the productivity and scientific insight of the code. The consequence is more serious in safety software products where the smallest mistake can have an enormous effect on the products.

Testing involves selecting a finite subset of inputs that can help in measuring quality of the product. Testing can identify discrepancies between actual results and expected behavior or demonstrate functions are working or not according to the documented specification or provide a hint of correctness, safety, performance, reliability, security, fault tolerance, usability, etc. [5].

Testing, in the context of software, is "*the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component*" [6]. It is also "*the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items*" [7]. It is "*an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component* [7]." It can also comprise of any verification process to assess and improve software quality.

For deterministic software system, software testing involves defining known output for every input. The actual result is compared with the expected one after entering values, making some selections and navigating the application. We make a nod if they match otherwise we probably have a bug.

Software testing involves an essential combination of software verification and validation as frequently used by practitioners. Software verification is to find any present discrepancies between what a program is intended to do and what it does. It is the "*process of evaluating a system or component during or at the end of the development phase to determine whether it satisfies specified requirements*" [7]. In this case, the product at the end of the phase can be intermediate product, such as requirement specification, design specification, code, user manual, or even the final product. On the other hand, software validation is the act of checking the program behavior and its specifications with respect to the expectation of the users.

The importance of software testing cannot be overemphasized. Software testing helps in ensuring the entire specified functionalities are put into implementation while demonstrating that there are no faults in the implementation. Errors and mistakes in software is real and ignoring them till after deployment is foolhardy. Error detection and removal are achieved through software testing. Also, the level of reliability of the software under test is determined thereby ensuring confidence in the software.

Broadly speaking, software quality can be investigated using techniques that are categorized into two main groups: ***static analysis*** and ***dynamic testing***. Static analysis comprises of team of reviewers who read the code line by line correlating it with the logic of the specification. It is composed of inspection, walkthroughs and reviews. On the other hand, dynamic testing is a testing approach whereby the program code under test is executed with inputs and its behavior is observed. Due to human unwillingness to discover errors in their own work, testing is commonly performed by a separate group of people who are not part of the development team.

The following are some objectives of testing software [8],[1]:

- ✓ Ensuring the software under development is delivered error-free
- ✓ Ensuring the conformity of the software development to the requirements.
- ✓ Uncovering errors (if any/found)
- ✓ Attempting to have confidence that the end-product carries out the entire functionalities proposed.

A software can fail if there are wrong or missing requirements. Faulty design, faulty code and improper implementation of design can also cause software failure. Generally, software testing helps in identifying faults, correcting/removing faults and preventing future faults.

2.2 Mutation Testing

In the 1970s, mutation analysis was first introduced as a technique used to evaluate the effectiveness of a test suite [9]. A test suite is said to be effective if it is powerful enough to detect faults injected (intentionally or accidentally) into a software artifact, although the intent of mutation testing is to intentionally seed artificial faults which represent the real errors usually created by typical programmers. The software artifact could be a program code, specification, use cases and so on. This helps in giving some insight on how to improve the effectiveness of the test suite if there is need to do so. The same set of test cases in the test suite are executed on the mutated version called mutant. A mutant is a version of the original program with a simple syntactic change. This syntactic change is applied through mutation operators. The different changes made to the original program are known as mutation operators[1] while a mutant is obtained when a mutation operator is applied to a code [1]. Mutation Score = Number of killed mutants/Total non-equivalent mutants * 100

[1] The term has been used differently. It is also known as mutant operators, mutagens, mutagenic operators, mutation rules, mutation transformations.

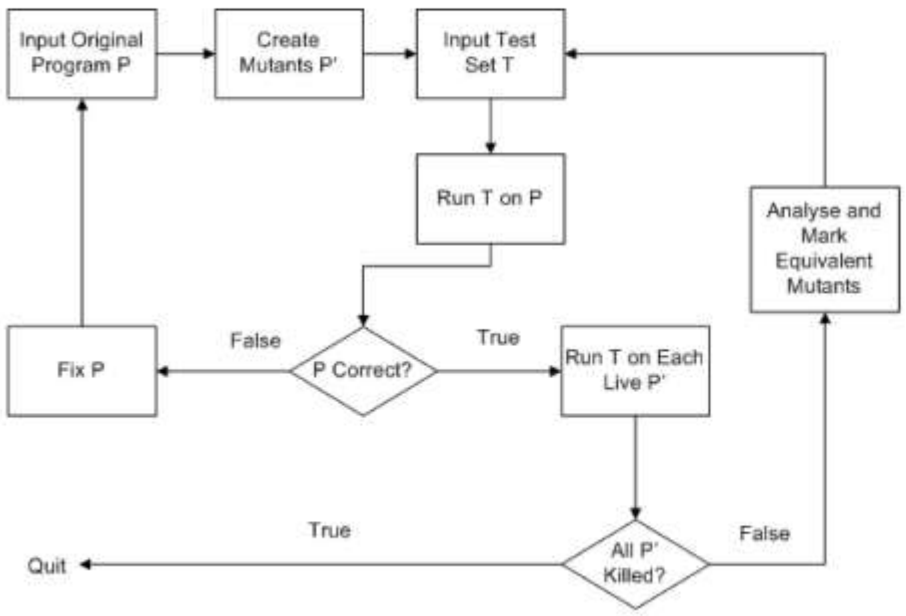


Figure 1: Mutation Testing Process Flow chart (from [10])

Testing of a program using mutation is considered as secondary level testing because mutation testing cannot be conducted unless unit testing is successfully carried out. The main inspiration behind invention of mutation testing is not too complex. A number of simple errors are introduced to the original program based on the mutation operator selected, generating test cases to differentiate these mutants gives a tendency of detecting the real faults. This is similar to coupling effect that states that a test data set that catches all simple faults in a program is so sensitive that it will also catch more complex faults. It is a powerful testing technique, however it has very low applicability in industries. A number of drawbacks have restricted its practical impact. A high number of mutants generated from the standard set of mutation operators makes it too expensive to implement in practice. This problem is minimized by selecting few appropriate mutation operators.

Steps in Performing Mutation Testing

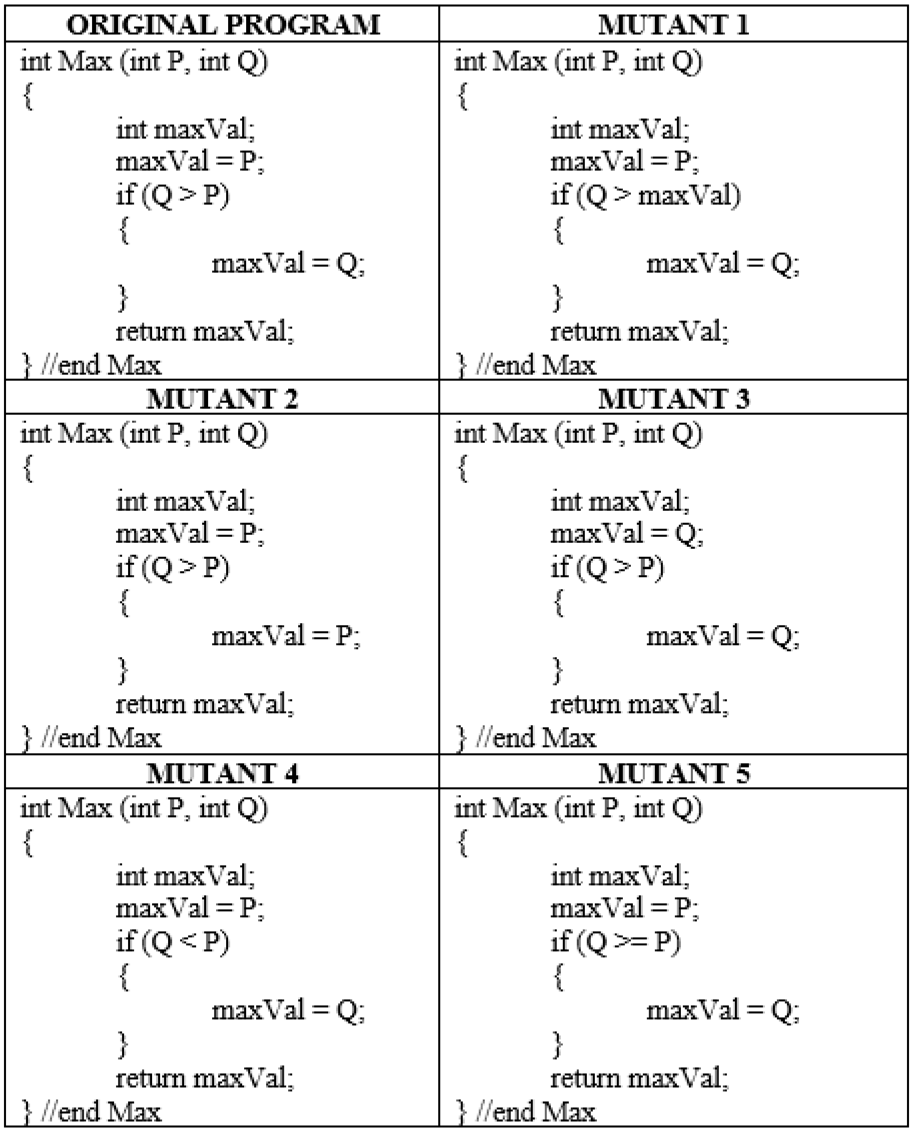
Given an original program P_0 and a set of test cases T, traditional mutation testing can be elucidated as a series of steps to evaluate set of tests as follows:

1. Apply every member of a set of mutation operators to P_0 to produce a set of mutants P_M.
2. Execute the test set T on P_0 and each mutant p_m in P_M ($p_m \in P_M$).
3. Carry out the comparison between output of $p_m(t)$ and $P_0(t)$ for all t in T. If the outputs are equal, then mutant p_m is killed; otherwise p_m is alive: no test output has been affected by p_m's mutation.
4. Analyze the live mutants to determine the equivalence of any of them; equivalent mutants are discarded as they are syntactically identical to the original program.
5. An attempt to kill the nonequivalent mutants by adding new test(s) to the test set.

Repeat steps 2-5 until results are satisfactory.

A test suite is passed to both original program and its mutant, if the output differs the mutant is said to be killed otherwise it is alive. The test suite is therefore incrementally augmented with more effective test cases to further detect the unexposed mutants until the alive mutants are killed or considered to be semantically equivalent to the original program. Some mutants cannot be killed by any test case – these are called equivalent mutants. A mutant of a program is said to be an equivalent mutant if it is functionally and/or semantically the same as the original program, else it is called a non-equivalent mutant. One of the main properties of an equivalent mutant is that it cannot be killed at all. But for non-equivalent mutants, some can be killed while some may not be killed depending on the effectiveness of the test cases. If the test cases are effective enough, the non-equivalent mutant(s) would be killed otherwise there will be need for additional test cases or effective ones.

For very small program, there may be a numerous number of mutants that can be generated. Example of such mutants is shown in Figure 2. The figure shows how even small-sized programs can generate many mutants.



ORIGINAL PROGRAM

```
int Max (int P, int Q)
{
        int maxVal;
        maxVal = P;
        if (Q > P)
        {
                maxVal = Q;
        }
        return maxVal;
} //end Max
```

MUTANT 1

```
int Max (int P, int Q)
{
        int maxVal;
        maxVal = P;
        if (Q > maxVal)
        {
                maxVal = Q;
        }
        return maxVal;
} //end Max
```

MUTANT 2

```
int Max (int P, int Q)
{
        int maxVal;
        maxVal = P;
        if (Q > P)
        {
                maxVal = P;
        }
        return maxVal;
} //end Max
```

MUTANT 3

```
int Max (int P, int Q)
{
        int maxVal;
        maxVal = Q;
        if (Q > P)
        {
                maxVal = Q;
        }
        return maxVal;
} //end Max
```

MUTANT 4

```
int Max (int P, int Q)
{
        int maxVal;
        maxVal = P;
        if (Q < P)
        {
                maxVal = Q;
        }
        return maxVal;
} //end Max
```

MUTANT 5

```
int Max (int P, int Q)
{
        int maxVal;
        maxVal = P;
        if (Q >= P)
        {
                maxVal = Q;
        }
        return maxVal;
} //end Max
```

Figure 2: Examples of mutants

Five different operators are applied to the original program to produce the five mutants shown in Figure 2. The variable P in line 5 (***Δ if(Q>maxVal)***) is replaced by maxVal to produce the first mutant. A variable P replaced the variable Q in line 7 (***Δ maxVal=P***) to generate the second mutant. The third mutant is also obtained by changing the variable P in line 4 to Q (***Δ maxVal=Q***). The relation operator in line 5 (if(Q>P)) is substituted by < and >= to produce mutant 4 and mutant 5 respectively.

For very large program, the number of mutants generated can be too much to handle and as such the cost of carrying out mutation testing would be prohibitively expensive.

Traditional Mutation Testing has been applied by software engineers/testers for more than 4 decades not only to detect faults in software artifacts but also to evaluate their tests.

Mutation Testing guarantees a promising and effective approach to generate adequate test data out of which real faults are found. It is almost impossible to generate all possible mutants because the number of such potential faults for any given program is prohibitively huge. This is the reason why the traditional mutation testing focuses on those faults that are close to the correct version, which are only a subset of the faults with the likelihood that they will be enough to simulate the whole faults. This principle is explained by two hypotheses: The Competent Programmer Hypothesis and the Coupling Effect.

The Coupling Effect and Competent Programmer Hypothesis were postulated by DeMillo et al. [11] in 1978. While Competent Programmer Hypothesis affects the programmer's behavior, the Coupling Effect involves the type of faults applied in mutation analysis. Coupling Effect states that "test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors” [8].

Andrew et al. [12] showed that real faults are easier to detect than hand-seeded faults but the authors argued that no matter how much research is conducted on testing using mutation, some questions still remain unanswered, such as "*Do mutation operators provide sufficient coverage of all possible fault types?*" and "*Are mutation operators a better means of creating faulty code than hand-seeding*?".

There are three necessary criteria to ensure that a mutant is killed. They are Reachability, Infection and Propagation. They are represented in a model known as RIP model. Each of the conditions subsumes its predecessor. It should be noted that mutated statement must be executed in order to detect a mutant.

a. Reachability (R): This is the first condition required for mutation to take place. The program must be executed by a test case ensuring that the statement that is mutated is “reachable”. The statement should not contain dead code – which is unreachable.
b. Infection (I): The faulty statement results in an incorrect state by the test. The state of the mutated program must be different from that of the original program after the execution of the test case on the mutant, i.e. the state of the mutant must be infected. This condition is achieved by both weak and strong killing of mutants. The last condition would distinguish strong killing from weak killing.
c. Propagation (P): This causes the incorrect state to propagate into incorrect output(s). Any test case that achieves this condition (propagation) is said to ‘strongly kill’ the mutants. In this case, weak killing of mutants does not achieve propagation. This means weakly killing satisfies only reachability and infection, but not propagation.

The incorrect state has been corrected or does not have effect on the final output of the program.

Strong Mutation Testing: Strong mutation testing is believed to be the traditional mutation testing. And the idea is to make a number of small changes, one at a time for non-higher order mutation, to a particular program. Then an attempt is made to generate test data that would expose the mutation by distinguishing it from the original program. Any mutation that satisfies the three conditions (discussed above i.e. Reachability, Infection, and Propagation) is referred to as a strong mutation.

Weak Mutation Testing: A weak mutation testing is the one that satisfies only *reachability* and *infection* but not *propagation* unlike the strong mutation testing. One of the main disadvantages of strong mutation testing is its cost of computation, which is caused as a result of the large number of possible mutants and also the requirement of executing each test case to completion. The introduction of weak mutation was to defeat the implicit cost of strong mutation testing. Different test execution is not necessarily required for each mutation in weak mutation. However, the main disadvantage of weak mutation is that several different components of a particular program can generate different results from the original program following different executions but can combine to assign the overall accurate outcome to the statement concerned or indeed to the entire program execution [13].

Mutation testing requires the code structure knowledge. Possible faults that could occur in a software component is considered in order to generate test data and carry out effective evaluation of testing.

However, among the advantages of mutation testing are the following: (1) it guides to produce reliable software product, (2) it helps in uncovering ambiguities in program code, and (3) it makes the testing process to be more comprehensive.

2.2.1 Mutation Operators

Mutation operators can be classified into major groups: replacement, deletion, insertion, etc.

A subset from the set of mutation operators can be selected from the following list:

- Arithmetic Operator Replacement,
- Comparable Array Replacement,
- Comparable Constant Replacement,
- Comparable Variable Replacement,
- Logical Operator Replacement,
- Relational Operator Replacement,
- Unary Operator removal/insertion.

Deletion Mutation Operators:

The deletion group of mutation operators comprises and not limited to the following:

- Statement deletion operator
- Operator deletion operator
- Variable deletion operator
- Constant deletion operator

According to researches [14]–[17], statement deletion mutation operator has been applied to improve the cost-effectiveness of mutation testing. Although, the statement deletion mutation generates relatively few mutants not more than the number of statements in a code under test, effective tests are yielded because only few equivalent mutants are generated as a result. The concept of applying a single but powerful mutation operator has led to generation of effective test set with a low cost in a process known as One-Operator or Single-Operator mutation. Statement Deletion (SDL) is an example of operators that employ such one-operator mutation.

The reason why statement deletion mutation is said to generate relatively few equivalent mutant is SDL mutants can only be equivalent to the original program if the statement deleted is, in the first place, unnecessary.

Although there is a connection between mutation operators across different programming languages, they must be selected specifically for each language because the language feature affects the operators. Naturally, passing all the possible error a programmer can commit to create mutated programs would be sufficient to ensure the effectiveness of the test cases. But, however, ascertaining that it is feasible to construct all possible potential errors is unrealistic with a few exceptions. In lieu of this, a subset of the entire possible mutants is selected. This has caught a number of researchers' attention to a concept known as "Selective mutation operators" – which reduces the number of potentially generated mutants through decreasing the number of mutant operators. For more information on selective mutation, the reader can refer to [18]–[25].

Table 1: Java Class-level mutation operators

MUTATION OPERATORS	**DESCRIPTIONS**
AMC	Access modifier change
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overridden method calling position change
IOR	Overridden method rename
ISI	super keyword insertion
ISD	super keyword deletion
IPC	Explicit call of a parent's constructor deletion
JTI	this keyword insertion
JTD	this keyword deletion
JSI	static modifier insertion
JSD	static modifier deletion
JID	Member variable initialization deletion
JDC	Java-supported default constructor deletion

EOA	Reference assignment and content assignment replacement
EOC	Reference comparison and content comparison replacement
EAM	Accessor method change
EMM	Modifier method change
PNC	new Method call with child class type
PMD	Member variable declaration with parent class type
PPD	Parameter variable declaration with child class type
PCI	Type case operator insertion
PCC	Cast type change
PCD	Type cast operator deletion
PRV	Reference assignment with other comparable variable
OMR	Overloading method contents replace
OMD	Overloading method deletion
OAC	Arguments of overloading method call change

The table above shows the class-level mutation operators for Java.

2.2.2 Problems of Mutation Testing

Despite the growing interest received by mutation testing in academia, it is hardly applied in industries because of two main reasons among others: cost of generating mutants (and/or test cases to kill those mutants) and ability to identify equivalent mutants. As a result of this, it is almost impossible to achieve a mutation score of 100%. Mutation score is the percentage of mutants killed.

Equivalent mutants result because some programs are only syntactically different but semantically the same and/or some fragments of the code may not be executed because they are not reachable, which is a concept referred to as dead code.

However, the difficulty experienced in identifying and killing equivalent mutants remains one of the limitations and disadvantages of mutation testing. Also, it is time-consuming unless it is automated.

2.3 Genetic Algorithms

This section presents the biological background of Genetic Algorithms (GA). It also discusses some details of GA.

2.3.1 Biological Background of Genetic Algorithms

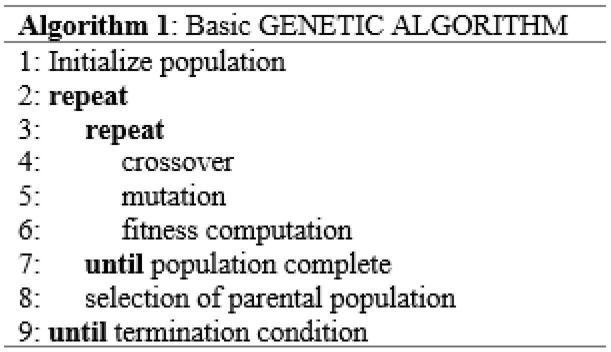
Genetic Algorithm is a computational counterpart of biological genetics. Genetics is the study of genes. It deals with the description of genes, what they perform and how they perform their work. It studies how features or traits are transferred from parents to children. The study of genetics helps in answering questions like "*Why do offspring look like their parents?*" and "*How can different diseases transmit in families?*" An informal study of genetics has been in existence since time immemorial but its study as a study as a set of analytical procedures and principles did not start until 1860s, when Gregor Mendel, an Augustinian monk, conducted a set of investigations that indicated the existence of biological materials now known as genes [26]. A living organism is composed of cells. A cell can be described as a unit of life i.e. unit of living organisms. The cell is the fundamental functional and biological unit of all living organisms. It is usually referred to as the “building block of life”. The field of biology dedicated to the study of cell is known as cell biology. A cell could be a plant cell or an animal cell, they have many common features and few differences. Each cell has its lifespan and can easily be replaced. One of the prominent and common features of cells of advanced organisms (Eukaryotes) is the nucleus. There is usually only one nucleus in a cell. The nucleus operates to process cell information. It performs this by storing the cell’s hereditary material (DNA) and coordinates the cell’s activities, such as growth, protein synthesis, and reproduction. Cell reproduction is otherwise called cell division. DNA is composed of four nucleotides, each comprising of deoxyribose sugar, phosphate, and one of these four nitrogen bases: Adenine (A), Thymine (T), Guanine (G), and Cytosine (C). They are encoded as ACGT. DNA has two nucleotide chains arranged in an antiparallel direction to each other and held firmly together by pairing A with T and G with C. The nitrogen bases are grouped into two namely: purines and pyrimidines. Adenine and guanine are purines while cytosine and thymine are pyrimidines. The DNA encodes information needed by a cell to express certain genes. Genes are the determinants of the inherent properties of species of organisms. Biologically, the gene of an organism is decided by both or one of the parents depending whether the organism is replicated through asexual or sexual reproduction respectively. For example, a bacterium is obtained from one parent cell dividing into two cells and comprised of the same genes as its parent cell. On the other hand, human being has a pair copy of each gene – a set from the father and the other one from the mother. Therefore, individual’s physical feature like skin color is usually defined by the mixture of multiple genes. Although the individual's environment is also an important factor that impacts the expression of genes.

Biologically, every living organism is made up of different cells. Each cell has a set of chromosome, which are DNA (DeoxyriboNucleic Acid) strings which is the main composition of an organism. A chromosome is a specialized structure made from many tightly packed strands of DNA and proteins known as histones. Different strands of DNA are wrapped around the histone proteins to form a long worm-shaped configuration known as "chromatids". Two of the chromatids join together to form a chromosome. Chromosomes are created in the nucleus of a cell when the cell is dividing in a process called ***cell division***. The number of chromosomes varies among different species. Some species have more chromosomes than 100 while others have as few as two but humans have 46 chromosomes [27].

GA was invented by John Holland and can be used to schedule tasks, design computer algorithms and to solve optimization problems. The genetic algorithm is exterminated by two factors: when the optimal value/solution is obtained or when the number of generation is exhausted [27].

2.3.2 Details of Genetic Algorithms

Genetic Algorithms are optimization techniques used to solve non-linear or non-differentiable optimization problems. They are named as such because they are instigated by the principles of natural selection and genetics. They are regarded as optimization algorithms because they are applied to determine the optimal solution by obtaining the minimum and maximum of a function. They apply concepts from evolutionary biology to search for a global minimum to an optimization problem. The principle of the "survival of the fittest" proposed by Charles Darwin was followed to implement them. The GA was invented by John Holland at the University of Michigan in the 1970s. It repeats fitness evaluation, selection and crossover, and population reassembly. A sufficient number of children are created from few parents. Each time, two parents are copied, crossed over and mutated. This results into two children every time two parents are copied. One of the reasons why it is becoming more popular than the conventional AI is due to its robustness. Also, minor change in the input does not easily break GA. It also proposes substantial advantage over typical search optimization techniques (such as breath-first, depth-first, heuristic, and linear programming) especially in searching a very large space, n-dimensional surface, and multi-modal search space. The name was adopted due to the fact that they are mimicking the evolutionary biology techniques. They are implemented as a computer simulation in which a population of abstract representations of candidate solutions to an optimization problem evolves toward better solution. The solutions are traditionally represented in binary as strings of 0s and 1s, but other encodings are also possible. GA works by initial generating of candidate solutions that are tested against the objective (fitness) function. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are selected from the current population (based on their fitness), and modified (recombined and possibly mutated) to form a new population. Subsequent generations are obtained from the first one through some genetic operators: selection, crossover and mutation. Usually, the algorithm terminates when either a satisfactory fitness level has been reached for the population or a maximum number of generations has been produced. Although, if it terminates due to a maximum number of generations reached, a satisfactory solution may or may not have been obtained. A typical GA needs to be defined by two things namely: genetic representation of the solution domain and a fitness function to evaluate the solution domain. The basic Genetic algorithm is shown Figure 3.



```
Algorithm 1: Basic GENETIC ALGORITHM
1: Initialize population
2: repeat
3:     repeat
4:         crossover
5:         mutation
6:         fitness computation
7:     until population complete
8:     selection of parental population
9: until termination condition
```

Figure 3: Basic Genetic Algorithm

Genetic/Chromosome Representation:

The performance of any GA-based function optimizer depends on the representation of the chromosomes. Different problems have different methods of representing their chromosomes in GA such as binary, gray, integer or floating data types. The bit (binary digit) format is the most common type. In this case, the variable values are the combination of zeros and ones {0,1}.

