Genetic Algorithm for improving the Mutation Testing

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering

in

Software Engineering

Submitted By

Rupinder Kaur

(Roll No. 801531012)

Under the supervision of:

Mr. Vinay Arora
Assistant Professor

Thapar University, Patiala

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

June 2017

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Genetic Algorithm for improving the Mutation Testing*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Vinay Arora* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Rupinder Kaur)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mr. Vinay Arora)
Assistant Professor
Computer Science and Engineering Department.

Abstract

Software Testing is an approach where different errors and bugs in the software are identified. To test software we need the test data and test cases. In our research work, we proposed an approach for refining the test case using Evolutionary Algorithms (EA) and test the software to detect the presence of errors, if any. We have taken two measures *viz*. path coverage and adequacy criterion to test the validation of our approach. In our approach, we have used Genetic Algorithm (GA) to refine the test cases. We are executing the test cases on three mutation operators using genetic algorithm.

Our method combines random generation and refinement. Each test case is generated randomly in the first step, and then a set of test cases is refined by the genetic algorithm. To measure the adequacy of the test case set, we have used mutation scores, which are based on the mutation analysis of software testing. In our proposed method, it is applied on a C language program, automatically generates test case sets with 100% branch and boundary value coverage.

We are using genetic algorithm to get the optimal test cases that covers all the feasible test paths from some initial random test case. Path coverage based testing approach generates reliable test cases. A test case set is reliable if its execution ensures that the program is correct on all its inputs. But, Adequacy requires that the test case set detect faults rather than show correctness. Hence, for adequacy based testing we are using the concept of mutation analysis. For this we are using mutation score by injecting mutants in the resultant effective test cases.

Key Words: Software Testing, Mutation Testing, Genetic Algorithm.

Acknowledgement

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life. It is a great privilege to express my gratitude and admiration towards my respected supervisor **Mr. Vinay Arora,** Assistant Professor, Computer Science & Engineering Department. They have been an esteemed guide and great support behind achieving this task. This work would not have been possible without the encouragement and able guidance of them. I also thank my supervisor for their time, patience, discussions and valuable comments. Their exultation and optimism made this experience both rewarding and enjoyable. I am truly grateful to her for extending her total co-operation and understanding whenever, I needed help and guidance from her. I am also heartily thankful to **Dr. Maninder Singh**, Associate Professor and Head, Computer Science & Engineering Department and **Dr. Ashutosh Mishra**, PG coordinator, for motivation and providing uncanny guidance and support throughout the preparation of the thesis report.

I will be failing in my duty if I do not express my gratitude to **Dr. S. S. Bhatia**, Senior Professor and Dean of Academic Affairs, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

I am also thankful to the entire faculty and staff members of Computer Science and Engineering Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable. Last but not least, I would like to thank my family for their wonderful love and encouragement, without their blessings none of this would have been possible.

Rupinder Kaur

(801531012)

Table of Contents

Certificate……..…….i

Abstract…….……....ii

Acknowledgement…….……..iii

Table of Contents……..iv

List of Figures…...……v

List of Tables……vi

Chapter 1: Introduction……...1

1.1 Software Testing…….…….....1

1.1.1 Necessity of Software Testing……2

1.1.2 Principles of Software Testing…….…..…….2

1.2 Types of Testing……..……...4

1.2.1 White Box Testing……..……5

1.2.2 Black Box Testing……..5

1.3 Mutation Testing……..…….....5

1.3.1 Basic Steps for Mutation Testing…….….7

1.3.2 Types of Mutation Testing……....7

1.3.3 Advantages of Mutation Testing…….……..8

1.3.4 Basic Assumption of Mutation Testing…….....8

1.3.5 Mutation Score…….…..9

1.4 Genetic Algorithm…….……...….9

1.4.1 Genetic Algorithm Operators……..……10

Chapter 2: Literature Survey…….…….…….....12

2.1 Analysis of Mutation Testing…….…….…..…..……..12

2.2 Cost Reduction Techniques in Mutation Testing…….……13

2.2.1 Techniques for reducing the Execution Cost…….....13

2.2.2 Techniques for reducing the Runtime Cost…….....14

2.3 Mutation Testing Using Genetic Algorithm……15

Chapter 3 Gap Analysis and Problem Statement…….……19

3.1 Gap Analysis in Existing Work……..……..……19
3.2 Problem Statement…….……....19
3.3 Objectives…….……..20
Chapter 4 Proposed Methodology…….…….....21
4.1 Model Description …….....21
4.1.1 Sample Source Code……...22
4.1.2 Test Path Generation……..22
4.1.3 Refinement of Test Sequences…….....22
4.1.4 Application of Improved Genetic Algorithm ……23
4.1.5 Processing of Test Paths using Mutation operators……...23
4.2 Proposed Algorithm…….....24
Chapter 5 Implementation …….……...25
5.1 Implementation of Improved Genetic Algorithm on Test Cases……25
5.2 Hardware and Software Requirements……...…….....27
Chapter 6 Experimental Results …….…….28
Chapter 7 Conclusion and Future …….…….....32
7.1 Conclusion …….……..…….....31
7.2 Future Scope …….……..…….....31
References……..……..……33
Plagarism Report……..…….…38