Although, arrays of other types and structures are used essentially the same way, but array of bits is the standard representation of the solution. These genetic representations are easy due to GA's nature and convenient to implement because they have fixed size and can easily be aligned to facilitate simple crossover operation. A certain level of complexity is involved in variable length representations. Usually, the composition of the binary digits makes up a chromosome which is a potential solution to a problem which can in turn consist of a set of variables. For instance, if the problem has only three input variables P, Q, R, then the representation of the chromosome can be the concatenation of the binary equivalence of each of P, Q, and R as shown in Table 2.

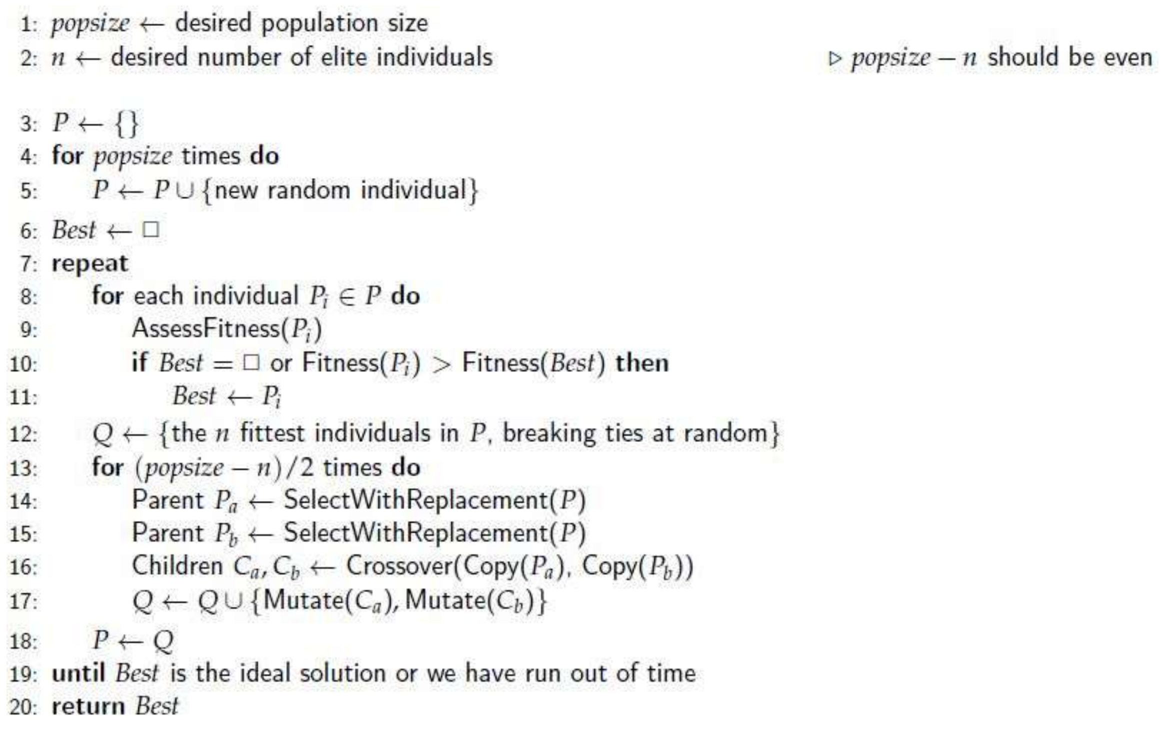
Table 2: Chromosome representation and interpretation

CHROMOSOME			
P	**Q**	**R**	
0 0 0 0	0 0 0 1	1 0 0 1	P = 0, Q = 1, R = 9
0 1 0 1	1 1 0 0	0 0 1 0	P = 5, Q = 12, R = 2
0 1 0 0	1 1 1 1	1 0 1 1	P = 4, Q = 15, R = 11

Each of P, Q, and R can be used to denote the size of a triangle or the coefficient of a quadratic equation. It should be noted that it is not necessary that the binary encoding of each of P, Q, and R be of the same length but has to be consistent across different chromosomes.

Elitism is a slightly modified version of the traditional GA. It injects the fittest individual or individuals into the next population from the previous population. The fittest individuals are otherwise known as ***elites***. The highly-fit parents compete with their children and results in an exploitative behavior. Since the elites are added into the next population, crossover needs to be carried out by subtracting the number of elites and divide by two in order to maintain the population size. The default value of elite count – which is the number of individuals that are guaranteed to survive to the next generation because of their good fitness value – is usually 2. High value of elite count drives the GA towards more exploitation, which as a result can make the search less effective.

The algorithm in pseudocode is shown in Figure 4.



```
$popsize \leftarrow$ desired population size
$n \leftarrow$ desired number of elite individuals          $\triangleright$ $popsize - n$ should be even

$P \leftarrow \{\}$
for $popsize$ times do
    $P \leftarrow P \cup \{$new random individual$\}$
$Best \leftarrow \square$
repeat
    for each individual $P_i \in P$ do
        AssessFitness($P_i$)
        if $Best = \square$ or Fitness($P_i$) $>$ Fitness($Best$) then
            $Best \leftarrow P_i$
    $Q \leftarrow \{$the $n$ fittest individuals in $P$, breaking ties at random$\}$
    for $(popsize - n)/2$ times do
        Parent $P_a \leftarrow$ SelectWithReplacement($P$)
        Parent $P_b \leftarrow$ SelectWithReplacement($P$)
        Children $C_a, C_b \leftarrow$ Crossover(Copy($P_a$), Copy($P_b$))
        $Q \leftarrow Q \cup \{$Mutate($C_a$), Mutate($C_b$)$\}$
    $P \leftarrow Q$
until $Best$ is the ideal solution or we have run out of time
return $Best$
```

Figure 4: The Genetic Algorithm with Elitism (from [28])

Fitness Function:

A fitness function "is a type of objective functions which summarizes the goodness of a solution with a single figure of merit" [29]. This is used to compute how good the solution represented by a chromosome is in relation to the global optimum [if known]. Each chromosome in each population has its fitness computed by the fitness function. This creates a factor to compare the different individuals and to rank them. The individual with the highest fitness denotes the nearest to the optimum solution. The GA can get feedback from the problem through the fitness value.

For instance, if we are to optimize a function f(X) = $2x^2$ given $x \in$ [0,1,3,5], then the fitness function would be represented as follows:

$$fitness = 2x^2$$

The following are the characteristics of a fitness function:

* Measurement: The fitness function must be quantitatively measurable as this will tell if candidate solution is fit and/or how fit it is.

* Fast: This is because fitness function accepts the candidate solution and assess it to know how fit/good it is. This is done repeatedly, hence the reason why it should be sufficiently fast so as not to delay the entire processes.

In GA, the initial chromosomes (population) are generated randomly.

There are three main GA operators namely:

Selection:

From the generation of chromosomes, selection operator chooses two individuals to be used for recombination. The selection can be randomly or based on some heuristics such as the fitness value. This means that if the selection is randomly, each of the individuals has equal chance of being selected while chromosomes with higher fitness value have higher chance of being selected if they are selected with regards to their fitness values.

Selection implies retaining the best performing chromosomes. There exist a number of different strategies to select the individuals to be copied over into the next generation. For a binary problem (i.e. problem with binary representation), selection means to preserve the bit strings that has better performance from a generation to the next generation. In other words, it determines among the population which individuals survive to the next generation. This is carried out in each iteration (generation) to create the new population from the old ones after evaluating them. Roulette-wheel, Elitist, Fitness-proportionate, scaling and rank selection are different methods of selection [28].

Crossover:

This is one of the binary operators that work with two operands. The operands are the two selected chromosomes (parents). It works by interchanging substrings to produce two offspring that are included in the next generation. In some cases, the offspring are included into the next generation without establishing whether they are eligible to be in the population. In other words, they represent invalid chromosomes. In this case, they are assigned poor fitness that makes them to be excluded in the subsequent generations. This can be illustrated in knapsack problem. A chromosome that represents total available objects in the knapsack to be greater than the capacity of the knapsack is considered an invalid chromosome. It is either not included in the first place or included and assigned a very poor fitness. It should be noted that in such case, crossover of two valid chromosomes can result into one or two invalid chromosomes. The objective of crossover is to create a better (fitter) individuals over time. It takes place according to a crossover probability *Pc*.

Mutation:

Before explaining what mutation operator does, let us consider the following population

0 0 1 1 0 0 1 1

0 1 1 0 0 1 1 0

0 1 0 1 1 0 1 0

0 1 1 0 1 1 0 1

0 0 1 0 1 0 1 1

There is no amount of reproduction/crossover that can change the first bit to 1. So if the optimum candidate must have its first bit to be 1, then the optimum would be missed definitely. Therefore, changing the first bit from 0 to 1 can help. Changing bits from 0 to 1 (or 1 to 0) with a probability of Pm is called mutation in the context of genetic algorithm. It should be noted that the value of Pm should be very small.

Mutation is a random process whereby a gene of a chromosome is replaced by a new one generating a new genetic structure. It is randomly applied with low probability. Its role can be depicted as a protection or safety to recover good genetic material that may be lost through implementation of selection and crossover operators.

2.4 Test Data Generation

As it is known that testing is a major task in software development, test case generation is most crucial to software testing. In fact, it is one of the most complicated tasks in software testing process. It is aimed at generating sets of test cases that can detect as many faults as possible in a software artifact. Ability to generate an effective and efficient test cases enhances the achievement of testing objectives. Test cases are not only obtained from source code but also in other design artifacts. Generating test cases from design documents enables the availability of the test cases prior to the testing phase and thereby speeds up the process and allows more effective planning of test cases. It is worth noted to know that any bugs or inconsistencies detected early saves the development time. This means if the test cases are generated earlier enough, the ambiguities in the specification and design can be get rid of and allowing them to be improved even before writing the program.

Although, test data is more pronounced for code-based testing, it is also applicable to specification-based and model-based testing. Here, more emphasis is given to code-based testing. Generating test data is not a trivial task as each product of software development phase generates a huge information. Therefore, in order to generate effective tests at the same time lowering the cost, test designers should analyze the input and output domain. Not all values in an input domain of a program have the same meaning and importance but some values have special meanings; i.e. some are more important than others. This can be illustrated, for instance, by studying the *factorial* function. The factorial of a nonnegative integer n is calculated as follows:

Given: factorial (0) = 1; and factorial (1) = 1;

Factorial(n) = n * factorial(n-1).

A programmer may carelessly and wrongly implement the factorial function as:

factorial (n) = 1 * 2 * 3 * … * n;

The above implementation may seem correct as it will produce correct results for all positive values of n but will fail if n = 0. As it can be seen that 1 is an output for two different factorials (i.e. 0 and 1).

To sum it up, not only input and output domain should be considered when designing test case but specification, source code should also be considered. Considering information from several sources will assist in providing complementary information required to design test cases.

Test data can be generated either by black box approach or white box approach. Below is the brief overview of the two methods.

2.4.1 Test Case Design for Black Box Testing

A number of industries have adopted the black box test design techniques as their best practice. This helps them in saving lots of testing time and obtaining good test coverage. One good feature of black box test design techniques is that the knowledge of the internal structure of the artifact under test is not necessary. Test cases are derived from the requirement specification document and based on the expertise of the testers using the following test design techniques [30]:

- Boundary Value Analysis: This is used to find errors in a program at boundaries of the input domain as opposed to using inputs in the center of the domain.
- Decision Table: It is used whenever a complicated logic is to be modeled. It is used to detect any missing combination of conditions in the logic.
- Equivalence Partitioning: This involves dividing the test conditions into groups. From each group, only one condition is tested with the assumption that each member of the same group behave similarly.
- Exploratory Testing: This involves continuous optimization of the quality of testing by simultaneously treating test design and test execution in parallel throughout the process.
- Error Guessing: Bugs are discovered in a software based on tester's previous experience. For example, entering invalid values like entering alphabets in the numeric field, and submitting a form without entering values.

- State Transition Testing: Design of tests to execute both valid and invalid state transition to investigate the behavior of the system under test.

2.4.2 Test Case Design for White Box Testing

Most systems such as mission critical systems and components adopt white box testing techniques because of the attention to detail these techniques can offer. It is a well-known fact that an exhaustive (complete) testing is impossible and that testing cannot guarantee the absence of faults. As such, there is need to select a subset of test cases from all possible test cases that has the highest likelihood to detect as much faults as possible. This leads to test case design strategies. Each strategy depends on the scenario and the domain knowledge. Test case design can be obtained from the requirements of the program under test (i.e. its specification), informal description, set of scenarios (use cases), set of sequence diagrams, and state machine. It can also be obtained from the program itself, set of selection criteria, heuristics and experience. Program code is tested and executed (i.e. covered) using one of the following kinds of coverage: statement, path, (multiple-) condition, decision/branch, loop and definition –use (def-use) coverage.

2.4.3 Test Data Generation Using Genetic Algorithms

Over the years, a number of researches have been conducted to generate test cases. The trend of research is now deflected towards generating automatic test data. This is an attempt to reduce the high cost of testing software manually and also to increase the reliability of the software artifact under test. This leads to the evolution of different approaches ranging from generating test cases from requirements, use cases, models or source codes applying different test objectives such as coverage criteria, with several different techniques and algorithms depending on the problem domain. Most of the researches are considered white-box approaches in which there is no need for any specifications, although the existence of specification can aid test case generation [31].

Over time, the process of generating test cases had been automatically carried out. This can be broadly divided into three different categories: random, static and dynamic.

Although, random test data generation process is not difficult to automate, it stands the chance of creating too much number of test data or may fail to find test data that is capable of satisfying a test requirement. This is as a result the necessary information concerning the test requirement not incorporated into the process of generation.

Static generation cannot be automated because it does not require the program execution. A typical example of static technique is the symbolic execution. It is done by navigating a Control Flow Graph (CFG) of a program and in terms of the input variables, which constructs the internal variables for the desired path. A number of constraints are established by the branches in the code. Solving these constraints results to the required test data. Dynamic generation of test data is different from static techniques in the sense that it requires the execution of the program which leads to a directed search for test.

Many researchers applied optimization techniques to automate and generate test data. This is facilitated not only by the fact that a substantial number of testing problems can be formulated as search problem, but also because they can be formulated as optimization problems.

Applying metaheuristic techniques like Genetic Algorithms to generate test data in software testing, the inputs are optimized based on certain criteria. In that regard, software testing is seen as an optimization problem. Before achieving the conversion of software testing into optimization problem, software metrics that are to be optimized should be defined or chosen. The metrics should have direct or indirect measurability from the software. In white box testing, possible metrics can be test coverage metric: code, condition, or path coverage. But in black-box testing, the metrics to be optimized could be error based; for example, amount of warnings, calculation or rounding errors, leakage of memory, etc., or temporal based e.g. best or worst execution times or response times (B/WCET) [32].

In black-box testing, it is not the tester's problem to detect what causes the unexpected output because the tester is to test the given software as well as possible and report as clearly as possible what has been obtained to the programmer. It is now the responsibility of software developers to search, inspect and fix the erroneous code lines.

CHAPTER 3

LITERATURE REVIEW

A number of researches have been conducted to address the test case generation problem. Test cases are being generated using different approaches. Some researches focus on applying mutation analysis, some use genetic algorithm to generate tests while few utilized the combination of both mutation analysis and genetic algorithm, among others. Below is the review of some of the related works that are considered important in respect of the test cases generation techniques using genetic algorithm, mutation testing and search-based mutation testing. In this chapter, a comparison framework is presented to identify the strengths and drawbacks of the several different test data generation techniques with/without mutation testing after presenting the summarized discussion on some of the existing works in the field. Most of the search based techniques were applied to mutation analysis in order to optimize either mutants or test cases or both. Mutants optimization can be reduction of the number of mutants, which can be a good representative of the entire mutants or identifying and eliminating equivalent mutants. Test cases can be optimized by reducing the number of test cases or increasing the effectiveness of the test cases as a whole. A list of M.Sc. and PhD theses is also documented to identify available work and detect some of the research gaps in the field of mutation-based evolutionary test data generation.

3.1 Genetic Algorithms Based Test Case Generation

A number of techniques have been used to generate test cases while carrying out mutation testing.

In recent years, researchers have been exploring researches in Genetic Algorithm (GA) theory and applications. It is used in solving many problems while efforts have been made to improve the performance of GAs especially in applications to solve optimization problems. In order to apply a GA to solve a particular problem, some factors are exceptionally crucial to be considered; such as identification of the object(s), problem representation, design of a GA and interpretation of the search results to the solution [33]. Each GA is designed taking into consideration the nature of the problem. This makes the GA to have different input values, input formats and even data structures. A quite number of researches have focused on applying GA to generate test cases for testing software artifacts. Some of them are as follows:

DeMillo and Offutt [34] proposed a fault-based technique that applies algebraic constraints to describe test case to uncover fault categories. They implemented their technique in a tool called Godzilla, which generates constraints and solve them automatically. The tool is used to carry out both unit and module testing. It is integrated with the Mothra testing system.

Lin and Yeh [35] developed test data creation technique for path testing using GAs. An iterative sequence of operators was executed to generate test cases to test paths coverage in a program using GA. A metric was formulated to determine which test case survives to the next generation and fitness function was designed based on the formulated metric. Hamming Distance [36] was modified to calculate the fitness function as Normalized Extended Hamming Distance (NEHD).

Doungsa-ard et al. [37] proposed a framework for generating test data from software specifications. The test data generated is a sequence of actions from the software specification and the UML state machine diagram. They measured the quality of the test data by the number of transitions which is fired by the input. GAs are used to optimize the sequence of triggers to find the best one to cover the most transitions.

Michael et al. [38] presented a technique on generation of test cases by function minimization using genetic search. Test data were generated using branch coverage criteria. They implemented the technique in a tool known as **G**enetic **A**lgorithm **D**ata **GE**neration **T**ool (GADGET).

Ghiduk et al. [39] proposed an automatic test data generation technique using definition-use path coverage satisfying data-flow coverage criteria using GA. They developed a new multi-objective fitness function to evaluate the generated test data using the concept of dominance relations between nodes. Control flow graph of the program was used to generate dominance tree. They stated that the reduction in the size of test suites and the total number of iterations to satisfy the data-flow criteria prove the effectiveness of their approach in relative to random testing. The inputs for the technique were the set of test requirements, a version of the program under test, dominance tree and the usual GA parameters.

Ahmed and Hermadi [40] proposed and presented a method to improve the efficiency of using GA to generate test data by designing an automatic GA-based test data generator for white box testing covering multiple target paths. The results obtained are promising as they show better performance than other existing approaches used in comparison.

Srivastava and Kim [41] presented a testing approach using GA to find the most critical paths in a software construct. This was achieved by creating variable length GA that does not only optimize but also select the software path clusters that are weighted according to the criticality of the path. This makes the most critical paths to be tested first, since an exhaustive test is rarely practical, which in turn can increase the efficiency of the testing process. In their technique, each edge of the control flow graph was assigned weights and the sum of the weights of the entire edges in a specific path forms the fitness function. The criticality of the path is proportional to the fitness values.

Domínguez-Jiménez et al. [42] designed their fitness function by penalizing groups of mutants which are killed by the same set of test cases, without regarding the location, the mutation operator applied or the number of mutants in the group. Harman [46] presented a keynote talk by summarizing the existing work, the analysis of performance of several search algorithm used in test data generation and techniques to minimize search spaces.

A number of outstanding and comprehensive surveys of the test data generation using search-based software testing approach have been presented [43],[44],[45].

3.2 Mutation-Based Test Case Generation

Killing mutants is a better way of testing the tests. A number of researches have been recorded in mutation testing. Some concentrate on defining new mutation operators, while others develop mutation system. Research in mutation testing can also be to invent innovative ways to reduce the cost of mutation testing [42]. This research is focused on developing cost-effective mutation system. As earlier stated, mutation testing suffers a number of shortcomings, which minimize its adoption in industry. Despite the little survey work in the literature on mutation testing, there has been a number of research work presenting different types of techniques in an attempt to transform mutation testing into a realistic and practical testing paradigm. A research conducted by DeMillo [47] was the first of its kind to summarize the research achievements and background of Mutation Testing at the preliminary stage of its development.

Fraser et al. [31] presented μTEST, an approach to generate test data for object-oriented classes based on mutation analysis automatically. Apart from the test cases, mutant-based oracles are also generated which allow the tester to check whether the expected behavior is reflected by the assertions generated. The assertions were generated by matching the test case execution on a program and its mutants in order to distinguish between them. The test cases generated are mutant-based and impact-driven aimed to minimize test cases and assessment effort. This is achieved by optimizing test cases and oracles towards detecting mutation with maximal impact.

Yao et al. [48] investigated on the causes and prevalence of equivalent mutants and how they are related to stubborn mutants. They manually analyzed 1230 mutants obtained from 18 different programs, the result shows a highly uneven distribution of mutants' stubbornness and equivalence. This means the selection of mutation operators should be carefully done because their results show that previous test effectiveness of fault seeding could be skewed. The findings of the work show that there is a contradiction to the popular assumption that equivalence is an extreme case of stubbornness. This is because it was found that equivalence is correlated with program size and the total mutants generated while stubbornness is not. Also, the findings showed that ABS (Absolute Value Insertion) operators should be discarded or at least applied with care because they generate few stubborn and many equivalent mutants. On the other hand, some operator classes like LCR (Logical Connector Replacement) are useful as they generate relatively more stubborn and fewer equivalent mutants.

Offutt et al. [25] performed a statistical regression analysis of actual programs, showing that the number of lines did not contribute to the number of mutants. Applying only the SDL (Statement Deletion) operator is a do-fewer approach known as SDL-mutation. The SDL operator was implemented for Java and its benefit was evaluated in terms of how well were the SDL mutants killed by tests generated when run on all of muJava's method-level mutants. They started by defining SDL on single statements, then extended the definition to other control structures. SDL was implemented by Mothra by replacing each statement with CONTINUE because FORTRAN has a CONTINUE statement, which only provides a placeholder. On the other hand, Java implemented SDL by commenting out each statement. It does not make sense to apply SDL to variable declaration because the mutants would not compile, to start with.

Also, applying SDL to control structures that include block(s) of statements (such as "if", "for", and "while") necessitates deleting the entire block. They generated the test cases to kill the entire SDL mutants by hand (i.e. manually). They sanitized the tests by iteratively generating them while discarding those that did not kill additional mutants strongly. The mutants that are not killed were concluded to be equivalent. This leads to the conclusion that the deleted statement has no effect on the program. They finally evaluated the SDL-adequate test set against the whole muJava's mutation operators. Other mutation operators can be discarded if the SDL-adequate test set can kill all mutants. A mutation score of 92, with 80% fewer mutants were formed. Also, 41% fewer equivalent SDL mutants were discovered.

Harman and Jia presented a detailed survey and analysis of trends and results on mutation testing. The survey comprises of works on empirical studies, optimization techniques, mutation tools, and equivalent mutation detection. The results of the survey show that mutation testing is achieving popularity as its transition from academic to industrial application is rising gradually [9]. Papadakis et al. [49] describes a systematic mapping carried out to collect techniques and approaches for test data generation in mutation testing. In 2017, Jatana et al. [50] published a systematic literature review on application of search based techniques on mutation testing. The result of the study shows that within two decades, the following techniques have been harnessed to mutation testing namely: Hill Climbing, Ant Colony Optimization, Genetic Algorithm, Bacteriological Algorithm, and Immune Inspired Algorithm.

As shown previously that some researches are mutation-based while some are GA-based, another trend of research is the application of mutation and GA in synergy to solve some problems encountered in test data generation. Below are few works that concentrate on the combination.

Bottaci was considered the first researcher to apply genetic algorithm to mutation testing [51], [52].

S. Selevakuma and N. Ramaraj [53] proposed an idea for generating a minimized test suite in test case generation using the combination of mutation and Genetic Algorithm. The idea was to resolve the problem of too many test cases to kill huge number of mutants generated by Mutation Testing. The approach models a test case as a predator while a mutant program is considered as a prey. The idea is to generate test cases to kill as many mutants as possible. The approach, mutant gene algorithm, was modeled into a tool for generating and minimizing test suites.

Sharma et al. [54] used adequacy-based testing criteria to generate test data. Mutation analysis was applied to check the adequacy of the test cases. The approach used did not follow the traditional way of applying mutation which is after the test data generation, but rather applied mutation analysis only at the period of generating test data. The approach ensures that the best data generated are adequate and the time taken is minimized because only the time taken to generate test data is included but the time to examine the adequacy is excluded. The authors applied GA to generate the test cases while validating the technique using ten real time C programs [55][56]. R.A. Silver et al. [57] presented a comprehensive systematic review on search based mutation testing. They identified five meta-heuristic techniques used to optimize test data generation, mutant generation and selection of effective mutation operators. For more details on the techniques, reader can consult their work [57].

Jatana et al. [50] presented a systematic literature review on search-based mutation testing, where they identified Ant Colony Optimization, Genetic Algorithm, to be the popularly adopted search-based techniques in optimizing mutation testing. They concluded that the techniques are used to generate test data, select, minimize and optimize generation of mutants

Analyzing the above-mentioned related literatures, a framework is identified for the classification of the research carried out in the area and the test criteria. It can be deduced that a substantial amount of work has been done on white box testing while only few work has been done on black box testing.

Table 3: Summary of Mutation-based Test Case Generation

Author	**Techniques**	**Results**	**Performance Evaluation**	**Mutant Generation**	**Test Generation**	**Language**	**Average Perf**
DeMillo and Offut [34] (1991)	Test data generation based on constraint	Test data generation based on constraint	Mutation score	Manual	Automatic	Fortran	98%
Offut et al. [58] (1999)	Dynamic Domain Reduction	Generation of test cases	NA	Manual	Automatic	Fortran	NA
Baudry et al. [59] (2005)	Genetic Algorithms	Optimization of test data generation to kill mutants	Mutation score	Manual	Automatic	C#	85%
Ayari et al. [60] (2007)	Ant Colony Optimization + Mutation score	Test data generation techniques	Mutation score	Manual	Automatic	Java	88%
Papadakis et al. [61] (2009)	Enhanced Control Flow Graph	Generation of mutation adequate test data	Path coverage	Manual	Automatic	Java	90.2%
Zhang et al.[62] (2010)	Dynamic Symbolic Execution	Automatic generation of test inputs to kill mutants	Mutation score	Manual	Automatic	C#	90%

Papadakis et al. [63] (2010)	Dynamic Symbolic Execution	Generation of effective test data	Mutation score	Manual	Automatic	C	63%
Harman et al. [64] (2011)	Execution and search-based testing	Generation of strongly adequate test data to kill first and higher order mutants	Mutation score	Manual	Automatic	C	71%
Malhotra et al. [55] (2011)	GA & mutation testing	Test data generation based on adequacy-based testing criteria	Path coverage, mutation score and generating time	Manual	Automatic	C	NA
Hanh et al. [65] (2014)	Genetic Algorithm		5 Simulink models	Manual	Automatic	Simulink	85.7%
Mohi-aldeen et al. (2016)	Negative Selective Algorithm	Generation and reduction of test cases	Path Coverage	Manual	Automatic	Java & C++	NA
Sharifipour et al. (2017)	Memtic Ant Colony Optimization and Evolution Strategy	Test data generation	Branch coverage and convergence speed	Manual	Automatic	MATLAB	NA

3.3 Research Questions

From our literature review, we could not find from the existing studies ones that handle both mutant generation and killing at the same time. This study focuses on the development of an approach using GA to generate mutants and kill them while optimizing both processes competitively. The mutator tries to generate non-trivial mutants that would be difficult to kill, while tester makes effort to generate effective test cases to kill the generated mutants. This is in form of a non-cooperative game between the tester and the mutator. The experiments carried out in this study were planned to empirically answer the following Research Questions (RQs):

RQ 1: What is the effectiveness of the generated test cases in killing the generated mutants?

This would investigate on how effective the test cases generated by the approach are. The effectiveness measure gives an insight as to how good the test cases are performing. The more mutants killed by a set of test cases, the more effective the test cases.

RQ 2: How strong are the mutants generated?

To ensure that the generated test cases are effective, there is need to ascertain that the generated mutants are non-trivial. A strong mutant is the one that is difficult to kill.

RQ 3: Is the GA-based approach better than random generation of both test data and mutants?

This question inspects the effect of the GA on the generation of test cases and mutants. It shows the role played by GA in the presented game-like approach.

RQ 4: What set of GA parameters gives the best performance with regards to our search-based mutation testing?

Answering this research question ensures the avoidance of using GA parameters by mere guessing. This is because each problem has its unique set of optimized parameters that would give the best performance.

CHAPTER 4

PROPOSED APPROACH

This chapter presents our proposed approach to generate test data and hard-to-kill mutants applying mutation testing and search-based techniques. It also explains the fitness functions applied and how the problem was formulated. The approach was developed in an attempt to bridge the gap found in the survey of the literature carried out. The detail of the critical survey is presented in the literature review chapter earlier. In this research, Mutation Testing does not only produce faulty programs for the Genetic Algorithms to optimize, but also sorts the transitional test cases with respect to the number of mutants they killed. Also, it is employed to measure the fitness values of our tests, leading to reduction in redundancy.

4.1 Methodology/Approach

The problem of generating test cases to kill the mutants is presented as an optimization problem. Consequently, an objective function also known as fitness function is designed to leverage the power of meta-heuristic techniques, like Genetic Algorithm, in generating test case data.

We harnessed the power of Genetic Algorithm to automate the procedure. We have two different contrasting GAs competing against each other. The first GA (tester) known as test-GA(tGA) creates test cases to use in the testing process while the second GA (mutator) called mutant-GA (mGA) generates mutants (i.e. faulty programs which are valid variants of the original program each with single syntactic difference).

Since GA is a general method to solve combinatorial problems, therefore the problems to be solved differ from one to another. The domain knowledge is to be considered. Before designing the GA or any metaheuristic method, there should be a designated representation scheme for the problem. In other words, designation of how the individuals would be represented in the population of GA. Below is the description of the candidate solution representation, and fitness formulation & calculation.

In this research, we propose an approach of generating test cases implementing it using Genetic Algorithms. This implementation presents an innovative way to use GAs to generate mutants in sync with test case generation. In other words, GAs are used to generate mutants of an original program and create test cases consecutively. Each player makes effort to win its opponent. The mGA generates mutants that are difficult to kill by test cases while tGA creates test cases that try to kill any mutant generated by mGA. The approach generates a subset of all the possible mutants, selecting them with mGA. This is continued consecutively until a stopping criteria or certain number of iteration is reached.

The benefit of this technique is estimated by applying it on program codes implemented in MATLAB. In order to maximize the capability of GA, its fitness function must be designed accurately and efficiently.

The steps to follow in generating mutants and analyzing their strength is represented by the flowchart depicted in Figure 5.

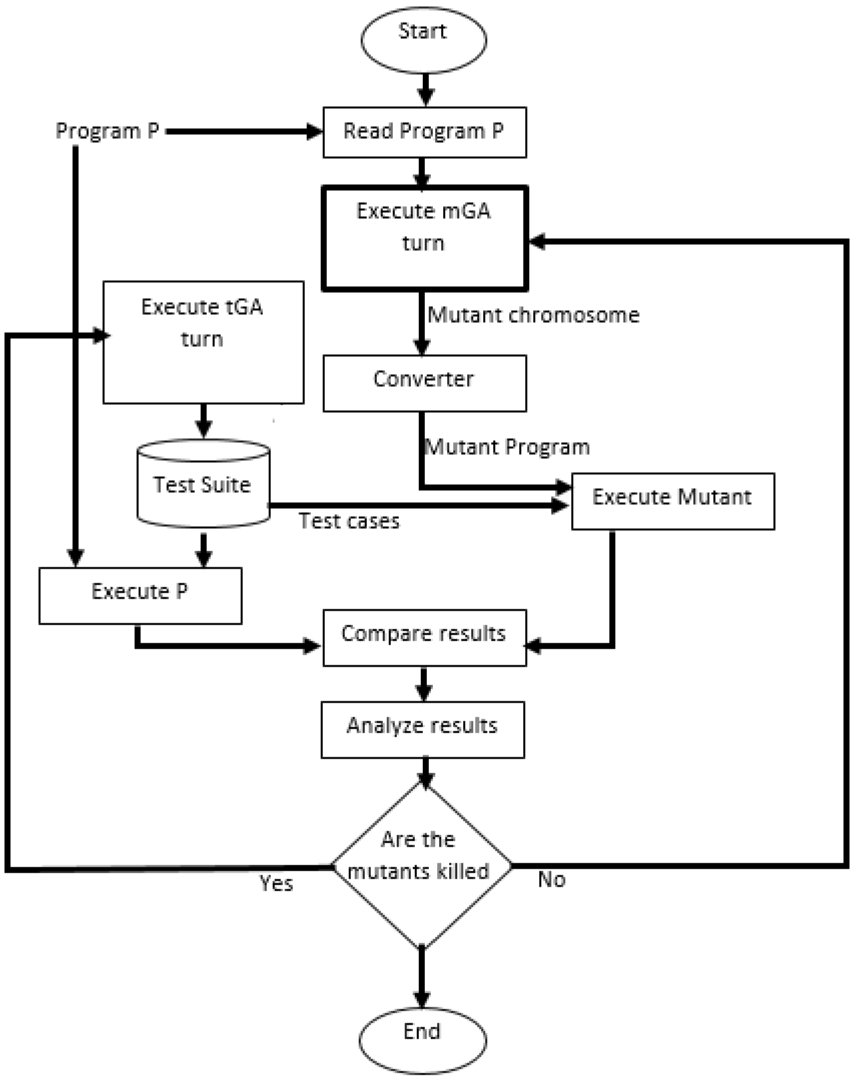


Figure 5: Flowchart of mutant generation and analysis

The original program is read to know the number of lines in the program. The number of lines in the program is used by mGA to generate mutant chromosomes that are based on the number of lines in the original program. The chromosomes are taken by the converter and transformed into the real mutant program, which is in turn executed against the test cases generated by the tGA. The original program is also executed against the same set of test cases. The result of executing original program is compared with that of the mutant, if the results are different, the mutant is killed otherwise it is not killed. Then, it has to be re-executed with different sets of test cases as this can be taken to mean the test cases are not effective initially. If the mutants are killed, there is a check to know if the end of generation is reached, in order to terminate the process. If the end of the generation is not reached, then the mGA is re-executed.

The converter in Figure 5 is converting the mutant chromosomes from the mGA to real mutant program. The flowchart for the mutant conversion by the converter is shown in Figure 6 .

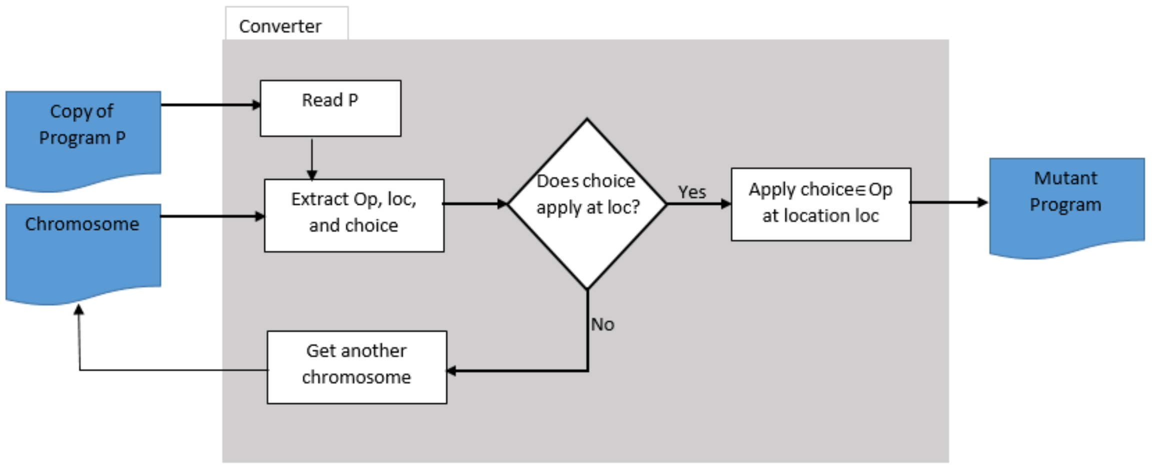

Converter
Copy of Program P
Read P
Chromosome
Extract Op, loc, and choice
Does choice apply at loc?
Yes
Apply choice∈Op at location loc
Mutant Program
No
Get another chromosome

Figure 6: Flowchart for mutants conversion

A copy of the original program and the chromosomes generated by mGA are passed to the converter module. The program is read and the chromosomes are decoded to extract the category of operator, location and the exact mutation operator represented by the chromosomes. There is a check to verify if the mutant to be generated is valid by validating the existence of the mutation category at the specified location in the program code. If the operator is present, the mutation is applied and the new variant of the original program is generated, otherwise the chromosome is regenerated. This is done to prevent invalid mutant from being produced and reduce the computational cost involved in testing the validity of the mutant later in the process.

The individual for TESTER (tGA) is the test case to be generated and the fitness is computed on the programs; while MUTATOR (mGA) generates programs as its individuals and compute the fitness values on the test cases generated by TESTER.

In this research, we designed the fitness function using Reward-Penalty approach to evaluate the population chromosome of the mGA. This means reward is assigned to good chromosomes while a penalty is tasked against the poor chromosomes. Since the function of mutation GA is to generate mutants, which are valid variants of the original program, after applying a particular mutation operator at a specific location. This means two things are involved in generating mutants, i.e. mutation operator and the location. In addition to mutation operator and the location, the actual operator is also of paramount importance.

Each mutant is evaluated by computing fitness function on it. Due to the fact that this is a black-box approach, we need to execute the mutant against the test suite. The outcome of the execution can be represented in an execution matrix. If the number of mutants in the population is M and the number of tests cases in the test suites is T, then the dimension of the execution matrix would be M × T as shown below. The approach was first applied by Domínguez-Jiménez et al. [42].

$$(m_{ij})_{M\times T} = \begin{pmatrix} 0 & 0 & 1 & \dots & 1 \\ 1 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 2 & 2 & 2 & \dots & 2 \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix} \qquad (1)$$

m_{ij} is 1 or 0 when the mutant i is executed by test j is killed or alive respectively. Where 0 denotes mutants that are alive while 1 denotes killed mutants.

The fitness function comprises of the number of test cases that are able to kill a mutant and the number of mutants killed by the particular tests. In this case, the fitness of a particular mutant (let's say μ with the collection of test sets T) is given by:

$$\text{Fitness}(\mu, T) = M \times T - \sum_{j=1}^{T}\left(m_{\mu j} \times \sum_{i=1}^{M} m_{ij}\right) \qquad (2)$$

where $\sum_{j=1}^{T} m_{ij} \in \{0.\ldots.T\}$, for all i and

$\sum_{i=1}^{M} m_{ij} \in \{0.\ldots.M\}$

Therefore the value of the fitness will continuously be within [0, M×T].

The significance of this fitness function is that it penalizes every group of mutants that are killed by the same set of test cases without taking into consideration the mutation operator, location of mutation, or the total number of mutants in the group.

4.2 Mutant Fitness Function

The evaluation of mutants is carried out as follows:

Each mutation operator is evaluated such that higher fitness is assigned to operators whose mutants are potentially strong and difficult to kill given a test suite.

Each operator OP_i is assigned a constant probability (say C), such that each operator has 50% chance of being picked as shown in the Table 4.

Table 4: Initial Probability of operators

Operators	Probability
OP_1	C_1
OP_2	C_2
OP_3	C_3
…	…
OP_N	C_N

After the selection of the operators and the corresponding mutants are generated, the number of mutants for each operator is computed and the respective probabilities score of each operator are updated with new probability (say P) which is later normalized. This is shown in the Table 5:

Table 5: Updated Probability of operators

Operators	Probability
OP_1	P_1
OP_2	P_2
OP_3	P_3
…	…
OP_N	P_N

To compute the updated probability of each mutation operator, there is need to know the number of mutants by the operator and the number of mutants that are difficult to kill. For example:

In each iteration i,

$$increment = 2 * \frac{nMutSamp - nKilled}{nMutSamp} - 1 \quad \in [-1.1] \qquad (3)$$

Where nMutSamp = Number of mutants generated by OPi at iteration i

nKilled = Number of killed mutants generated from OPi

If no mutant is killed,

Then $\frac{nMutSamp-nKilled}{nMutSamp} = 1$ (4)

∴ increment = (2 * 1) – 1 = 1 [Maximum value]

But if all the mutants are killed,

Then $\frac{nMutSamp-nKilled}{nMutSamp} = 0$ (5)

∴ increment = (2 * 0) – 1 = -1 [Minimum value]

This means that any value of increment would be [-1,1].

$$\text{Probability Score} = \begin{cases} C. & \# Iteration = 1 \\ P. & \# Iteration > 1 \end{cases} \qquad (6)$$

So if the number of iteration/generation is 1, i.e. MaxIt=1

$$P = C + \frac{increment}{totalMutant} \tag{7}$$

But if the number of iteration, MaxIt=2,

$$P = C + \frac{increment}{2 * totalMutant} \tag{8}$$

Generally, $$P_i = P_{i-1} + \frac{increment}{MaxIt * totalMutant} \tag{9}$$

This would force the final probability of the mutation operator, $P_{op} \in [0,1]$.

If more than half of the mutant sample are killed, increment would be negative thereby decreasing the value of the next probability. In other hand, if less than half of the mutant sample are killed, increment would be positive thereby increasing the value of the next probability of the same mutation operator. This can be represented mathematically below:

$$0 < nKilled < nMutSamp/2 \tag{10}$$

$$nMutSample/2 < nKilled \leq nMutSamp \tag{11}$$

If the two equations above are evaluated to TRUE, they would cause the values of increment to be positive and negative respectively.

In other words, we would ensure the $0 \leq P_{op} \leq 1$

The probability score is summed and normalized to 1.

$$\text{i.e.}\frac{1}{const}\sum_{i=1}^{nOp} P_i = 1 \; such \; that \; P_{norm} = \frac{P_i}{\sum_{i=1}^{nOp} P_i} \qquad (12)$$

The proportion of each operator is used to give precedence to the generated mutants together with the line of the program in the next generation.

Similarly, we have probability of killing programs mutated at a particular line of code. We also update the probability of not killing a program mutated at certain line number just like we did for the mutation operator.

Table 6: Probability of program line number

Line Number	Probability
1	P_1
2	P_2
⋮	⋮
L	P_L

Where L is the number of line of the program under test. Since no deletion nor insertion of statement operator is used, the number of line of the original program is the same as the number of line of each mutant.

The probability of line number is also normalized as follows:

$$P_{\text{line}} = \frac{p_j}{\sum_{j=1}^{n} p_j} \qquad (13)$$

Where j = 1,2,…,n, p_j is the probability of line j and n is the number of line of the program.

The final fitness of the mutant in question is the combination of its mutation operator's probability score and its line number probability score. But in order to keep the fitness normalized to 1, the summation is averaged to give a single number between zero and one.

i.e. $\text{Final Fitness} = \frac{1}{2}(P_{op} + P_{line})$ (14)

The fitness described above is the fitness of a mutant. The overall fitness of the set of mutants can be obtained by computing the average of the entire mutants' fitness.

Therefore, a mutant is rated by the mutation operator it has and the position of the mutation in the original program.

4.3 Test Fitness Function

The fitness function of the test cases is derived from the test execution matrix, which is obtained after executing the entire mutants with the whole test cases. In other word, each mutant is executed against every test case. The fitness of a test case is dependent on the competition with the rest of the test cases. This means the fitness of a test case can affect the fitness of others. The approach is explained as follows. Table 7 shows the sample of the execution matrix as shown below:

Table 7: Test Execution Matrix

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
M1	0	0	0	1	0	1	0	0	1	0
M2	0	0	1	0	0	1	0	1	0	0
M3	0	0	1	0	0	1	0	0	1	1
M4	1	0	0	0	1	1	1	1	0	0
M5	0	1	0	0	0	1	0	0	1	0
M6	1	1	1	1	1	1	1	1	1	1

M7	0	0	1	0	0	1	1	1	0	1
M7	1	1	0	0	0	1	0	1	1	0
M9	0	0	0	0	0	0	0	0	0	0
M10	1	0	0	0	1	1	0	1	0	1

The value in each cell of the matrix can be [0,1]. If it is ZERO, it means the mutant was not killed by the test case represented by the intersecting column of the matrix. In other word, the value is ONE if it is killed by the test case. Therefore, the initial values are [0,1].

The values are then updated by looking into how many test cases kill a mutant and how many mutants are killed by a particular test case.

Using the example above, the table would be modified and updated resulting in the values shown in Table 8 :

Table 8: Updated test execution matrix

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
M1	0	0	0	0.333	0	0.333	0	0	0.333	0
M2	0	0	0.333	0	0	0.333	0	0.333	0	0
M3	0	0	0.25	0	0	0.25	0	0	0.25	0.25
M4	0.2	0	0	0	0.2	0.2	0.2	0.2	0	0
M5	0	0.333	0	0	0	0.333	0	0	0.333	0
M6	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
M7	0	0	0.2	0	0	0.2	0.2	0.2	0	0.2
M7	0.2	0.2	0	0	0	0.2	0	0.2	0.2	0
M9	0	0	0	0	0	0	0	0	0	0
M10	0.2	0	0	0	0.2	0.2	0	0.2	0	0.2

Each test cases is evaluated from the values in Table 8. This is because, for example, mutant M1 was killed by three test cases (T4, T6, and T9). Therefore, they share the point among each test cases. It should be noted that killing a mutant is rewarded one point (1 point).

Since three of the test cases killed the mutants, they share it equally and each test case gets 1/3 (0.3333). More so, for mutant M3, four of the test cases killed the mutants; therefore, each test gets reward of 0.25 (1/4). Also, all the ten test cases killed mutant M6; resulting to a reward of 0.1 (1/10) for each test case. So the aggregate point of each test case is computed by adding up the total point by the test against each mutant and assigned to each test as shown in Table 9 :

Table 9: Score of test cases

Test	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
Point	0.7	0.633	0.883	0.433	0.5	2.15	0.5	1.233	1.217	0.75

The test cases are now sorted based on their computed power of killing the mutants.

The fitness of each test case in Table 9 can be sorted to result in the sequence shown in Table 10 .

Table 10: Sorted test cases with respect to their killing power

Test	**T6**	**T8**	**T9**	**T3**	**T10**	**T1**	**T2**	**T5**	**T7**	**T4**
Point	2.15	1.233	1.217	0.883	0.75	0.7	0.633	0.5	0.5	0.433

$T(j)_{fit}$ = Fitness point of test case j

Test suite Fitness = sum of the fitness points as shown in equation (15)

$$\sum_{j=1}^{n} T(j)_{fit} \qquad (15)$$

The total point of the test suite was 8.999, therefore the overall fitness of the test suite is ≈9. The actual point was less than 9 because of the several rounding made in the process of computing the fitness. This can be verified by cross-examining the test execution matrix in *Table 7*. It can be seen that only mutant M9 was not killed by any test case in the test suite. In this case, out of 10 mutants, only 9 mutants were killed. Since the minimum and maximum values for the test suite fitness value are 0 and 10. In general, the range of values the fitness can have is [0, m] where m is the total number of mutants.

This fitness computation of test suite is similar to mutation score. Dividing by the total number of mutants, the fitness becomes mutation score. This is because the mutation score of the test suite is 0.9 which signifies 90% of the mutants being killed. This is accurate because only 9 out of 10 were killed which is exactly 90% of the total mutants. However, this fitness is better than mutation score because mutation score computes only the overall effectiveness of the test suite without knowing which of the test cases performs better than the other. Our fitness computation helps in differentiating between the test cases in the test suite. The test cases that kill more alive mutants and difficult-to-kill mutants get more fitness. This made it easy to apply GA to prioritize the test cases based on their fitness so that they can be propagated to the next generation expecting to have more mutants being killed by those effective set of test cases.

4.4 Mutants Generation

As mentioned above, the GAs utilize binary digits to represent their chromosomes. Before mutation[2] can take place, there is need to define the mutation operators and the location where the mutation is to take place.

In this work, mutation analysis is performed using an approach called evolutionary mutation testing [42]. A Genetic Algorithm is used to generate encoded mutants used in generating possible mutants for carrying out the analysis – mutants' generator. The number of live mutants generated is reduced gradually as they are killed by test cases by favoring the strong mutants, which can be a useful tool to improve the quality of the test suite initially created using test case generator. Mutants are encoded as individuals of the algorithm, which implies the encoded mutants are generated and their fitness values are used to select those that would transit to the next generation. Since this testing technique is a black-box oriented, the encoded mutants have to be decoded and executed against the set of test cases generated initially. Subsequent generation of the mutants is instigated and affected by the quality of test suites. Before carrying out the mutation testing, an original program is obtained and the correctness of the program is ascertained. Also, the list of mutation operators to be applied should be identified. The generation of mutants of the original program is encoded using three fields (as shown in Figure 7) so as to be acceptable by the genetic algorithm.

Operator	Line Number	Choice

Figure 7: Representation of mutant chromosome

[2] Note that the mutation here is not the one in genetic algorithm.

An identifier of the mutation operator to be applied is represented by Operator. Line Number signifies the line number of the original program where the mutation operator is applied while Choice specifies the particular replacement to be performed where there are multiple options (e.g. +, -, *, /, etc).

So as to make the mutants generation guided, each encoded mutant is formed after computing the values a field can take in the specified program. The actual mutants are then produced by a converter from the encoded fields from the table above. The mutant generator takes the encoded mutant chromosomes and encoded test cases from mGA and tGA respectively together with the original source code under test. From the encoded mutants, mutation operators and line number of mutation are extracted. The Mutant Generator creates the mutants using the mutation operators and the result of this Generator is mutant in form of MATLAB m-file containing only one fault in each mutant. Each mutant is produced after randomly generating individuals at the initial stage. Crossover and mutation operators are applied to the randomly generated mutants to form children individuals. The generated individuals are then evaluated and the more fit ones are made to transit to the next generation. The crossover operator ensures exchange of content of one field of an individual with the other in a systematic way. The operator is designed to evade invalid individual generation. The following are the valid mutant chromosomes for QuadraticSolver Program:

Table 11: Valid mutants for QuadraticSolver program

0001100	0001101	0001110	0001111
0010000	0010001	0010010	0010011
0011000	0011001	0011010	0011011

0100000	0100001	0100010	0100011
0101100	0101101	0101110	0101111
0110000	0110001	0110010	0110011

4.5 Mutant Program Generation

After mutant GA generates the chromosomes representing the genotype of the typical mutant program, there is need to have the phenotypical depiction of the program. A converter program was created that takes the binary representation of the mutant as shown above. The chromosome (binary representation) encodes the operator, line number and the choice of operator in each category. The converter takes the chromosome and the original program and decodes the chromosome based on the original program. This generates a mutant program, which is used in the actual execution. Figure 6 shows the concept in high level. The converter decodes the mutant chromosome, extracting the operator (for example, Arithmetic Operator Replacement-AOR), the location where the mutation operator is to be applied and the actual operator to be applied (for example, addition [+]). These are all extracted from the encoded mutant. The converter reads the original file and gets a copy of it. After obtaining the mutation operator category, location and the exact operator, the converter applies them to the copy of the original program. This is done by locating the line number of the program copy using the mutation location encoded in the chromosome and applying the exact operator (which belongs to the category specified by the chromosome) and replaces the original operator by the encoded one. This results in a new valid program similar to the original program except the replaced operator. This makes it become a first order mutant of the original program. Both mutant and the original program can then be executed using the same test case to investigate if the test case can kill the mutant. The same sequence of events is repeated for every chromosome in the population. In order to prevent wastage of memory, no two mutants exist at the same time. In other words, only one mutant is available at any particular time during the execution process. The creation of the subsequent mutants is carried out and the file is saved as the previous mutant file name. This makes the execution a bit easier and the memory wastage is avoided. For example, take QuadraticSolver program in Appendix B to illustrate the procedure of how the real mutant is generated. Given a mutant chromosome, which has been decoded to give the mutation operator category as AOR, location as 3 and the actual operator as '+'. This makes the line 3 of the original program to be read.

```
        d = sqrt(b^2-4*a*c);
```
(16)

And one of the arithmetic operators (say '-') in the program line is replaced with the encoded operator, the program statement becomes

```
        d = sqrt(b^2+4*a*c);
```
(17)

This single modification makes the program to be different from the original one. The mutant GA that generates the mutant has been guided so that it generates only valid chromosomes considering the original program. This means a mutation operator and a location are joined in a single chromosome if only the operator category exists in the line code number represented by the location extracted from the encoded mutant chromosome.

For the sake of simplicity, consider the following running example. Given a program to compute roots of a quadratic equation (see Appendix B), the program has 14 LOC. Arithmetic Operator Replacement (AOR) can be applied to lines 3, 5, 6, 8, 11, and 12. While Relational Operator Replacement (ROR) can be applied to lines 4 and 7. The total number of lines in the program is encoded into binary as 1110. This means four bits are sufficient to encode the line number. AOR and ROR are encoded as 0 and 1 respectively. Changing addition (+) into subtraction (-), multiplication (*), division (/) and exp (^) can be encoded as 00, 01, 10, and 11 respectively. Also, replacing (>) into (<), (<=), (>=), and (!=) are encoded as 00, 01, 10, and 11 respectively. The leading '0' of a mutant chromosome represented by "0010110" would imply selecting AOR as the category of the operator to be performed. Then 0101 means line number 5 would be mutated and 10 means the addition (+) operator would be replaced by division (/) operator. This would change the statement [x(1) = (0-b + d)/(2*a)] into [x(1) = (0-b / d)/(2*a)].

[x(1) = (0-b + d)/(2*a)]   Δ [x(1) = (0-b / d)/(2*a)]

It should be noted that the mutation is done based on the actual mutation operator, but not the category.

4.6 Test Case Generation

One of the GAs, tGA, is responsible for the generation of test cases and each chromosome in the population is representing a test case to be used in this experiment, i.e. the execution of programs (original and its mutants). As usual with GA, an initial set of population is generated randomly taking into consideration the format of the individual representation, which is a sequence of binary strings in our case. In subsequent generations, the test cases are guided so that more effective test cases evolve to next generation by selecting the more fit individuals based on the fitness function of the test cases. Therefore, each individual denotes an element in the set of test cases in which its fitness depends on its effectiveness. Test cases need to have high efficiency as well. This is the ratio of the number of mutants killed to the total number of test cases. The individual chromosome is made up of strings of binary digits which is the concatenation of different substrings in which each substring represents the input of the program under test. For example, 1010100001001100101 is a sample of a test case to be generated. If the test case is a combination of three input values, the test case can then be represented as follows:

Input 1	Input 2	Input 3

Figure 8: Representation of test chromosome

This means each input value is a string of binary digits, so joining them together forms the chromosome. After generating the individual chromosomes, they are analyzed by decoding them in order to obtain the values of each input. The chromosome represented above can be decoded as **101010 0001001 100101** so that **101010** is the input 1 while **0001001** is the input 2 and **100101** is the input 3. The first bit of each input is the sign of the input while the remaining bits are used to form natural numbers. The input value is positive if the first bit is 1 and negative if it is zero (0). The above input can be decoded as follows:

Input 1: 101010	**→**	**1**	**01010**	**→**	**+11**
Input 2: 0001001	**→**	**0**	**001001**	**→**	**-10**
Input 3: 100101	**→**	**1**	**00101**	**→**	**+6**

The test case represented by the chromosome above is depicted as (11, -10, 6).

4.7 Selecting The GA Parameters

Every experiment on GA conducted is by selecting parameters of the GA to optimize its performance. Some are trial and error while others are chosen based on user-experience.

Most of the researches are reported with the parameters without explaining the reason why and how they are chosen. In such situation, the performance of the GA cannot be maximized. The parameters of the GA depend on the specific problem. A combination of such parameters may be useful for a problem and not for another problem. We looked at the different combination of parameters and observed the effects on the results. The values of the parameters can be maximized using an approach based on Taguchi Experimental Design for the parameter tuning. GA parameters are divided into two categories namely: structural and numerical parameters.

Structural parameters: it is challenging and difficult to deal with these set of parameters in any GA application. They dictate the structure of GA, as the name implies. The coding pattern of GA demands substantial modification if any of these parameters are changed. Examples of these parameters include coding scheme representation, types of operator and stopping criterion. For example, the one-point crossover can be applied to knapsack problem but cannot be applied to sequence representation.

On the other hand, numerical parameters involve changing the values of some factors affecting the performance of the GA. Example of the main factors considered as numerical parameters are population size, maximum iteration (generation), type of initial population, mutation and crossover probabilities. Altering these parameters does not involve recoding of the GA, it only results in changes in GA performance.

The choice of mutation probability depends on the desired outcome. For instance, if the application desires all members to have very high fitness, a lower mutation rate is suggested so as to have a less likelihood of disrupting good solutions. But if simply one or two highly fit individuals are required, a higher mutation rate may be chosen especially if ensuring good coverage of the search space is given preference over the cost of disrupting copies of good candidate solutions. In this case, we decided to make mutation rate low because we need to have several highly fit individuals (difficult-to-kill mutants).

In this research, a set of structural parameters is selected because of its suitability to the problem under investigation. Each of the GAs has certain degree of overlapping on numerical parameters with the other as shown in Table 12 and Table 13.

Table 12: Tester GA Parameters and Levels

Parameter	**Code**	**Level**			
		1	**2**	**3**	**4**
Selection Function	A	Roulette wheel	Tournament	-	-
Crossover function	B	Single point	Two point	-	-
Crossover probability	C	0.75	0.8	0.9	-
Mutation Probability	D	0.35	0.3	0.25	-
Population size	E	20	30	40	45

Table 13: Mutation GA Parameters and Levels

Parameter	**Code**	**Level**			
		1	**2**	**3**	**4**
Selection Function	A	Roulette wheel	Tournament	-	-
Crossover probability	B	0.75	0.8	0.9	-
Mutation Probability	C	0.1	0.08	0.06	-
Population size	D	35	40	45	50

We identified five (5) parameters which need to be tuned to know the optimal values of each one of them. The parameters are: selection function, crossover function, crossover probability, mutation probability, and population size. For mGA, not every single point crossover nor every two-point crossover generates valid chromosomes, so we decided to remove crossover function from the set of parameters to be optimized because we applied a customized crossover function. Therefore, four (4) parameters were optimized in the case of mGA as shown in Table 13 above.

Based on the number of parameters considered and number of parameter levels identified, the detail of the experimental design and levels for the tGA is shown in the Table 14 :

Table 14: Experimental Design for Tester GA Parameter Selection

Experiment	**Parameter of GA**				
	A	**B**	**C**	**D**	**E**
1	1	1	1	1	1
2	1	1	1	1	2
3	1	1	1	2	3
4	1	1	2	2	4
5	1	2	2	3	1
6	1	2	2	3	2
7	1	2	3	1	3
8	1	2	3	1	4
9	2	1	1	2	1
10	2	1	1	2	2
11	2	1	1	3	3
12	2	1	2	3	4
13	2	2	2	1	1
14	2	2	2	1	2

15	2	2	3	2	3
16	2	2	3	2	4

The fitness is the overall fitness of the test suite. The computation of the overall fitness has been discussed.

Similarly, Table 15 shows the number of parameters considered alongside the number of parameter levels for mGA.

Table 15: Experimental Design for Mutator GA Parameter Selection

Experiment	**Parameter of GA**			
	A	**B**	**C**	**D**
1	1	1	1	1
2	1	1	1	2
3	1	1	2	3
4	1	1	2	4
5	1	2	3	1
6	1	2	3	2
7	1	2	1	3
8	1	2	1	4
9	2	1	2	1
10	2	1	2	2
11	2	1	3	3
12	2	1	3	4
13	2	2	1	1
14	2	2	1	2
15	2	2	2	3
16	2	2	2	4

The fitness here is also the overall fitness of the mutants generated. And this should not be taken as the fitness of a single mutant. The computation has been discussed in the subchapter 4.2.

CHAPTER 5

EXPERIMENTS, RESULTS AND DISCUSSION

This chapter explains the experiments carried out to implement the approach presented in the previous chapter explaining the design of the experiment. It also presents how the GA-based test data generator is implemented, not only the design but also the setup and the implementation. The power of the operators and parameter settings are investigated by carrying out several experiments with various settings. The results obtained from the experiment were presented, discussed and analyzed. This section explains the experimental setup and the evaluation of the results. The experiments were implemented and executed on a PC with intel® CORE™ i5-4200U CPU @ 1.60GHz processor, 6GB RAM and 64-bit Operating System, x64-based processor running Windows 10 Operating System. The MATLAB version used was R2015a (8.5.0.197613). MATLAB is one of the versatile high-level languages and easy to handle. Its advanced data analysis, visualization and toolboxes provide user with the necessary means to present and discuss their experimental results. In this section, the results are discussed in details.

5.1 Experiment Design

It is well acknowledged that obtaining good values of parameters for good GA performance is essential as it is one of the challenges of GA. However, little work has been recorded on investigating how GA parameters affect the performance and how they are tuned. Most of the practitioners select default values that are chosen by conventions; for example, low mutation rate. Mostly, GA parameters are selected through user-experience and trial-and-error as mentioned earlier. It is imperative to investigate the effect of combining different crossover rates and mutation rates. This is because different problems have dissimilar properties and for this reason, distinct parameter sets are required. This section focuses on parameter tuning. We investigated 5 different parameters that can influence the performance of the GA as follows: selection function, crossover function, crossover probability, mutation probability and population size.

In this research, 5 different experiments were conducted using MATLAB programming environment. We selected five (5) MATLAB program codes as experimental subjects of different purposes, sizes and complexity. Most of them were taken from textbooks and research papers and adapted to MATLAB format while others were written from scratch.

5.2 Description of Programs Under Test

The programs used for the experiment are described as shown in Table 16 . Each program is described by its inputs and outputs and what it does.

Table 16: Description of Programs under Test

Name of Program	**Description**	**Number of Line**	**Inputs**	**Outputs**
QuadraticSolver	To find the roots of equation $ax^2+bx+c=0$ by analyzing its parameters/coefficients	14	Three coefficients: a, b, c	Two different solution, One solution (two identical

				solutions), No real solution
TriType	To determine the triangle type by evaluating all its three sides. The relationship between the sides gives the type of triangle they represent	14	Three sides: a, b, c	Scalene, Equilateral, Isosceles, Right-angled, Non-triangular
MID	Find the middle number from the list of three numbers	20	Three numbers x,y,z	The mid number
Line-Rectangle Classifier	To determine relative positions relationship between a line and a rectangle	36	Line coordinates: (xl1,xl2,yl1,yl2), Rectangle coordinates: (xr1,xr2,yr1,yr2)	Error, Line wholly outside rectangle, Line wholly inside rectangle, Line partially inside rectangle
Point-Circle Classifier	To establish the relationship of the circle and a given point based	12	Center coordinates: (x,y) Radius: r,	Inside the circle, On the circumference of

	on their position by taking into consideration the coordinates of the given point, center coordinates and radius of the circle		Point coordinates: a,b	the circle, Outside the circle

For this experiment, FIVE test programs were chosen as Program Under Tests (PUTs) as shown above. The subject programs were implemented in MATLAB and PUTs were used for the experiment.

5.3 Results and Discussion

This section discusses processes followed to select the parameters used in the experiment. The results are discussed and the optimal set of parameters were selected. The experimental results of the test cases generation to kill generated mutants were presented for each program under test. This comprises of the results of executing the test cases generated by tGA against the mutants generated by mGA. Randomly generated test cases were also executed against optimized mutants and optimized test cases were on the other hand executed against randomly generated mutants.

5.4 Parameter Selections for the GAs

The design of the experiment to select the suitable and optimal GA set of parameters is shown in Table 14 and Table 15. QuadraticSolver program is used in the parameter selection experiment. The results of the experiments are the fitness of the test suite, which is similar to mutation score. The fitness of test suite becomes the mutation score as soon as it is divided by the number of mutants. Each experiment is carried out ten times in order to calculate the confidence interval of each result. The results of the ten-time running of the experiment is shown in Appendix C.

The confidence interval of each of the experiment is calculated using mean with 95% confidence intervals. The plot for the confidence interval is shown in Figure 9 :

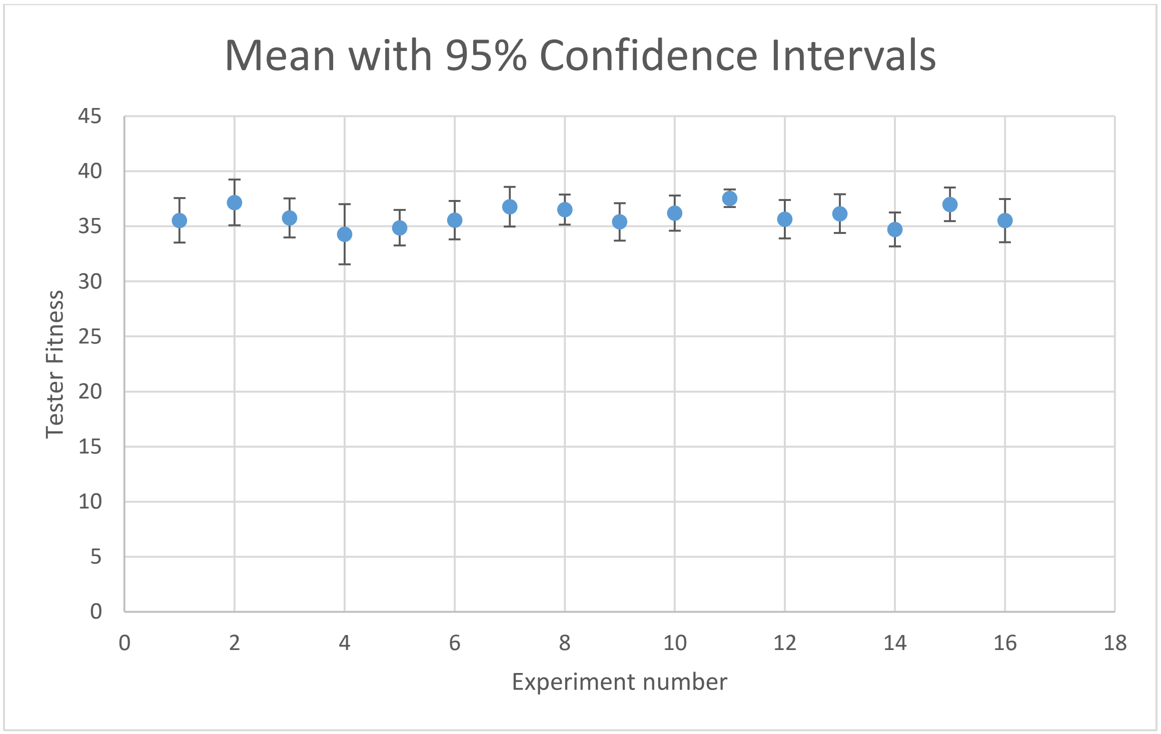


Figure 9: Plot of Tester GA Confidence Interval for parameters selection

By studying the plot above, it can easily be seen that the eleventh experiment is better than any of the other experiments. Although, it can be seen that most of the experiments were centered around 35, which means 35 out of 40 mutants were killed by the test cases. Details of how the value is obtained is in Section 4.3. This is due to the fact that it has the highest mean fitness and lowest error interval or less deviation. This resulted in selecting the parameter set of experiment 11 for the tester GA. The corresponding parameter set for the experiment is shown in Table 17.

Table 17: Selected parameters for tester GA

Selection function = Roulette wheel
Crossover function = Single point
Crossover Probability = 0.75
Mutation Probability = 0.25
Population size = 40

Similarly, experiments were carried out to investigate the best set of parameters for mGA. The results of the experiments run ten times are shown in Table 18 . Each of the results shows the highest.

Table 18: Results of experiment to select the best parameter set for mGA

Par/Run	1	2	3	4	5	6	7	8	9	10
1	0.8044	0.9444	0.9583	0.5	0.7593	0.4615	0.4217	0.8333	0.8788	0.8152
2	0.9333	0.8462	0.9091	0.8947	1	0.8947	0.7647	0.8077	0.7692	0.5522
3	0.8444	0.9474	0.8	1	0.9333	0.9412	0.8621	0.8235	0.875	0.8524
4	0.9444	0.9474	0.75	0.8846	0.9286	0.7143	0.7931	1	0.7143	0.9412
5	0.8824	0.7368	0.7778	0.6098	0.8	0.7273	0.5085	0.7143	0.9375	0.9286
6	0.8235	0.7838	0.8333	0.9	0.85	0.9286	0.9444	0.8462	0.875	0.8947
7	0.8667	0.9235	0.8764	0.9824	0.8867	0.9087	0.9129	0.7659	0.8739	0.8255
8	1	0.72	0.7647	0.8824	0.8824	1	0.9375	0.8966	0.8766	0.9167

9	0.5	0.7391	0.7188	0.7273	0.5556	0.7727	0.7619	0.6829	0.6667	0.963
10	0.9091	0.9231	0.8824	0.9375	0.8824	0.9333	0.8	0.6667	0.9167	0.8125
11	0.8889	1	0.8235	0.8182	0.8333	0.8762	0.8939	0.8884	0.9963	0.8698
12	0.8977	0.7898	0.8235	0.9918	0.9538	0.8759	0.8538	0.7965	0.7997	0.9418
13	0.8929	0.8462	0.8462	0.9063	0.7879	0.3636	0.9091	0.7333	0.7692	0.7282
14	0.9598	0.8754	0.9915	0.7899	0.8459	0.9985	0.8545	0.8762	0.9105	0.8965
15	0.8987	0.9476	0.8798	0.7895	0.8545	0.9512	0.7789	0.7548	0.7985	0.9055
16	0.7744	0.9611	0.7016	0.8659	0.8873	0.8346	0.9113	0.8674	0.8468	0.9861

The results shown in Table 18 is plotted to find the confidence intervals of each of the experiments using mean with 95% confidence intervals. The plot for the confidence interval is shown in Figure 10.

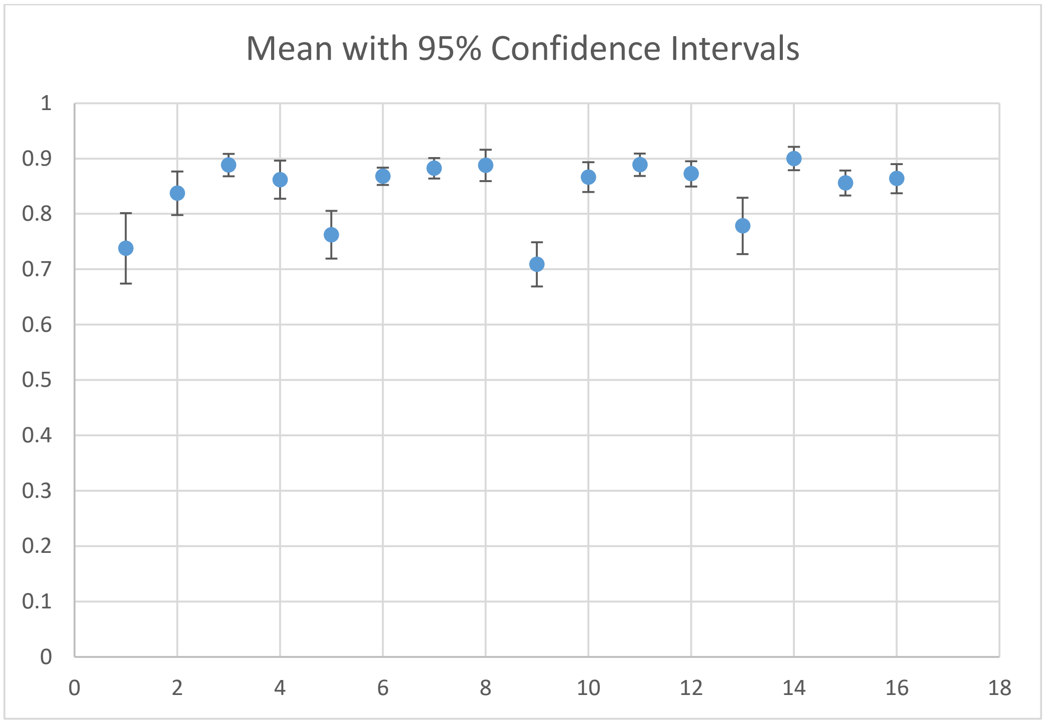


Figure 10: Plot of mGA Confidence Interval for parameters selection

Following the results shown in Figure 10, it can be seen that experiment 14 has the highest mean and relatively small length of error bars. The confidence interval for experiment 14 is the best among them as it shows the highest mean fitness of

0.8999. Experiments 7, 8, 3 and 11 have mean fitness of 0.8823, 0.8877, 0.8879 and 0.8889 respectively. Therefore, the best parameter combination for mGA is considered to be the parameters corresponding to experiment 14. These parameters are shown in Table 19.

Table 19: Selected parameters for mutant GA

Selection function = Tournament
Crossover probability = 0.75
Mutation probability = 0.1
Population size = 40

5.5 Discussion of the results of Experiment

In this section, the results of the experiment are discussed. Each program under test is used separately to carry out the experiment and the individual results are shown and discussed as follows.

5.5.1. QuadraticSolver

This program has 14 LOC with three branches. The GAs run for 100 generations and the result is shown in Figure 11.

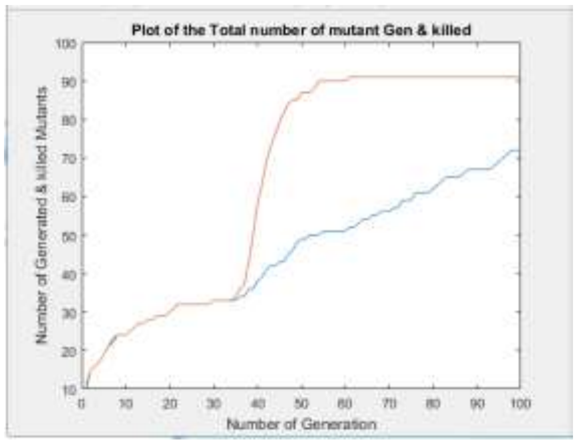


Figure 11: Total and Killed mutants for QuadraticSolver using 100 generations

By studying the result above, one can easily conclude that the GAs need more generations as the result for the number of killed mutants is yet to converge and looks promising (i.e. if more generation/time is allowed, more mutants would be killed). The percentage of the killed mutants and the number of unique tested mutants are shown in Figure 12.

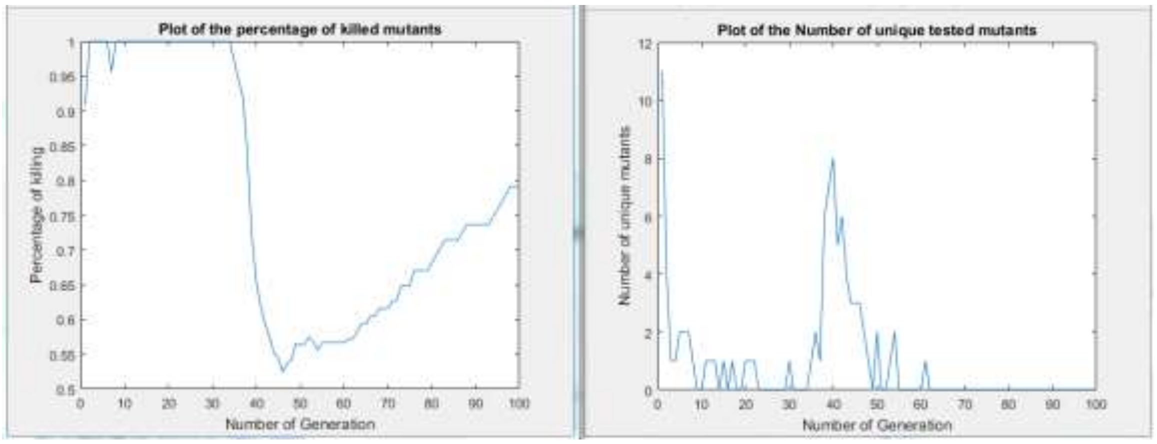


Figure 12:Killed mutants and unique mutants for QuadraticSolver (100 Generations)

The experiment was repeated 32 times and the best result was plotted in all the experiments. The percentage of the average number of killed mutants shown in Figure 12 also demands for increase in the number of generation. The number of generations was then increased to

150 expecting the number of unique tested mutants to increase over time. The result is shown in Figure 13 :

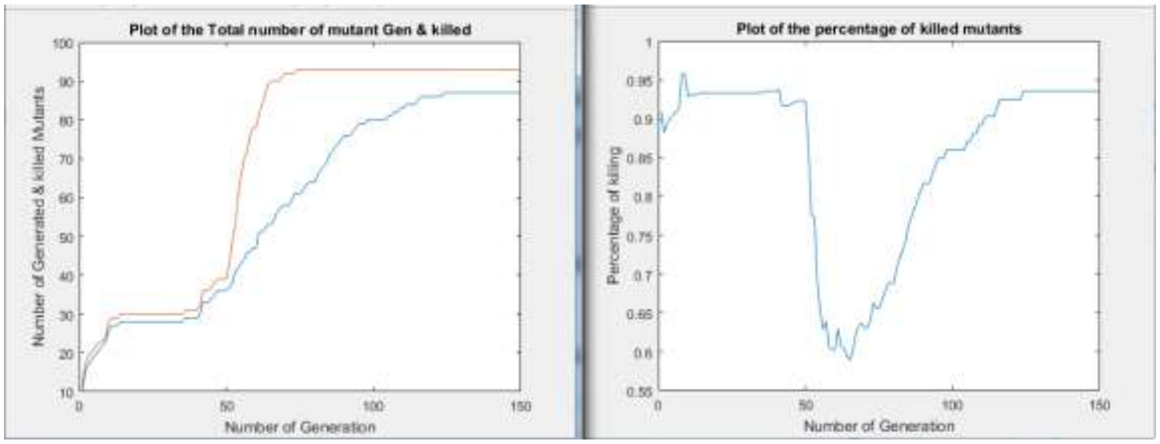


Figure 13: Killed mutants and unique mutants for QuadraticSolver (150 Generations)

The result in Figure 13 shows that the total (cumulative) unique mutants generated, on average, keep increasing until after 75 generations, then there was no new mutant generated. In other words, the mutants generated after 75 generations were already generated mutants. (This is shown by the upper curve of the left graph). The lower curve shows the number of total mutants killed across the generations. With increase in the number of generation, more mutants are being killed; this shows that the effectiveness of the test suite is improving in every next generation until when the generation reached 125, when no more mutants were killed. The graph on the right of Figure 13 shows the proportion of the uniquely killed mutants out of the total generated mutants in each generation. The plot shown in Figure 14 is another way to show the upper curve of the left graph in Figure 13. It shows the number of unique mutants generated in each generation (not total).

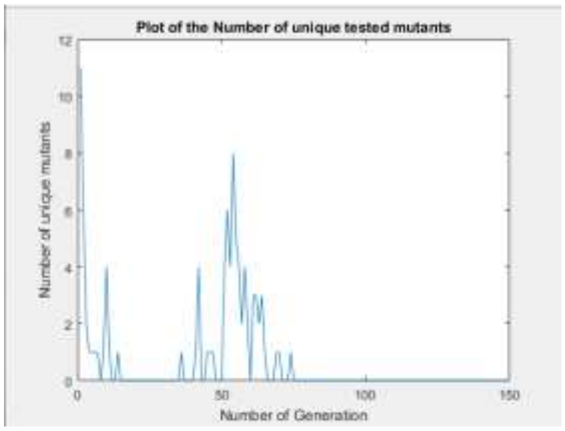


Figure 14: Number of unique tested mutants for QuadraticSolver (150 Generations)

As the number of generation reaches 75, no new mutant was generated.

The number of generation was further increased from 150 to 200. The result is shown in Figure 15.

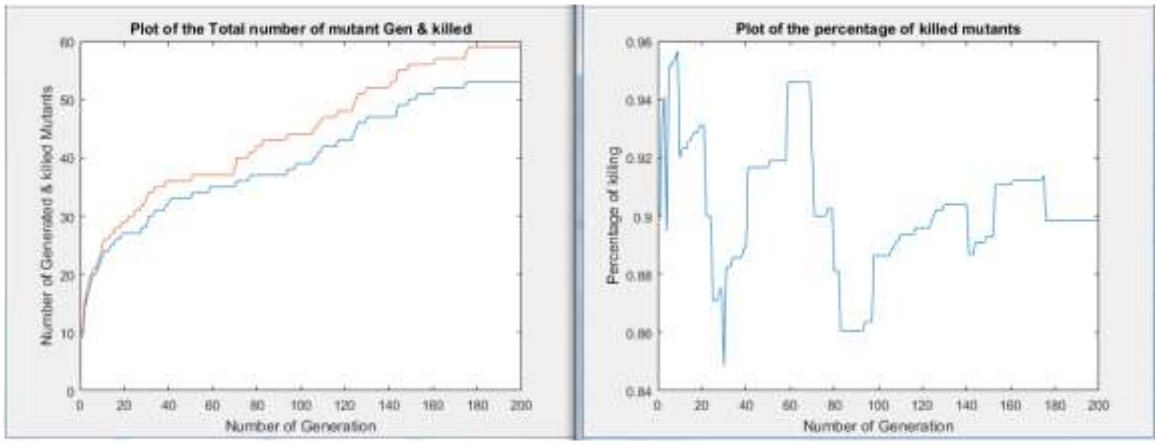


Figure 15: Killed mutants and unique mutants for QuadraticSolver (200 Generations)

The left graph of Figure 15 shows the cumulative mutants generated and the one killed. Initially, the majority of the mutants were killed. The graph on the right of the figure depicts the proportion of killed mutants. It shows that the proportion of mutants being killed fluctuates along the different generations until it reaches 180 – when it seems to be stable.

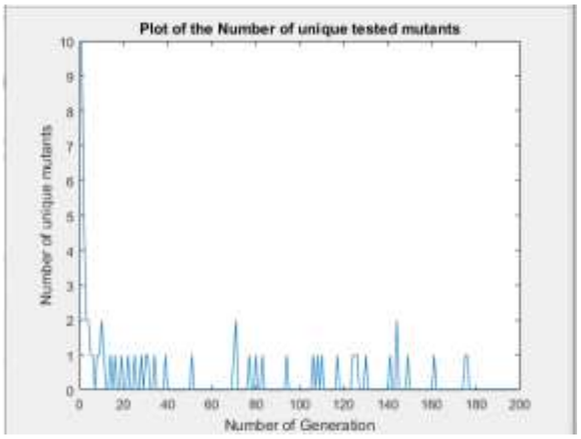


Figure 16: Number of unique tested mutants for QuadraticSolver (200 Generations)

The graph plot in Figure 16 shows the number of unique mutants tested in the experiment during each generation when the number of generation is increased to 200. At almost 178th generation, no new mutants were tested. This means the mutants generated are already generated in the previous generations.

Finally, the number of generation was further increased to 250 so as to be more confident about the results. The experiment was run and the results are shown in Figure 17.

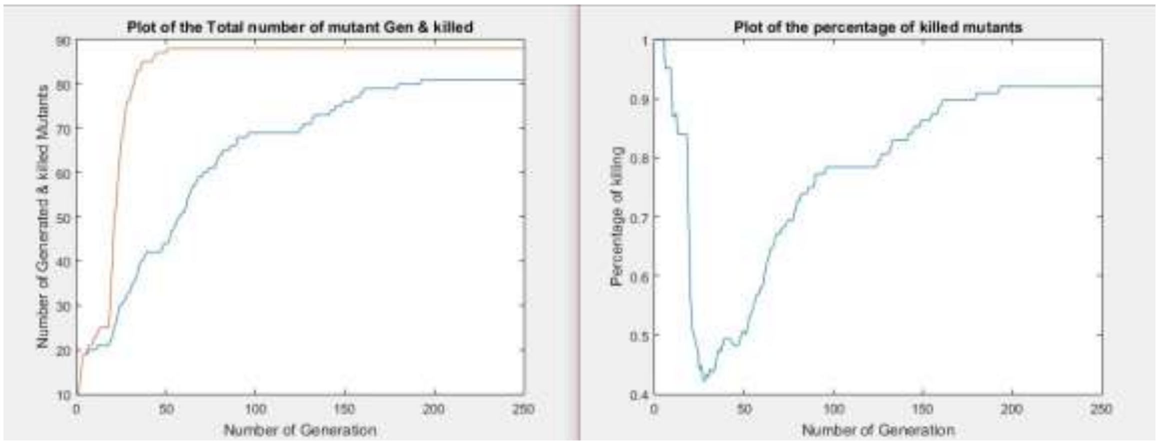


Figure 17: Killed mutants and unique mutants for QuadraticSolver (250 Generations tGA-mGA)

Having run the experiment for 250 generations, the results show that no more mutants are killed as the cumulative killed mutants remains unimproved for about 50 more generations.

Out of 88 mutants generated, only 80 were killed. This resulted in killing 90.9% of the mutants generated. The overall performance of the test cases was recorded to be **35.564**.

The same experiment was repeated but with randomly generated mutants with the test cases generated by the GA. The results are shown in Figure 18 .

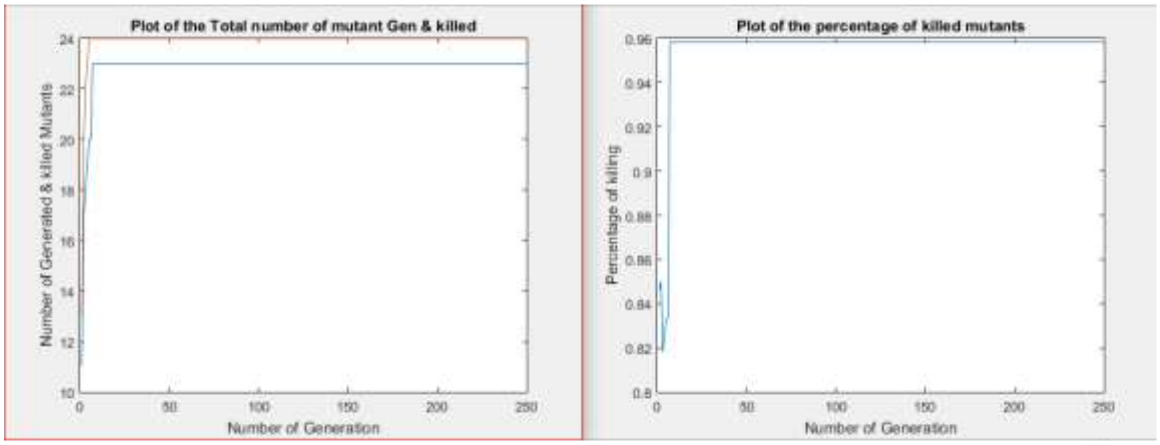


Figure 18: Killed mutants and unique mutants with randomly generated mutants for QuadraticSolver (250 Generations)

The results in the graphs show that less mutants (24 out of 87) were generated because they were not guided by any heuristic rather than random generation. In other words, 27.6% of the total mutants generated by GA was generated by random generator. What is clear is that most of the already generated mutants were repeatedly generated and the total unique mutants generated are only 24, (23 were killed) which is too small compare to the number when GA is used to generate them as shown in Figure 17. In that case, 96% of the randomly generated mutants were killed in less than 10 generations. Figure 19 shows the number of unique mutants generated in each generation. The random generator could not generate any new mutants before reaching the 10th generation.

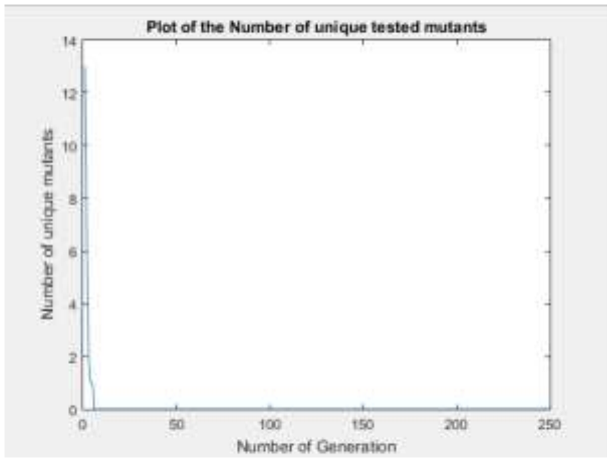


Figure 19: Number of unique tested mutants (randomly generated) for QuadraticSolver (250 Generations)

Conversely, randomly created test cases were executed against mutants generated by GA. This result is shown in Figure 20 .

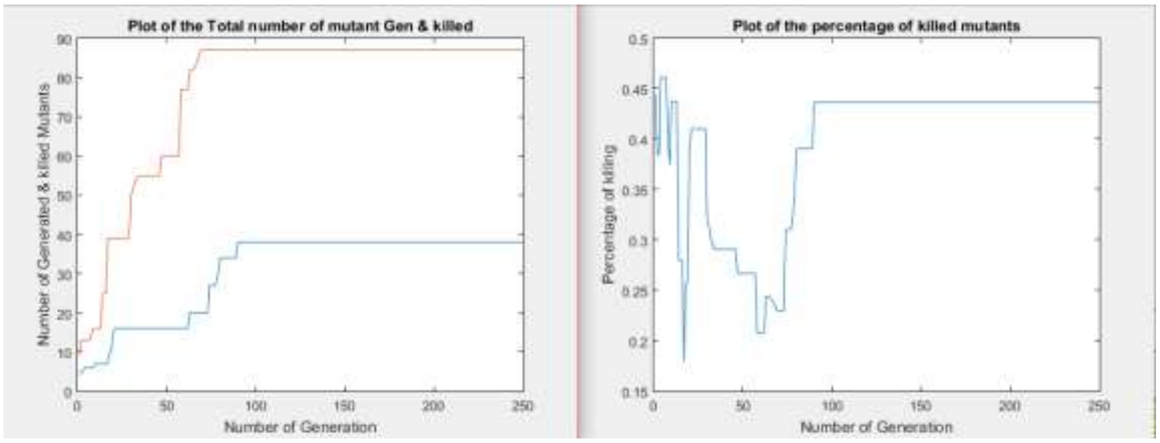


Figure 20: Killed mutants and unique mutants with randomly generated tests for QuadraticSolver (250 Generations)

Figure 20 shows the number and percentage (on average) of killed mutants and number of generated mutants. The mGA was able to generate 87 unique mutants until the 74th generation. Similarly, the random test was able to kill 38 unique mutants cumulatively at 92nd generation and no more mutants were killed. This shows that only 43.7% of the generated distinct mutants were killed by the randomly generated test cases.

In order to compare the results of applying GAs to generate both mutants and test cases, an experiment was conducted by executing randomly generated mutants with randomly generated test cases. The results are shown in Figure 21 :

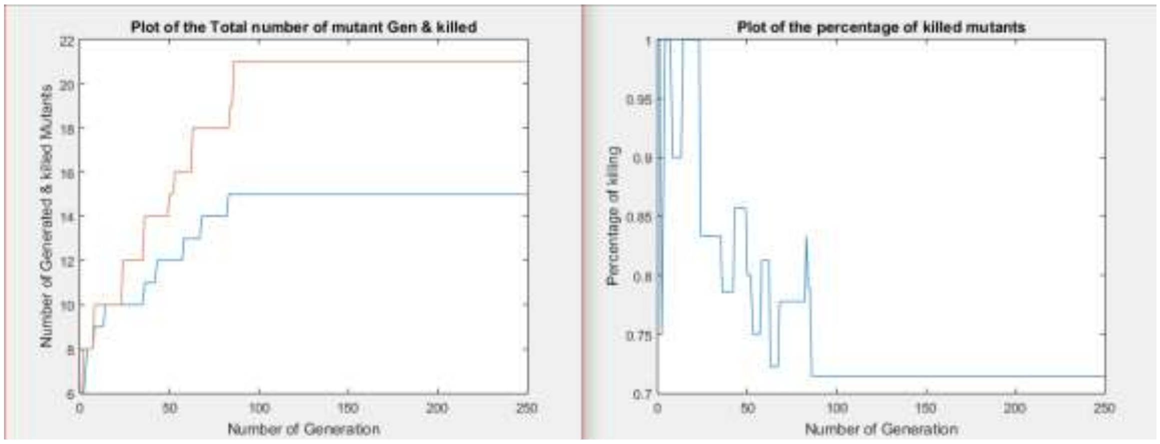


Figure 21: Killed mutants and unique mutants with random mutants and tests for QuadraticSolver (250 Generations)

The plots in the figure above displays the number of mutants generated and killed as 21 and 15 respectively resulting into killing 71.4% of the mutants generated arbitrarily.

5.5.2. TriangleType

This program has 14 LOC with three branches. It accepts three inputs which correspond to the three sides of a triangle. The output of the program is the type of triangle represented by the three sides as inputs. The GAs run for 250 generations and the result is shown in Figure 22 .

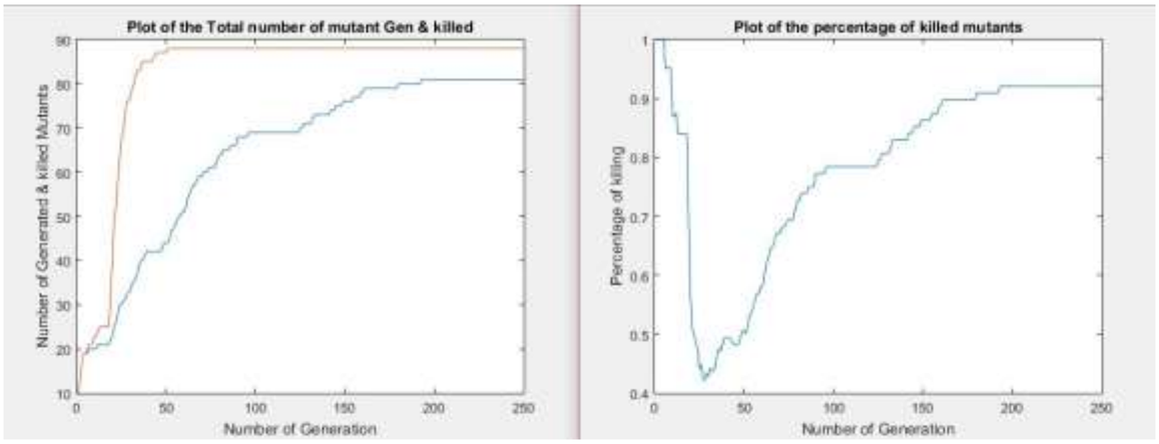


Figure 22: Killed mutants and unique mutants for TriangleType (250 Generations tGA-mGA)

The average number of unique mutants generated increases as the GA executes through the generations up to the 50th generation when no more mutants were generated by the mGA. The tGA generated and complemented test cases in every generation ensuring that maximum number of mutants were killed. At generation 195, the test suite has succeeded in killing 79 mutants out of the 88 generated mutants. This means 89.8% of the mutants were killed. The plot at the right of Figure 22 shows the proportion of mutants being killed and how it fluctuates from one generation to next generation. The overall fitness of the test cases was **34.95**.

The same experiment was repeated but with randomly generated mutants but with the test cases generated by the GA. The results are displayed in Figure 23 .

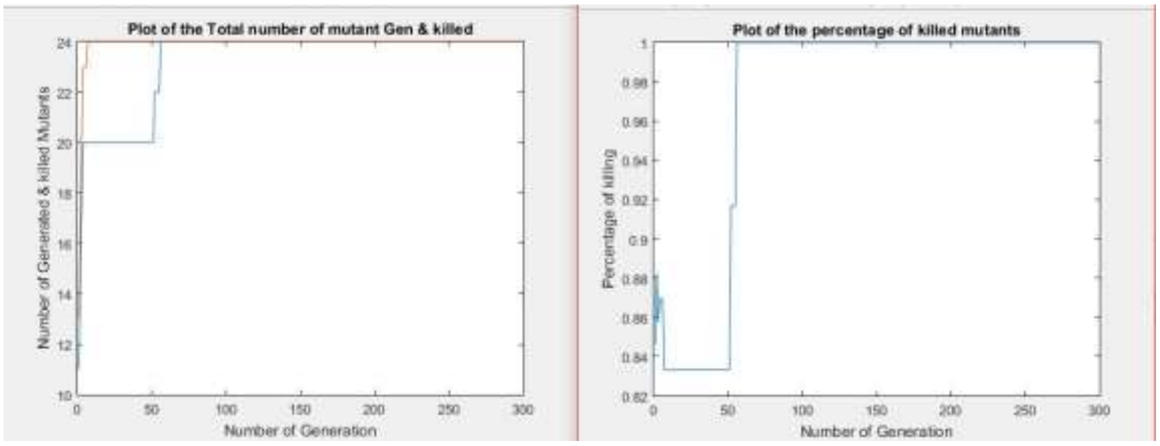


Figure 23: Number of unique tested mutants (randomly generated) for Triangle (300 Generations)

The results shown in the graphs in Figure 23 shows that only 24 mutants were generated on average and all the mutants were killed by the GA-guided test cases. The results also show that all the mutants were killed after 50 generations. This shows that the test cases generated by the GA are effective. And the reason for quick convergence of the plot is the fact that the mutants were randomly generated while the test cases were guided by GA. In other words, the mutants generated were easier to kill than their counterparts generated by mGA.

The same experiment was repeated but with randomly generated test cases against the optimized mutants generated by mGA. The results are shown in Figure 24 .

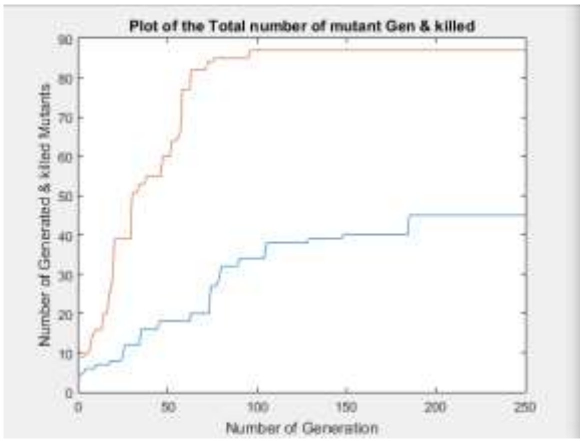


Figure 24: Number of unique tested mutants with randomly generated tests for Triangle (250 Generations)

In this case, the mutation generation is guided by mGA and as a result, a high number of unique mutants (87 to be precise) were created and only 44 were killed out of 97 resulting in killing 50.6% of the generated mutants. This is because the test cases are generated randomly, rendering the killing of the mutants not as effective as killing the mutants with test cases generated by tGA.

To show that GA is doing a great job in generating optimized mutants and test cases, an experiment is carried out by considering executing randomly generated mutants against randomly generated test cases. The plots in Figure 25 show the results of the experiment.

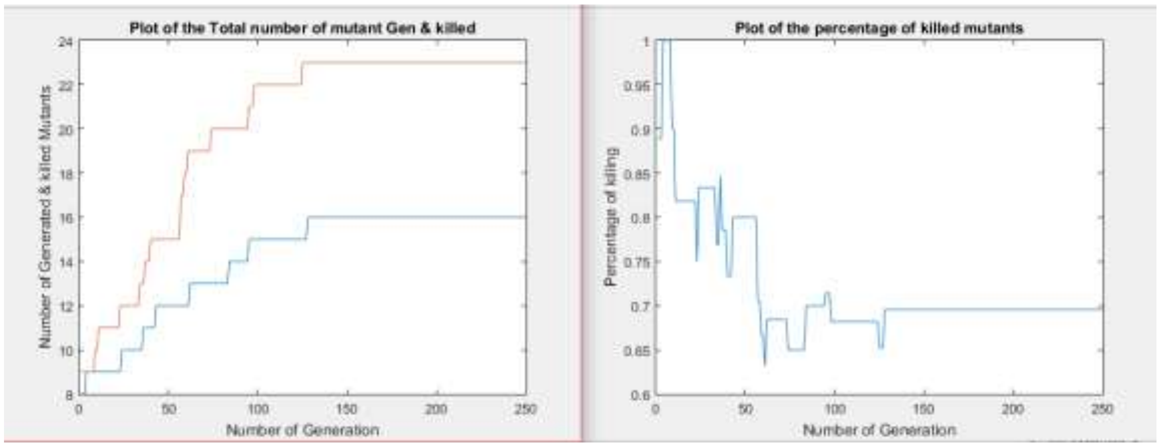


Figure 25: Killed mutants and unique mutants with random mutants and tests for TriangleType (250 Generations)

Out of 23 random mutants generated, only 69.9% were killed which equals to 16 killed mutants. As the generation of the execution reaches 135, no more mutants were killed nor generated.

5.5.3. MID

This program has 20 LOC and finds the middle number from list of three numbers as inputs. The GAs run for 250 generations and the following are the results obtained.

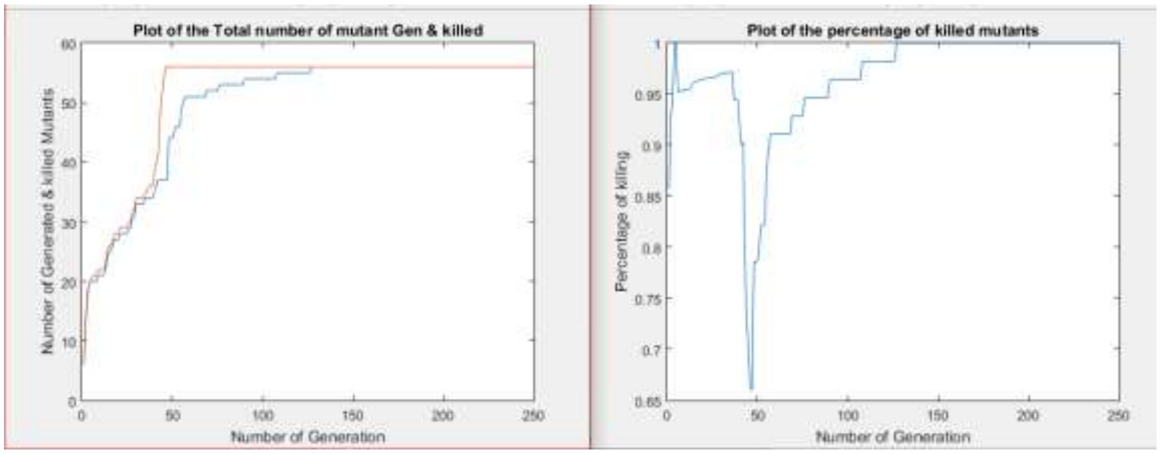


Figure 26: Killed mutants and unique mutants for MID (250 Generations tGA-mGA)

The graphs in Figure 26 explains the total average number of generated mutants and the number of those killed by the test generated by tGA. The total number of distinct mutants generated was 57 and all were killed. This means 100% of the generated mutants were killed by the test suite and the test suite is effective. The overall fitness value of the test suite is **40**.

A set of random mutants of MID program was also generated and executed on the test cases generated by the tGA. The results are shown below:

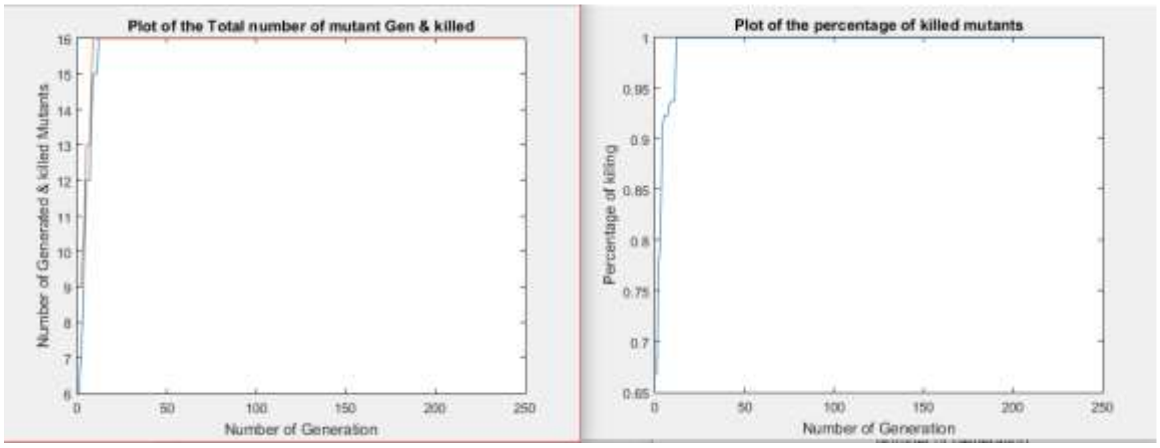


Figure 27: Number of unique tested mutants (randomly generated) for MID (250 Generations)

Figure 27 shows how all the randomly generated mutants were killed by test suites in less than 20 generations due to the fact that the mutants were just randomly generated, which leads to generating easy to kill mutants. This makes the optimized test cases kill the mutants in such a short time (generation).

Conversely, mGA was allowed to generate optimized mutants and executed against randomly generated test cases. The results are shown in Figure 28 :

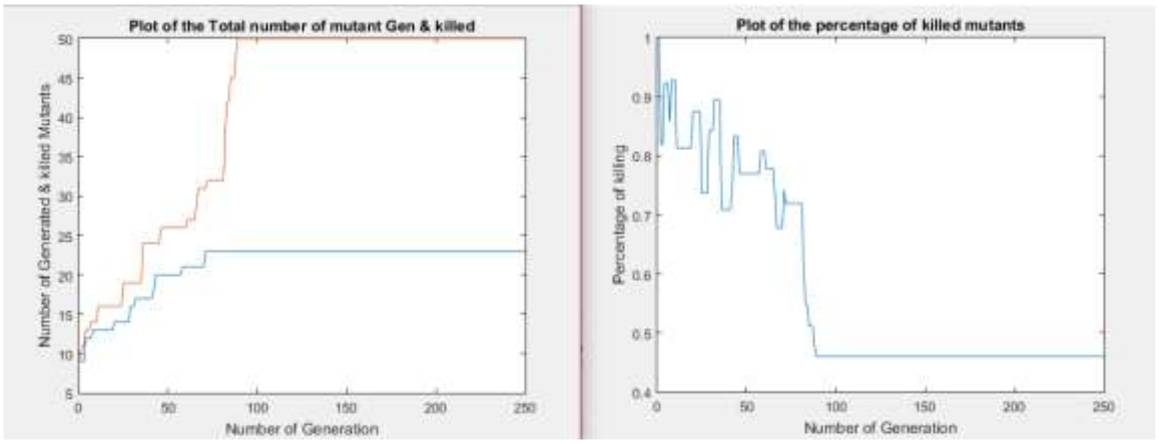


Figure 28: Killed mutants and unique mutants with randomly generated tests for MID (250 Generations)

The figure above shows that only 50 mutants were generated and 23 were killed by the randomly created test cases. This makes the proportion of the killed mutants to be 46% of the total generated mutants guided by mGA.

In order to validate and justify the effectiveness of using GA to generate mutants and test cases, another experiment was carried out by executing mutants that were generated randomly while taking randomly generated test cases as inputs. The results of the execution are shown in below:

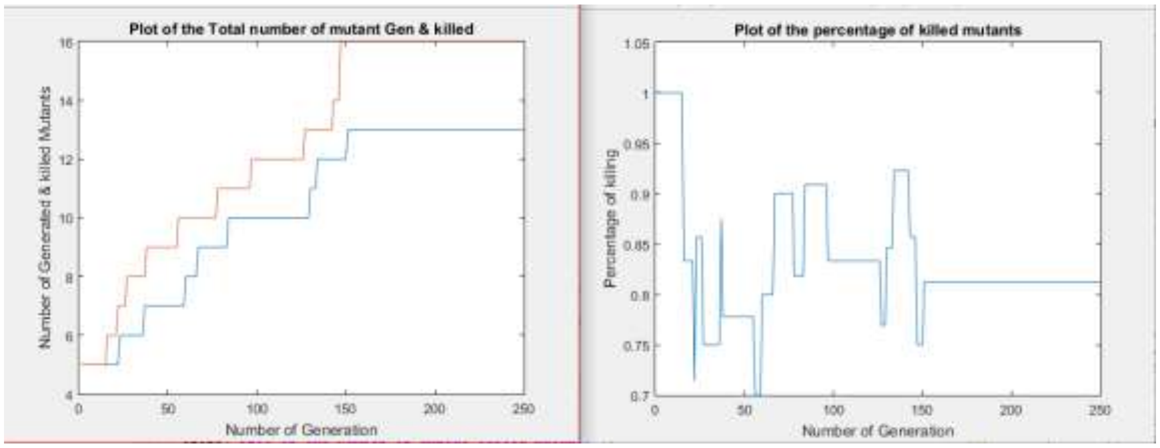


Figure 29: Killed mutants and unique mutants with random mutants and tests for MID (250 Generations)

The plots in the left graph of Figure 29 shows the total number of mutants generated and killed as 16 and 13 respectively causing the percentage of the killed mutants to be 81.3%.

5.5.4. LineRectangleClassifier

This program takes eight inputs (four inputs corresponding to the coordinates of a line and four inputs representing the coordinates of a rectangle). It determines the location of the line with respect to the rectangle. The line can be completely inside the rectangle or completely outside the rectangle. It can also be partly inside and partly outside the rectangle. Those that are found completely outside the rectangle can be at the top, bottom, left or right side of the rectangle. The GAs ran for 250 generations and the following are the results obtained.

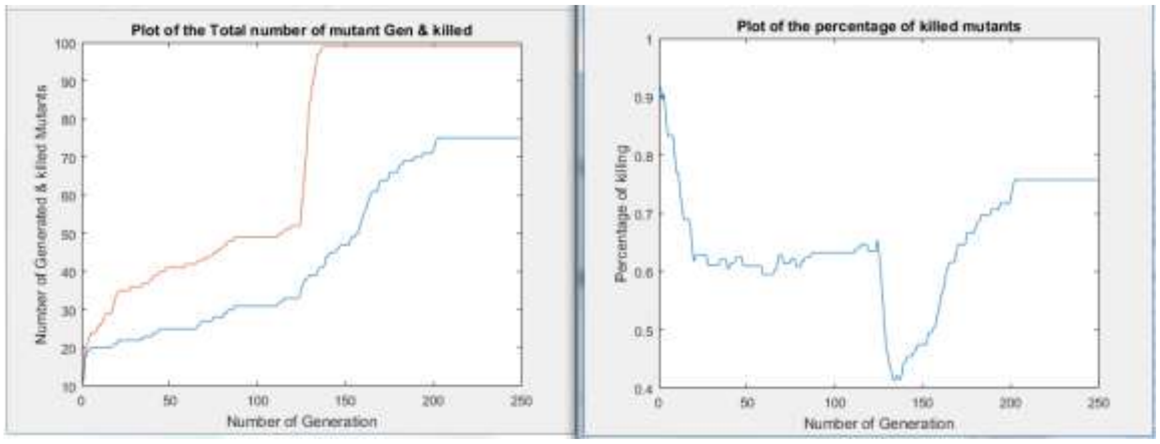


Figure 30: Killed mutants and unique mutants for LineRectangleClassifier (250 Generations tGA-mGA)

On average, a total number of 99 mutants were produced and only 74 were killed. This results in killing 74.74% of the total mutants generated. The plot on the right side of Figure 30 shows the percentage of the killed mutants in each generation. The overall fitness of the test suite was **33.98**.

On the other hand, set of random mutants were generated and test cases generated by GA were executed against the mutants. The results of the execution are shown below.

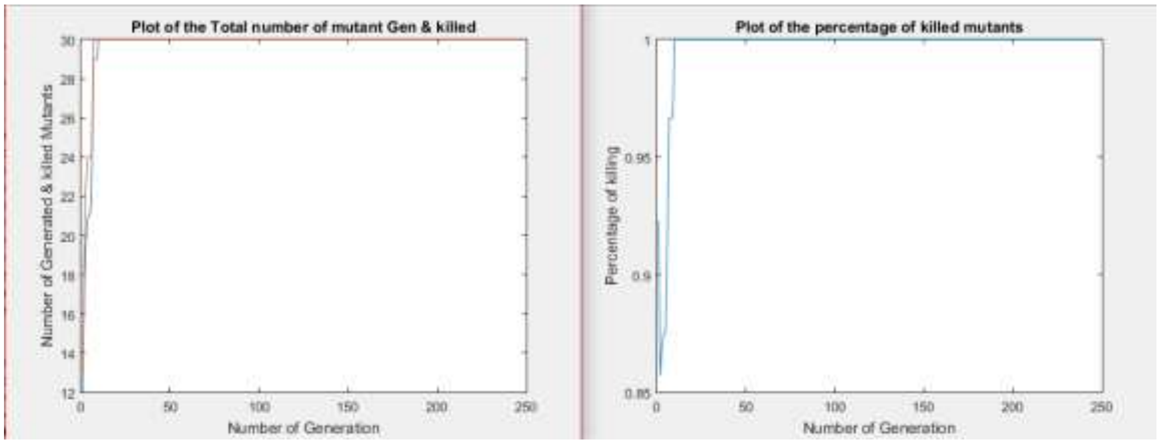


Figure 31: Number of unique tested mutants (randomly generated) for LineRectangleClassifier (250 Generations)

A total number of 30 mutants were randomly generated and 100% of the mutants were killed. This is because test cases were generated by tGA and as a result the effective test cases killed the entire mutants.

Randomly generated test cases were executed against mutants generated by mGA. The result of the execution is shown in Figure 32 :

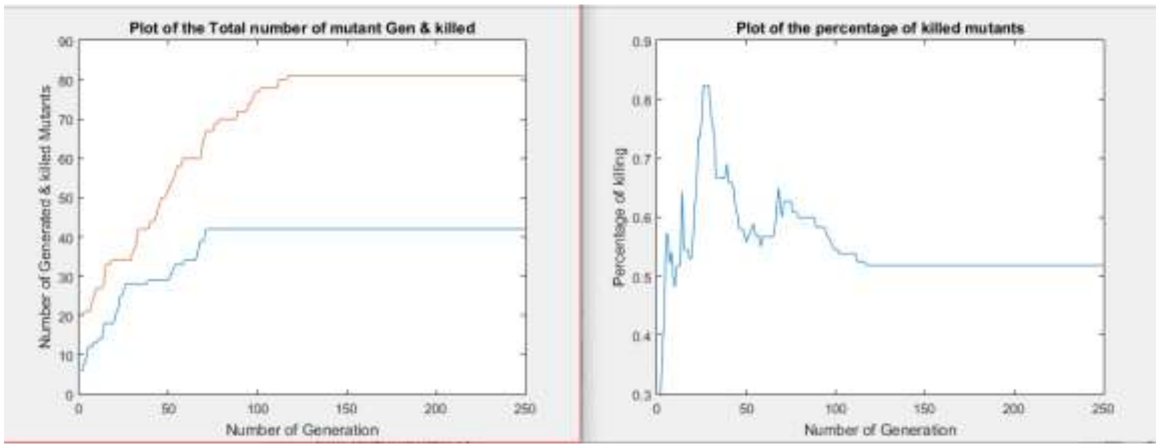


Figure 32: Killed mutants and unique mutants with randomly generated tests for LineRectangleClassifier (250 Generations)

The results shown in Figure 32 depict the total number of mutants of LineRectangleClassifier program generated by mGA and the number of mutants killed. It can be seen that 81 mutants were generated in total while only 51.9% equivalent to 42 mutants were successfully killed by the randomly generated test cases.

Another experiment was carried out by executing randomly generated mutants against randomly generated test cases.

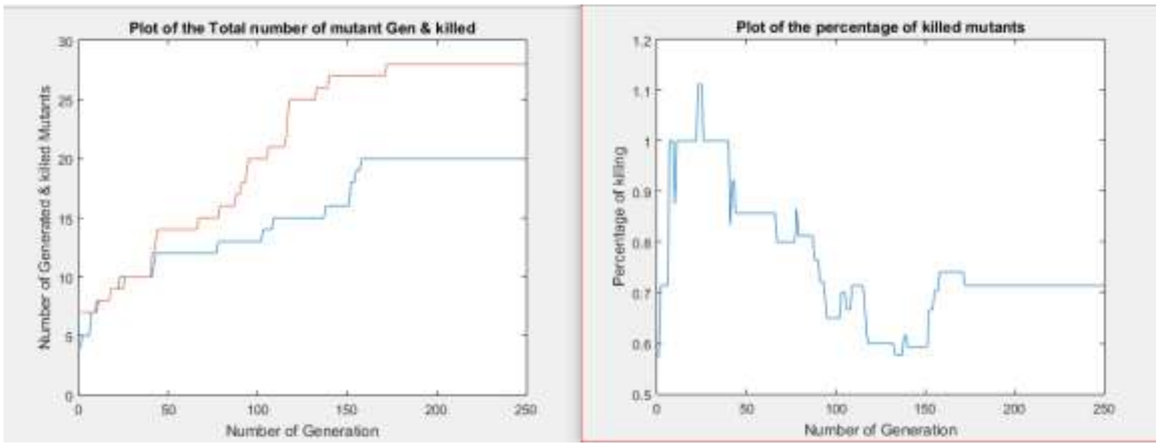


Figure 33: Killed mutants and unique mutants with random mutants and tests for LineRectangleClassifier (250 Generations)

The graphs show that only 20 mutants were killed out of 28 randomly generated mutants. And the percentage of the killed mutants in each generation is shown to be 71.4%.

5.5.5. PointCircleClassifier

The program PointCircleClassifier takes the coordinates of a circle, its radius and a coordinates of a point as inputs and detect if the point is inside the circle, outside the circle or on the circumference of the circle. It has 12 LOC. The GAs run for 250 generations and the following are the results obtained.

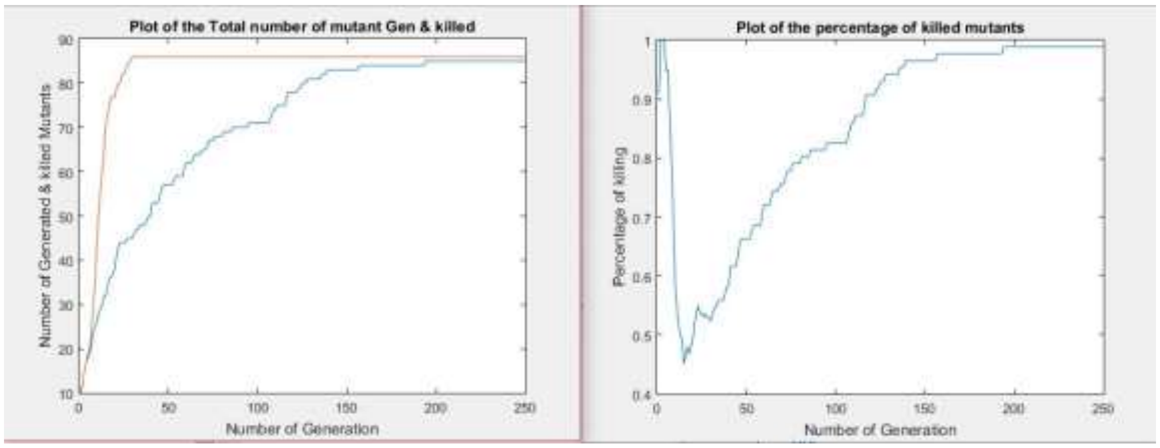


Figure 34: Killed mutants and unique mutants for PointCircleClassifier (250 Generations tGA-mGA)

The plot on the left side of Figure 34 shows the total average number of mutants generated by mGA and the number of killed ones. It shows that 85 out of 86 mutants were killed. The overall fitness evaluation of the test suite was **35.73**. This means 98.8% of the total mutants were killed by the optimized test cases. The plot on the right is the graph showing the percentage of killed mutants in each generation. Another variant of the experiment was conducted by generating mutants randomly and executing them against the optimized test cases. The result of the execution is shown in Figure *35* .

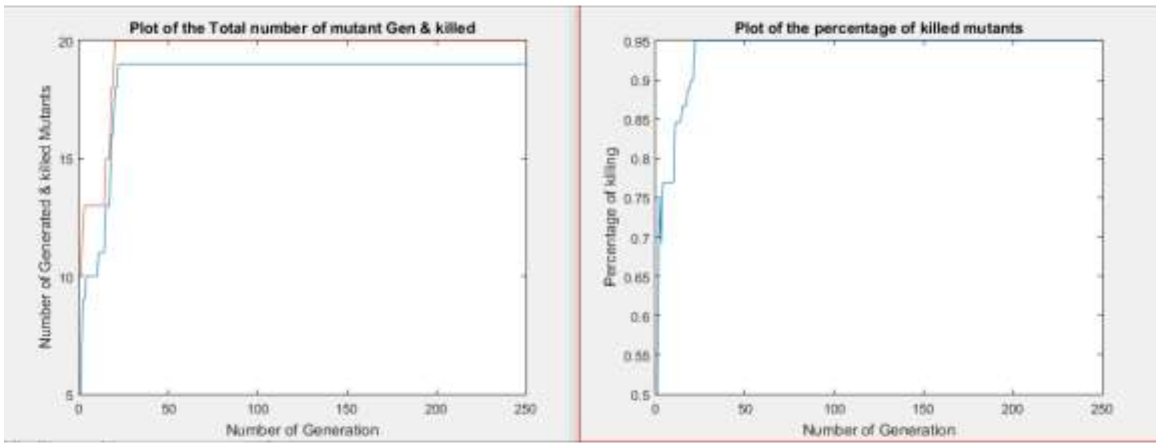


Figure 35: Number of unique tested mutants (randomly generated) for PointCircleClassifier (250 Generations)

The result shows that only 20 unique mutants were generated. This is because the mutants were generated randomly. In other words, the generation of mutants is not guided by any heuristic but only random generation. The result also depicts that 19 out of 20 mutants were killed, which is equivalent to 95% of the mutants being killed.

On the other hand, optimized mutants (i.e. difficult-to-kill mutants) are executed against randomly generated test cases. The result of the execution is shown in Figure 36 .

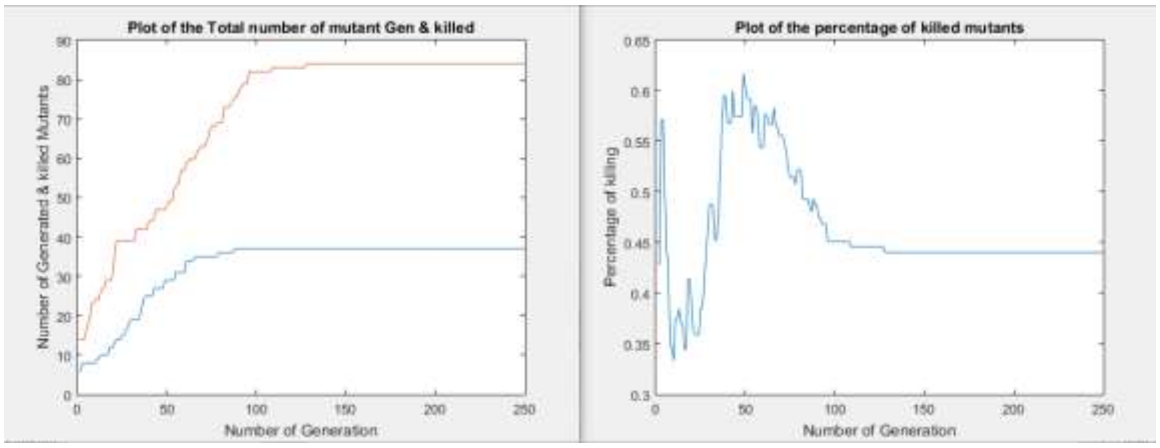


Figure 36: Killed mutants and unique mutants with randomly generated tests for PointCircleClassifier (250 Generations)

Similarly, the total number of mutants generated by mGA for PointCircleClassifier is 84 and only 44% of the mutants were killed corresponding to 37 mutants. The percentage of killed mutants is low because the test cases were just randomly generated while the mutants' generation is guided by mGA. This is why the gap between the total mutants and killed mutants is wide.

The result was also investigated by generating random mutants and random test cases. These test cases were evaluated by executing the mutants against the test cases. The results are shown below:

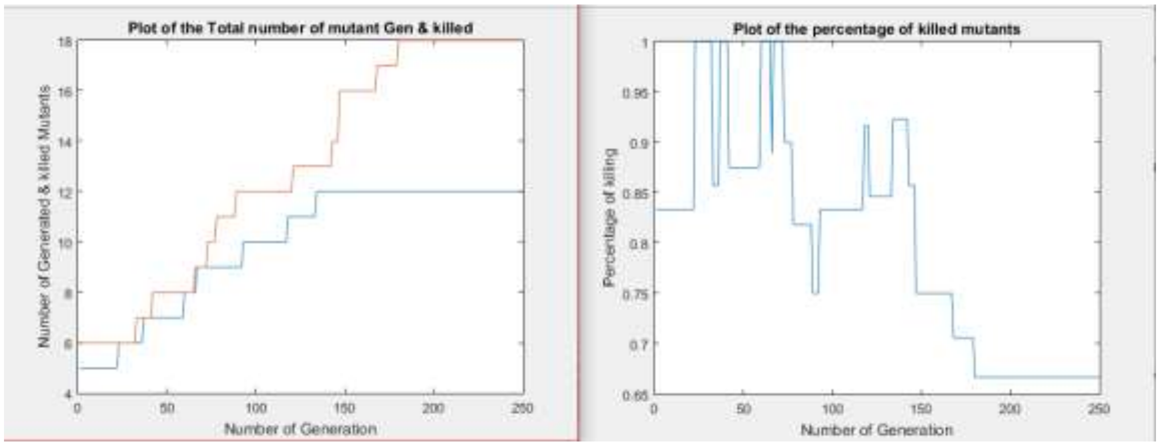


Figure 37: Killed mutants and unique mutants with random mutants and tests for PointCircleClassifier (250 Generations)

The result shows that 12 out of 18 generated mutants were killed in less than 150 generations, making the proportion of killed mutant to be 66.7%.

5.6 Confidence Interval

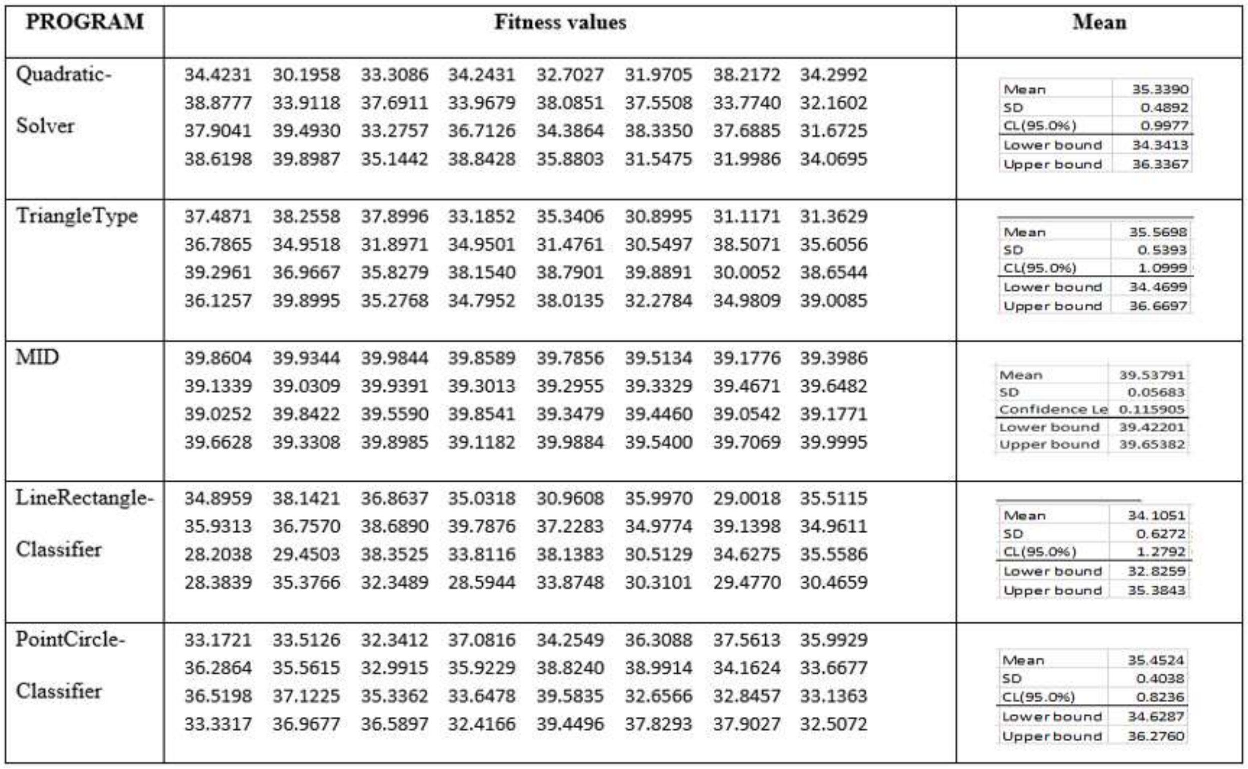
In order to find the confidence interval of data whose population standard deviation is known using the standard deviation and sample mean, the data has to be from a normal distribution. If there is no certainty with regards to the data being from a normal distribution, the number of data has to be large enough (at least 30) in order to apply the Central Limit Theorem which allows the usage of Z-values in the formula. In lieu of this, the experiment was repeated for each subject program for 32 times. Experiments are often repeated in order to give the following insights [66]:

- ✓ A large amount of results may make it easier to spot anomalies.
- ✓ Repetition reduces the likelihood of errors or anomalous results.
- ✓ Scientist repeat others' experiments to verify the accuracy of the findings.
- ✓ Repeating an experiment allows a person to refine the results or simplify the methodology.

- ✓ Experiments are often repeated in order to study why they brings about the results they do.

The results of the repetition are shown in Table 20 :

Table 20: Fitness of tGA of each subject program over 32 runs



PROGRAM	Fitness values								Mean	
Quadratic-Solver	34.4231	30.1958	33.3086	34.2431	32.7027	31.9705	38.2172	34.2992	Mean	35.3390
	38.8777	33.9118	37.6911	33.9679	38.0851	37.5508	33.7740	32.1602	SD	0.4892
	37.9041	39.4930	33.2757	36.7126	34.3864	38.3350	37.6885	31.6725	CL(95.0%)	0.9977
	38.6198	39.8987	35.1442	38.8428	35.8803	31.5475	31.9986	34.0695	Lower bound	34.3413
									Upper bound	36.3367
TriangleType	37.4871	38.2558	37.8996	33.1852	35.3406	30.8995	31.1171	31.3629	Mean	35.5698
	36.7865	34.9518	31.8971	34.9501	31.4761	30.5497	38.5071	35.6056	SD	0.5393
	39.2961	36.9667	35.8279	38.1540	38.7901	39.8891	30.0052	38.6544	CL(95.0%)	1.0999
	36.1257	39.8995	35.2768	34.7952	38.0135	32.2784	34.9809	39.0085	Lower bound	34.4699
									Upper bound	36.6697
MID	39.8604	39.9344	39.9844	39.8589	39.7856	39.5134	39.1776	39.3986	Mean	39.53791
	39.1339	39.0309	39.9391	39.3013	39.2955	39.3329	39.4671	39.6482	SD	0.05683
	39.0252	39.8422	39.5590	39.8541	39.3479	39.4460	39.0542	39.1771	Confidence Le	0.115905
	39.6628	39.3308	39.8985	39.1182	39.9884	39.5400	39.7069	39.9995	Lower bound	39.42201
									Upper bound	39.65382
LineRectangle-Classifier	34.8959	38.1421	36.8637	35.0318	30.9608	35.9970	29.0018	35.5115	Mean	34.1051
	35.9313	36.7570	38.6890	39.7876	37.2283	34.9774	39.1398	34.9611	SD	0.6272
	28.2038	29.4503	38.3525	33.8116	38.1383	30.5129	34.6275	35.5586	CL(95.0%)	1.2792
	28.3839	35.3766	32.3489	28.5944	33.8748	30.3101	29.4770	30.4659	Lower bound	32.8259
									Upper bound	35.3843
PointCircle-Classifier	33.1721	33.5126	32.3412	37.0816	34.2549	36.3088	37.5613	35.9929	Mean	35.4524
	36.2864	35.5615	32.9915	35.9229	38.8240	38.9914	34.1624	33.6677	SD	0.4038
	36.5198	37.1225	35.3362	33.6478	39.5835	32.6566	32.8457	33.1363	CL(95.0%)	0.8236
	33.3317	36.9677	36.5897	32.4166	39.4496	37.8293	37.9027	32.5072	Lower bound	34.6287
									Upper bound	36.2760

The values shown in Table 20 are the values of the fitness of tGA for each subject program executed 32 times. Each of the values of the subject programs are analyzed and the descriptive statistical values are obtained. The values are plotted to show the mean with 95% confidence interval as shown in Figure 38.

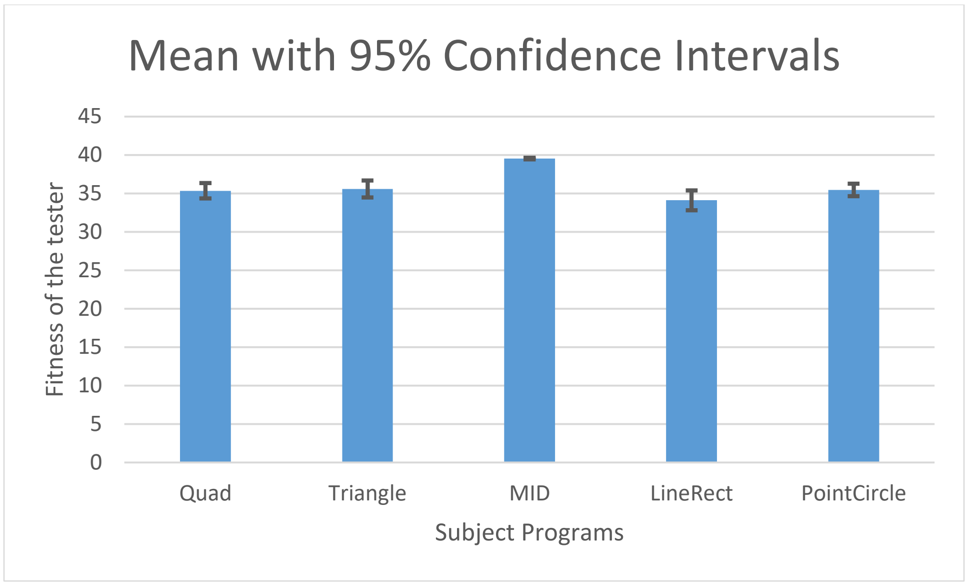


Figure 38: Confidence Intervals for 32 runs of the experiment on the subject programs

5.7 Answering Research Questions

There are four research questions as stated in CHAPTER 3. This section would answer the research questions.

RQ 1: What is the effectiveness of the generated test cases in killing the generated mutants?

GAs were used to generate both test cases and mutants. The mutants were made to execute against the test cases to measure the effectiveness of the test cases. The experiment was carried out on five subject programs. Figure 17, Figure 22, Figure 26, Figure 30, and Figure 34 respectively show the result of executing optimized test cases against optimized mutants. The results on the figures show 90.9%, 89.8%, 100%, 74.7% and 98.8% of mutants were respectively killed by the optimized test cases. These values show the effectiveness of the generated test cases.

RQ 2: How strong are the mutants generated?

Figure 17, Figure 22, Figure 26, Figure 30, and Figure 34 show that the number of mutants killed were increasing gradually, showing that most of the mutants were resisting killing by the test cases. Some of the mutants were only killed when the test cases were more optimized in the later generations. The mutation scores were only obtained when the mutants were attempted to be killed for about 200 generations. If the mutants were killed just in 10 – 50 generations, we would have concluded that the mutants are not strong.

RQ 3: Is the game-like approach better than random generation of both test data and mutants?

Experiments were carried out to investigate if the game-like approach presented in this study performs better than random generation. To verify this, another set of experiments were carried out generating random test cases but generating mutants using mGA. Also, randomly generated mutants were executed against optimized test cases generated by tGA. Figure 18 and Figure 20 show the results of the experiments for QuadraticSolver. The former shows that only few percentage of the possible mutants were generated because the mutants were generated randomly and almost all the mutants were killed because the test cases are optimized. While the latter shows result of randomly generated test cases against optimized mutants, the number of mutants generated is maximized but only very small percentage of the mutants were killed. This is because the test suite to kill the mutants is randomly generated. Similar results are shown for other subject program. The Table 21 shows the subject programs and the figures showing their results.

Table 21: Results of RQ3

Subject Programs	**Results shown in**
QuadraticSolver	Figure 18 and Figure 20
TriangleType	Figure 23 and Figure 24
MID	Figure 27 and Figure 28
LineRectangleClassifier	Figure 31 and Figure 32
PointCircleClassifier	Figure 35 and Figure 36

RQ 4: Both GAs were executed across different set of parameters and the set of parameters with the highest performance (i.e. yielding the highest score for each GA) is selected. The set of GA parameters used in the experiment for tester GA and mutant GA are shown in Table 17 and Table 19 respectively.

5.8 Hard to Kill Mutants

The hard-to-kill mutants generated from the experiment when both test cases and mutants respectively generated by tGA and mGA are executed in isolation without mixing with any other killed mutants to see if the test cases can kill them. The mutants were executed against the optimized test cases expecting some of the mutants to be killed. This was applied for each of the subject programs. The results obtained show that none of them were killed. We investigated why they were not killed using manual method by cross-checking the code of the hard-to-kill mutants. What we observed was that the mutated statement is not reachable. The reason why these statements were not reached is that the statements are part of the body of a conditional statement of which this conditional statement is testing equality of some combination of the program input with other different combinations. This is illustrated as follows:

For example, in QuadraticSolver, line 7 (‘***elseif (d==0)***’) is the condition to get lines 8 and 9 executed. If the mutation is applied on the program statement in line 8 or line 9, the program code would be unreachable unless the condition is satisfied. Again, for d (which is the determinant ***‘b^2-4ac’***) to be zero, the likelihood is very small. This problem can properly be addressed using white box testing or changing the fitness in such a way that the objective would be to get values that would be equal to zero or close. So that they can be guided to become zero in the long run of the execution. This would have changed our aim of presenting a black-box approach to kill as many mutants as possible. In fact, if a program does not have such condition, then the approach would be inappropriate for such program. The same challenge was recorded for the remaining subject programs except for the MID program, which has no such condition. This is a strong reason why test cases generated to kill the mutants of MID program were able to get 100% mutation score. In other word, there is a record of killing the entire mutants of MID program.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This chapter discusses the summary of the study, limitations of the study and some threats to validity of the results obtained.

6.1 Main Contributions of the Study

The following are the main contributions of this study:

1. We compared the existing GA-based test data generation techniques using a framework of features we developed;
2. We proposed a GA-based test data generation technique using mutation analysis;
3. We presented mutant generation using GA considering arithmetic and relational operator replacement as the mutation operators;
4. We developed mutant converter, which takes mutant chromosomes and converts them to real mutant programs;
5. We presented the mutants and test case generation in a form of a non-cooperative game;
6. We validated the approach using different subject programs and the results show that the approach is effective.

6.2 Limitations of the Study

This study suffers from the following limitations:

1. Our study did not include any technique to detect equivalent mutants. We did not check for any semantic similarity between the original programs and the mutants generated. In that regard, we are not sure if there are equivalent mutants in the generated mutants. The accuracy of our result would be affected by the presence of equivalent mutant (if any). We carried out the experiment under the assumption that there are no equivalent mutants as we have tried to minimize the likelihood of having equivalent mutants.
2. The study is limited to only two classes of mutation operators namely: Arithmetic Operator Replacement and Relational Operator Replacement. Applying the approach using more mutation operators can help in generalizing the results obtained.
3. All the program subjects used in the study are small-sized. This may limit the extent to which we can generalize the results obtained.

6.3 Threats to Validity

Despite the fact the experiments were cautiously designed to ensure fairness, a number of threats are posed to the validity of the results obtained. The threats are as follows: how the mutation operators are selected and the choice of test cases.

The huge number of mutants generated in mutation increases the cost of mutation testing. In order to reduce this cost, we employed selective mutation – where a particular set of mutation operators are selected from the whole set of the operators. The inability to carryout exhaustive application of the mutation operators may pose a threat to the validity of the results. The relational operator was therefore selected alongside arithmetic replacement operators because it alters the control flow in the mutant; thereby increasing the coverage of the program under test. This threat can be reduced in future by adding more operators to the list of operators.

6.4 Future Work

We have implemented a game-like approach to generating mutants and test cases to kill the mutants without being bothered by the number of generated mutants and test cases.

In future work, we would look into how we can apply GA to reduce the number of mutants generated while maintaining the efficiency and accuracy of the analysis using our game-like approach.

The need for a test oracle of program under test makes it mandatory to execute every program, thereby results in slowing down the testing process (i.e. the act of comparing the expected output with the real output under a set of inputs makes the testing a time-consuming process). Therefore, applying Machine Learning Techniques would get rid of the need to know the expected output prior the beginning of the testing activities. Some features of mutants and tests are taken to predict if the mutants represented by those features can be killed by the corresponding test cases without executing the mutants. A deeper research could be conducted to further reduce the cost of mutation testing using predictive mutation testing and metaheuristics whereby some features of mutants and tests are collected to forecast if a mutant would be killed or not without going through the stress of executing the whole mutants generated. Apart from identifying invalid mutants in this research, we will distinguish any redundant mutant from others in our future research.

It is considered a promising direction to future research to optimize the effectiveness of the fitness function by testing varieties of fitness functions. The afore-mentioned recommendations could be complemented by minimizing the number of test cases.

Another direction for future research is generating mutants for covering equality relational operators.

Applying other metaheuristic techniques like Ant Colony Optimization, Particle Swarm Optimization, and Artificial Bee Colony on this technique is a recommended future research.

Implementing the approach in other language than MATLAB as well as exploring more mutation operators can be considered as another direction to future work.

REFERENCES

[1] P. Offutt and A. Jeff, *Introduction to Software Testing*. Cambridge University Press, 2014.

[2] B. . Haskins, B. . Dick, J. . Stecklein, R. . Lovell, G. . Moroney, and J. Dabney, “Error Cost Escalation Through the Project Life Cycle,” *Incose -Annual Conf. Symp. Proceedings- Cd Rom Ed. 2004*, p. 8.4.2, 2004.

[3] W. Eric and M. Aditya, “Fault Detection Effectiveness of Mutation and Data Flow Testing,” *Softw. Qual. J.*, vol. 4, pp. 69–83, 1995.

[4] R. Baker and I. Habli, “An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 787–805, 2013.

[5] M. V Zelkowitz, “Software Testing Lecture Note MSWE 607,” 2000.

[6] IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, vol. 121990, no. 1. 1990.

[7] IEEE, *Standard for Software and System Test Documentation*, vol. 2008, no. July. 2008.

[8] A. P. Mathur, *Foundations of Software Testing 2E*. 2008.

[9] Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing,” *Softw. Eng. IEEE Trans.*, vol. 37, no. 5, pp. 649–678, 2011.

[10] J. Offutt and R. H. Untch, “Mutation 2000: Uniting the Orthogonal,” *Proc. Mutat. 2000 Mutat. Test. Twent. Twenty First Centuries*, pp. 45–55, 2000.

[11] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[12] S. Ecott, “Fault-based Testing of Web Applications,” pp. 1–3, 2008.

[13] M. Woodward, “Mutation Testing-an Evolving Technique,” *Softw. Test. Crit. Syst. IEE …*, pp. 1–6, 1990.

[14] R. H. Untch, “On Reduced Neighborhood Mutation Analysis using a Single Mutagenic Operator,” *ACMSE*, pp. 1–4, 2009.

[15] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, and J. Offutt, “Experimental Evaluation of SDL and One-op Mutation for C,” *Proc. - IEEE 7th Int. Conf. Softw. Testing, Verif. Validation, ICST 2014*, pp. 203–212, 2014.

[16] V. H. S. Durelli, N. M. De Souza, and M. E. Delamaro, “Are Deletion Mutants Easier to Identify Manually?,” *Proc. - 10th IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2017*, pp. 149–158, 2017.

[17] M. E. Delamaro, J. Offutt, and P. Ammann, “Designing Deletion Mutation Operators,” *Proc. - IEEE 7th Int. Conf. Softw. Testing, Verif. Validation, ICST 2014*, pp. 11–20, 2014.

[18] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, “Toward the Determination of Sufficient Mutant Operators for C,” *Softw. Test. Verif. Reliab.*, vol. 11, no. 2, pp. 113–136, 2001.

[19] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, “Sufficient Mutation Operators for Measuring Test Effectiveness,” *ICSE*, pp. 351–360, 2008.

[20] M. Sridharan and A. S. Namin, “Prioritizing Mutation Operators Based on Importance Sampling,” *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, no. Issre 10, pp. 378–387, 2010.

[21] A. S. Namin and J. H. Andrews, “On Sufficiency of Mutants,” *Proc. - Int. Conf. Softw. Eng.*, pp. 73–74, 2007.

[22] M. E. Delamaro, L. Deng, N. Li, V. Durelli, and J. Offutt, “Growing a Reduced Set of Mutation Operators,” *28th Brazilian Symp. Softw. Eng. SBES 2014,* pp. 81–90, 2014.

[23] M. Papadakis and Y. Le Traon, “Effective Fault Localization via Mutation Analysis: A Selective Mutation Approach,” *Proc. 29th Annu. ACM Symp. Appl. Comput.*, pp. 1293–1300, 2014.

[24] A. A. L. de Oliveira, C. G. Camilo-Junior, and A. M. R. Vincenzi, “A Coevolutionary Algorithm to Automatic Test Case S election and Mutant in Mutation Testing,” *2013 IEEE Congr. Evol. Comput.*, pp. 829--836, 2013.

[25] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, “An Experimental Determination of Sufficient Mutant Operators,” *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, 1996.

[26] A. Griffiths, S. Wessler, R. Lewontin, W. Gelbart, D. Suzuki, and J. Miller, “An Introduction to Genetic Analysis,” *Vasa*, p. 706, 2005.

[27] J. McCall, “Genetic Algorithms for Modelling and Optimisation,” *J. Comput. Appl. Math.*, vol. 184, no. 1, pp. 205–222, 2005.

[28] S. Luke, *Essentials of Metaheuristics*, Second. California, USA: Lulu, 2013.

[29] B. and Nadeem, “A Fitness Function for Modular Evolutionary Testing of Object-Oriented Programs,” *Search*, 2005.

[30] P. Pahwa and R. Miglani, “Test Case Design using Black Box Testing Techniques for Data Mart,” *Int. J. Comput. Appl.*, vol. 109, no. 3, pp. 18–22, 2015.

[31] G. Fraser and A. Zeller, “Mutation-Driven Generation of Unit Tests and Oracles,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 278–292, 2012.

[32] T. Mantere, “Automatic Software Testing by Genetic Algorithms,” University of Vaasa, Finland, 2003.

[33] C. W. Hang and Y. Cheung, “Using a GA Adaptor in Multi-Applications,” pp. 839–848, 2003.

[34] R. A. DeMillo and A. J. J. Offutt, “Constraint-Based Automatic Test Data Generation,” *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, 1991.

[35] J.-C. Lin and P.-L. Yeh, “Automatic Test Data Generation for Path Testing Using GAs,” *Inf. Sci. (Ny).*, vol. 131, no. 1, pp. 47–64, 2001.

[36] N. Mansour and M. Salame, “Data Generation for Path Testing,” *Softw. Qual. J.*, vol. 12, no. 2, pp. 121–136, 2004.

[37] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart, “Test Data Generation from UML State Machine Diagrams using GAs,” *Int. Conf. Softw. Eng. Adv. (ICSEA 2007)*, no. Icsea, p. 47, 2007.

[38] C. C. Michael, G. McGraw, and M. A. Schatz, “Generating Software Test Data by

Evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 12, pp. 1085–1110, 2001.

[39] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis, "Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage," *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, pp. 41–48, 2007.

[40] M. A. Ahmed and I. Hermadi, "GA-based Multiple Paths Test Data Generator," *Comput. Oper. Res.*, vol. 35, no. 10, pp. 3107–3124, 2008.

[41] P. R. Srivastava and T. Kim, "Application of Genetic Algorithm in Software Testing," *Intenational J. Softw. Eng. Its Appl.*, vol. 3, no. 4, pp. 87–96, 2009.

[42] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "Evolutionary Mutation Testing," *Inf. Softw. Technol.*, vol. 53, no. 10, pp. 1108–1123, 2011.

[43] McMinn P, "Search-based Software Test Data Generation: A Survey," *Softw. testing, Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[44] S. Ali and L. Briand, "A Systematic Review of the Application and Empirical Investigation of Search-based Test Case Generation," *IEEE Trans. Softw. Eng.*, vol. 36, no. 5, pp. 1–22, 2010.

[45] H. L. T. My, B. N. Thanh, and Tung Khuat Thanh, "Survey on Mutation-based Test Data Generation Survey on Mutation-based Test Data Generation," *Int. J. Electr. Comput. Eng.*, vol. 5, no. 5, pp. 1164–1173.

[46] M. Harman, "Automated Test Data Generation using Search Based Software Engineering," *Second Int. Work. Autom. Softw. Test (AST '07)*, vol. 30, no. 11, pp.

2–2, 2007.

[47] R. A. DeMillo, "Test Adequacy and Program Mutation," *ACM*, pp. 355–356, 1989.

[48] X. Yao, M. Harman, and Y. Jia, "A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence," *Proc. 36th Int. Conf. Softw. Eng.*, pp. 919–930, 2014.

[49] F. C. M. Souza, M. Papadakis, V. Durelli, and M. E. Delamaro, "Test Data Generation Techniques for Mutation Testing: A Systematic Mapping," *Conf. Softw. Eng.*, no. 17, pp. 419–432, 2014.

[50] N. Jatana, B. Suri, and S. Rani, "Systematic Literature Review on Search Based Mutation Testing," *e-Information Softw. Eng. J.*, vol. 11, no. 1, pp. 59–76, 2017.

[51] L. Bottaci, "Instrumenting Programs with Flag Variables for Test Data Search by Genetic Algorithm," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, 2002, pp. 1337–1342.

[52] L. Bottaci, "Predicate Expression Cost Functions to Guide Evolutionary Search for Test Data," in *Proceedings of the 2003 Conference on Genetic and Evolutionary Computation (GECCO '03),* 2003, vol. 2724, pp. 2455–2464.

[53] S. Selvakumar and N. Ramaraj, "A Tool for Generation and Minimization of Test Suite by Mutant Gene Algorithm," *J. Comput. Sci.*, vol. 7, no. 10, pp. 1581–1589, 2011.

[54] C. Sharma, S. Sabharwal, and R. Sibal, "A Survey on Software Testing Techniques using Genetic Algorithm," *Int. J. Comput. Sci. Issues*, vol. 10, no. 1, pp. 381–393,

2013.

[55] R. Malhotra and M. Garg, “An Adequacy Based Test Data Generation Technique Using Genetic Algorithms,” *J. Inf. Process. Syst.*, vol. 7, no. 2, pp. 363–384, 2011.

[56] B. P. Sharma, R. Malhotra, and M. Garg, “Empirical Validation of an Efficient Test Data Generation Algorithm Based on Adequacy based Testing Criteria,” *Softw. Eng. An Int. J.*, vol. 2, no. 1, pp. 20–39, 2012.

[57] P. S. L. de S. Rodolfo Adamshuk Silva,Simone do Rocio Senger de Souza, “A Systematic Review on Search based Mutation Testing,” *J. Inf. Softw. Technol.*, vol. 70, no. 2, pp. 113–117, 2011.

[58] A. J. Offut, Z. Jin, and J. Pan, “The Dynamic Domain Reduction Procedure for Test Data Generation: Design and Algorithms,” Fairfax, VA USA, 1994.

[59] B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. Le Traon, “Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment,” *Proc. 13th Int. Symp. Softw. Reliab. Eng.*, pp. 195–206, 2002.

[60] K. Ayari, S. Bouktif, and G. Antoniol, “Automatic Mutation Test Input Data Generation via Ant Colony,” *Proc. 9th Annu. Conf. Genet. Evol. Comput. (GECCO ’07)*, pp. 1074–1081, 2007.

[61] M. Papadakis and N. Malevris, “An Effective Path Selection Strategy for Mutation Testing,” *Proc. 16th Asia-Pacific Softw. Eng. Conf.*, pp. 422–429, 2009.

[62] L. L. Zhang, T. Xie, L. L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, “Test Generation via Dynamic Symbolic Execution for Mutation Testing,” *Proc. ICSM*,

pp. 1–10, 2010.

[63] M. Papadakis and N. Malevris, “Automatic Mutation Test Case Generation Via Dynamic Symbolic Execution,” 2010.

[64] M. Harman, Y. Jia, and W. B. Langdon, “Strong Higher Order Mutation-Based Test Data Generation,” *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, pp. 212–222, 2011.

[65] L. T. M. Hanh, K. T. Tung, and N. T. Binh, “Mutation-based Test Data Generation for Simulink Models using Genetic Algorithm and Simulated Annealing,” *Int. J. Comput. Inf. Technol.*, vol. 03, no. 04, pp. 763–771, 2014.

[66] Z. Crazy, “The Student Room.” [Online]. Available: https://www.thestudentroom.co.uk/showthread.php?t=1702327.

[67] J. Allen Troy Acree, “On Mutation,” PhD Dissertation, Georgia Institute of Technology, 1980.

[68] H. J.M., “Testing COBOL Programs by Mutation,” PhD Dissertation, Georgia Institute of Technology, 1980.

[69] T. A. Budd, “Mutation Analysis of Program Test Data,” PhD Dissertation, Yale University, New Haven, Connecticut United States, 1980.

[70] A. Tanaka, “Equivalence Testing for FORTRAN Mutation System Using Data Flow Analysis,” PhD Dissertation, Georgia Institute of Technology, 1981.

[71] A. J. Offut, “Automatic Test Data Generation,” PhD Dissertation, Georgia Institute of Technology, 1998.

[72] M. W. Craft, “Detecting Equivalent Mutants using Compiler Optimization Techniques,” Master’s Thesis, Clemson University, 1989.

[73] B. Choi, “Software Testing Using High-Performance Computers,” Doctoral Dissertation, Purdue University, West Lafayette, United States, 1991.

[74] Edward William Krauser, “Compiler-Integrated Software Testing,” PhD Dissertation, Purdue University, 1991.

[75] S. Fichter, “Parallelizing Mutation on a Hypercube,” Master’s Thesis, Clemson University, 1991.

[76] S. Lee, “Weak vs. Strong: An Empirical Comparison of Mutation Variants,” Master’s Thesis, Clemson University, Clemson SC, 1991.

[77] C. N. Zapf, “A Distributed Interpreter for the Mothra Mutation Testing System,” Master’s Thesis, Clemson University, 1993.

[78] M. E. Delamaro, “Proteum - A Mutation Analysis Based Testing Environment,” PhD Dissertation, University of Sao Paulo, 1993.

[79] W. E. Wong, “On Mutation and Data Flow,” PhD Dissertation, Purdue University, 1993.

[80] J. Pan, “Using Constraints to Detect Equivalent Mutants,” Master’s Thesis, George Mason University, United States, 1994.

[81] V. N. Fleyshgakker, “TECHNIQUES TO IMPROVE THE PERFORMANCE OF MUTATION ANALYSIS,” PhD Dissertation, The City University of New York, 1994.

[82] R. H. Untch, “Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method,” PhD Dissertation, Clemson University, 1995.

[83] S. Ghosh, “Testing Component-Based Distributed Applications,” PhD Dissertation, Purdue University, 2000.

[84] W. Ding, “Using Mutation to Generate Tests from Specification,” Master’s Thesis, George Mason University, 2000.

[85] V. Okun, “Specification Mutation for Test Generation and Analysis,” PhD Dissertation, University of Maryland Baltimore, 2004.

[86] Y. S. Ma, “Object-Oriented Mutation Testing for Java,” Doctoral Dissertation, KAIST University, Korea, 2005.

[87] P. May, “Test Data Generation : Two Evolutionary Approaches to Mutation Testing,” PhD Dissertation, The University of Kent, 2007.

[88] J. S. Bradbury, “Using Program Mutation for the Empirical Assessment of Fault Detection Techniques: A Comparison of Concurrency Testing and Model Checking,” PhD Dissertation, Queen’s University Kingston, Ontario, Canada, 2007.

[89] S. Hussain, “Mutation Clustering,” Master’s Thesis, King’s College London, 2008.

[90] K. Adamopoulos, “Search Based Test Selection and Tailored Mutation,” Master’s Thesis, King’s College London, 2009.

[91] D. Hook, “Using Code Mutation to Study Code Faults in Scientific Software,” PhD Dissertation, Queen’s University Kingston, Ontario, Canada, 2009.

[92] G. K. Kaminski, “Applications of Logic Coverage Criteria and Logic Mutation to

Software Testing," PhD Dissertation, George Mason University, 2010.

[93] V. Debroy, "TOWARDS THE AUTOMATION OF PROGRAM DEBUGGING," PhD Dissertation, The University of Texas, Austin, Texas United States, 2011.

[94] C. Zhou, "Mutation Testing for Java Database Applications," PhD Dissertation, Polytechnic Institute of New York University, United States, 2012.

[95] T. Sarkar, "Testing Database Applications using Coverage Analysis and Mutation Analysis," PhD Dissertation, IOWA State University, United States, 2013.

[96] M. A. Hays, "A Fault-Based Model of Fault Localization Techniques," PhD Dissertation, University of Kentucky, United States, 2014.

[97] V. S. Movva, "Automatic Test Suite Generation for Scientific MATLAB Code," Master's Thesis, University of Minnesota, 2015.

[98] X. Li, "The Use of Software Faults in Software Reliability Assessment and Software Mutation Testing," PhD Dissertation, The Ohio State University, United States, 2015.

[99] U. Praphamontripong, "TESTING WEB APPLICATIONS WITH MUTATION ANALYSIS," PhD Dissertation, George Mason University Fairfax, VA, 2017.

Appendix A

Experimental Data

We described the 5 subject programs used in the experiment. Although, they have been briefly explained in Chapter 6 but here, we give a more elaborate description of the codes.

QuadraticSolver: The QuadraticSolver program is used to get the roots of a quadratic equation. A quadratic equation is an algebraic equation with the degree of two and the form ax^2+bx+c where a, b, c are the coefficients of the equation and x is the unknown. The degree of a polynomial is the highest degree of its monomial (i.e. each term) with non-zero coefficients. The coefficients can be uniquely identified as the quadratic coefficient, linear coefficient, and constant (free term) respectively with $a \neq 0$. If $a = 0$, then the equation is no more a quadratic but rather a linear equation. The values of b and c can be zero, it does not change the characteristics of the equation being quadratic. The program takes three parameters as input and returns two roots as outputs. The outputs can be two distinct real roots or two equal real roots. It can also be two complex numbers.

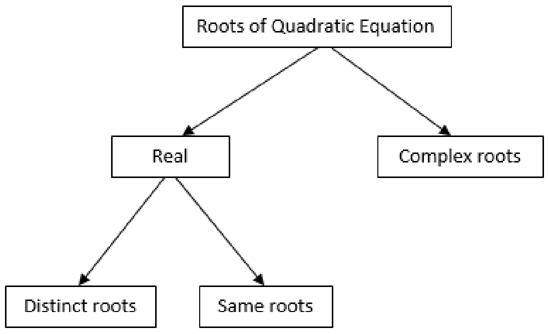


Figure 39: Roots of quadratic equation

TriangleType: The TriangleType program takes three inputs as the sides of a triangle and decides what type of triangle is represented by the three sides. All the three sides of triangle have non-zero values. Any triangle with a side having a zero value, is an invalid triangle. The output can be equilateral, isosceles, scalene or invalid triangle. Also, length of a side should not be greater or equal to the sum of two other sides.

MID: This is a program that takes three input values and return the middle one.

LineRectangleClassifier: This program determines the position of a line with respect to the position of a rectangle. In other words, it determines the relative positions relationship between a line and a rectangle. It takes eight input variables, four out of them (xr1,xr2,yr1,yr2) denote the coordinates of a rectangle and the remaining four variables (xl1,xl2,yl1,yl2) denote the coordinates of a line. It returns one of the following four outputs:

- ✓ The line is wholly inside rectangle,
- ✓ The line is partially inside rectangle,
- ✓ The line is wholly outside rectangle, and
- ✓ Invalid line or rectangle coordinates.

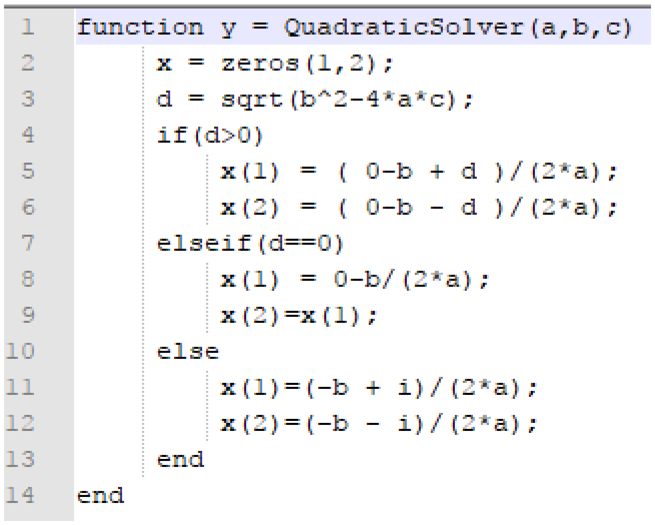
PointCircleClassifier: This program investigates the position of a given point with respect to a given circle by examining the center coordinates of the circle, its radius and the coordinates of the point. It returns one of three outputs as follows: point is outside, point is inside and point on the circumference of the circle.

Appendix B

Codes of Programs under Test

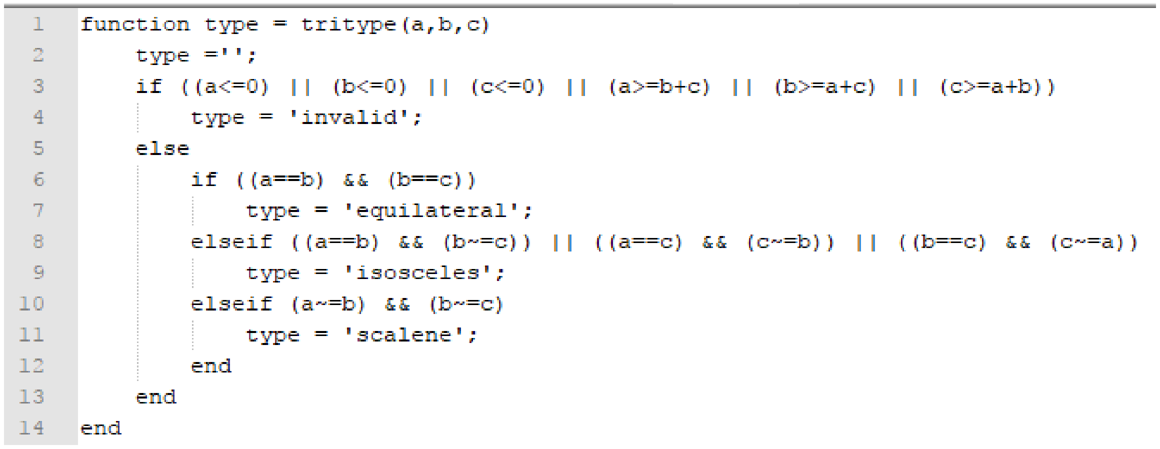
The program code is presented here:

QuadraticSolver.m



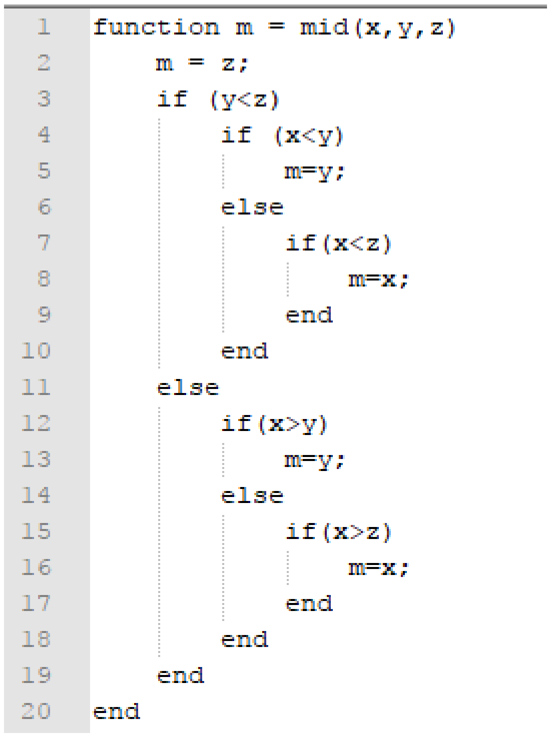
```
function y = QuadraticSolver(a,b,c)
    x = zeros(1,2);
    d = sqrt(b^2-4*a*c);
    if(d>0)
        x(1) = ( 0-b + d )/(2*a);
        x(2) = ( 0-b - d )/(2*a);
    elseif(d==0)
        x(1) = 0-b/(2*a);
        x(2)=x(1);
    else
        x(1)=(-b + i)/(2*a);
        x(2)=(-b - i)/(2*a);
    end
end
```

TriangleType.m



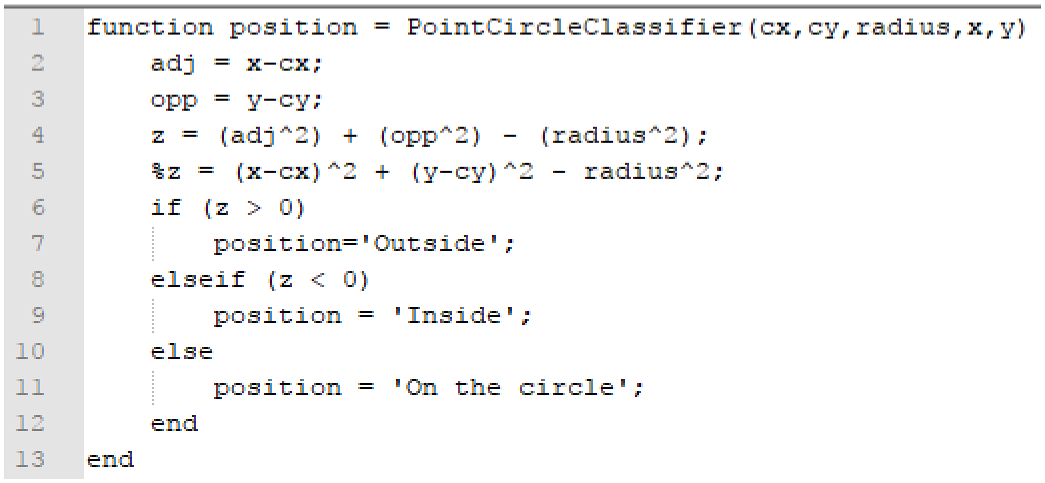
```
function type = tritype(a,b,c)
    type ='';
    if ((a<=0) || (b<=0) || (c<=0) || (a>=b+c) || (b>=a+c) || (c>=a+b))
        type = 'invalid';
    else
        if ((a==b) && (b==c))
            type = 'equilateral';
        elseif ((a==b) && (b~=c)) || ((a==c) && (c~=b)) || ((b==c) && (c~=a))
            type = 'isosceles';
        elseif (a~=b) && (b~=c)
            type = 'scalene';
        end
    end
end
```

MID.m



```
function m = mid(x,y,z)
    m = z;
    if (y<z)
        if (x<y)
            m=y;
        else
            if(x<z)
                m=x;
            end
        end
    else
        if(x>y)
            m=y;
        else
            if(x>z)
                m=x;
            end
        end
    end
end
```

PointCircleClassifier.m



```
function position = PointCircleClassifier(cx,cy,radius,x,y)
    adj = x-cx;
    opp = y-cy;
    z = (adj^2) + (opp^2) - (radius^2);
    %z = (x-cx)^2 + (y-cy)^2 - radius^2;
    if (z > 0)
        position='Outside';
    elseif (z < 0)
        position = 'Inside';
    else
        position = 'On the circle';
    end
end
```

Appendix C
Theses/Dissertations on Mutation Testing

Summary of Master's and PhD theses on mutation testing

Author	**Thesis Title**	**MSc/ PhD**	**University**	**Year of Pub**
Acree [67]	On Mutation	PhD	Georgia Institute of Technology	1980
Hanks [68]	Testing COBOL Programs by Mutation	PhD	Georgia Institute of Technology	1980
Budd [69]	Mutation Analysis of Program Test Data	PhD	Yale University	1980
Tanaka [70]	Equivalence Testing for FORTRAN Mutation System Using Data Flow Analysis	PhD	Georgia Institute of Technology	1981
Offutt [71]	Automatic Test Data Generation	PhD	Georgia Institute of Technology	1988
Craft [72]	Detecting Equivalent Mutants Using Compiler Optimization Techniques	Master	Clemson University	1989

Choi [73]	Software Testing Using High-Performance Computers	PhD	Purdue University	1991
Krauser [74]	Compiler-Integrated Software Testing	PhD	Purdue University	1991
Fichter [75]	Parallelizing Mutation on a Hypercube	Master	Clemson University	1991
Lee [76]	Weak vs. Strong: An Empirical Comparison of Mutation Variants	Master	Clemson University	1991
Zapf [77]	A Distributed Interpreter for the Mothra Mutation Testing System	PhD	Clemson University	1993
Delamaro [78]	Proteum – A Mutation Analysis Based Testing Environment	PhD	University of Sao Paulo	1993
Wong [79]	On Mutation and Data Flow	PhD	Purdue University	1993
Pan [80]	Using Constraints to Detect Equivalent Mutants	Master	George Mason University	1994
Fleyshgakker [81]	Techniques to improve the performance of mutation analysis	PhD	The City University of New York	1994
Untch [82]	Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method	PhD	Clemson University	1995
Ghosh [83]	Testing Component-Based Distributed Applications	PhD	Purdue University	2000

Ding [84]	Using Mutation to Generate Tests from Specifications	Master	George Mason University	2000
Okun [85]	Specification Mutation for Test Generation and Analysis	PhD	University of Maryland Baltimore	2004
Ma [86]	Object-Oriented Mutation Testing for Java	PhD	KAIST University in Korea	2005
May [87]	Test Data Generation: Two Evolutionary Approaches to Mutation Testing	PhD	University of Kent	2007
Bradbury [88]	Using Program Mutation for the Empirical Assessment of Fault Detection Techniques: A Comparison of Concurrency Testing and Model Checking	PhD	Queen's University Kingston	2007
Hussain [89]	Mutation Clustering	Master	King's College London	2008
Adamopoulos [90]	Search Based Test Selection and Tailored Mutation	Master	King's College London	2009
Hook [91]	Using Code Mutation to Study Code Faults in Scientific Software	Master	Queen's University, Ontario	2009

Kaminski [92]	Applications of Logic Coverage Criteria and Logic Mutation to Software Testing	PhD	George Mason University	2010
Debroy [93]	Towards the Automation of Program Debugging	PhD	The University of Texas	2011
Zhou [94]	Mutation Testing for Java Database Applications	PhD	Polytechnic Institute of New York University	2012
Sarkar [95]	Testing database applications using coverage analysis and mutation analysis	PhD	IOWA State University	2013
Hays [96]	A fault-based model of Fault Localization Techniques	PhD	University of Kentucky	2014
Movva [97]	Automatic Test Suite Generation for Scientific MATLAB Code	Master	University of Minnesota	2015
Li [98]	The Use of Software Faults in Software Reliability Assessment and Software Mutation Testing	PhD	The Ohio State University	2015

Prapha-montripong [99]	Testing Web Applications with Mutation Analysis	PhD	George Mason University	2017

	BEST FITNESS VALUES									
Experiment No	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Run 6**	**Run 7**	**Run 8**	**Run 9**	**Run 10**
1	33.7740	32.1602	37.9041	39.4930	33.2757	36.7126	34.3864	38.3350	37.6885	31.6725
2	38.5071	35.6056	39.2961	36.9667	35.8279	38.1540	38.7901	39.8891	30.0052	38.6544
3	39.9037	33.0026	34.9158	37.4098	35.0061	38.9077	34.3358	33.0678	37.3812	33.6207
4	39.1398	34.9611	28.2038	29.4503	38.3525	33.8116	38.1383	30.5129	34.6275	35.5586
5	34.1624	33.6677	36.5198	37.1225	35.3362	33.6478	39.5835	32.6566	32.8457	33.1363
6	32.4155	36.6058	38.7388	35.9978	35.5122	33.1925	32.2262	38.0534	38.3688	34.3484
7	32.9217	35.0007	38.6311	38.7342	37.3219	39.6811	39.5449	32.9016	37.1863	35.8464
8	38.1256	37.2233	35.0519	34.4001	34.7211	39.3514	35.6501	35.5400	35.6335	39.5623
9	33.7530	39.0592	32.1590	34.7341	38.1282	34.7424	36.9505	35.6242	32.0813	36.7926
10	36.8125	37.1953	34.7418	35.9464	37.6142	39.1024	32.4405	32.7869	37.1983	38.1126
11	36.9673	38.3572	38.7063	37.6003	36.7386	35.7500	38.9305	36.3107	37.2223	38.7747
12	38.9481	38.0093	35.3550	32.0018	33.1957	34.1907	38.9794	36.8100	34.5695	34.2743
13	35.4825	39.2301	39.4008	36.0423	37.0207	37.7541	32.1913	36.5995	32.3723	35.3803
14	35.7419	32.1810	32.5206	39.3917	36.2731	34.9344	34.9116	33.2110	33.1969	34.8064
15	34.6877	38.2722	35.8939	35.7184	33.0500	39.0911	37.3965	38.6813	37.2519	39.8713
16	39.8383	34.0012	36.9966	37.8259	35.9854	38.7986	33.5273	32.9932	32.0223	33.2236

Figure 40: Results of experiment to select the best parameter set for tGA

Vitae

Name	:HAYATULLAHI BOLAJI ADEYEMO
Nationality	: NIGERIAN
Date of Birth	:12/16/1986
Email	:HAYATU4ISLAM@YAHOO.COM
Address	:No 9 Olofa way, Offa Kwara State, Nigeria
Academic Background	:Bachelor of Science Computer Science Usmanu Danfodiyo University, Sokoto