List of Figures

Figure 1.1: Software Testing Principles.....2
Figure 1.2: Software Testing Types.....4
Figure 1.3: Process of Mutation Testing.....6
Figure 1.4: Steps for Mutation testing.....7
Figure 4.1: Proposed Methodology 21
Figure 5.2: Test Cases Generated through Path-Crawler Tool.....25
Figure 6.1 Comparing the Results of Different operators on Best Test Cases.....29
Figure 6.2 Comparison between Mutation operators on the basis of time 30

List of Tables

Table 5.1 H/W and S/W Requirement.....27
Table 6.1 Fitness of different operators for best test cases…….……28
Table 6.2 Time of different operators for best test cases…….……29
Table 6.3 Mutants generation for the Best test cases…….…….....31

Chapter 1

Introduction

Software testing is process applied on software product or application for producing correct and reliable software for the customer or stakeholders. This phase is the most important of the phases of the Software Development Life Cycle (SDLC) [1]. Instead of single activity, software testing is a process. It is a planned and disciplined process. The quality of software product is reflected by the quality of processes used in testing.

Software development includes steps such as requirement gathering and analysis, module designing, module coding, and system designing, there are various types of software testing exists in SDLC, which includes unit level testing, integration level testing, and system level testing [2].

1.1 Software Testing

The Software Testing is the process of finding the errors or bugs from the program. The reduced numbers of bugs or errors makes the software more efficient and reliable. Testing reveals out the error and the process of finding out the cause of error and removing it, is known as debugging [3]. An error is a conflict among the predictable results and the definite results. The error's greatest source might be tracked to faults that are accomplished in the software development phases [4].

The main sub-components for software testing are discussed as follows:

- The verification is the process, in which it is confirmed that if the software is meeting its industrial specifications.

- The validation is the process, in which the software meets all its business necessities. Requirements deliver the facts on how the data will be outlined, formed and displayed.

1.1.1 Necessity of Software Testing

Software Testing is really necessary because mistakes are made by each individual. Some mistakes are ignorable, but many of them can't be ignored which sometimes results in so many expenses. Everything needs to be checked again and again which is produced by human because humans make mistakes. So there is big chance of mistake in anything produced by human. So, testing has to be performed to examine the quality of product [5]. In comparison to the other phases of SDLC, maximum effort, time and cost is spent on testing phase. If the testing is unplanned then the whole time, cost and effort are useless. Therefore, it is advised to set up systematic strategies to perform testing on software [6]. For effective testing, some efficient technical review should be done. By achieving so, many bugs would be disappeared before the starting the test. Testing initiates at component level and it works by integrating the whole systems. There are many testing strategies that are suitable for different software engineering techniques at the distinct points of time. Testing is done under supervision of the software developer and a self-sufficient testing group. Testing and debugging both are distinct actions, but debugging or fixing the error is the action that can be taken in any of the testing strategy.

1.1.2 Principles of Software Testing

The principles given in the diagram are the basic guideline for both, Software Testing and coding [7].

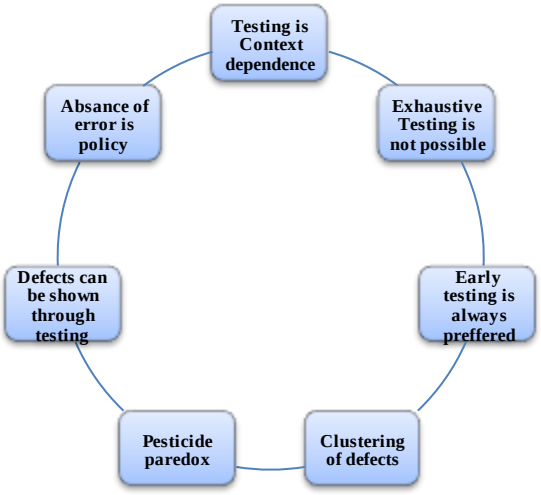


Figure 1.1 Software Testing Principles [7].

The points mentioned in the figure 1.1 are described as followed:

- **Testing is Context Dependent**

  The same test sets should not be implemented in the board as different software products have different unreliable requirements, functions and purposes. Various methods, approaches and various types of testing are associated to the type and nature of the software application.

- **Exhaustive Testing is not possible**

  There is no feasibility to test all sets of data and Test Scenarios as it will take too much time.

- **Early Testing is always preferred**

  In the SDLC, testing activities should be started as early as possible and focus on bounded objectives. As early as the preliminary products, such as requirements or design documents are obtainable, we can begin the investigation.

- **Clustering of Defects**

  During testing of software, it can be observed; most of the reported flaws are associated with small amount of modules surrounded by the system.

- **Pesticide Paradox**

  In case, if the same set of tests are kept on running over and over again or multiple times, then it will become hard to discover any type of new error/fault. "Pesticide Paradox" means that regular inspection as well as update of test cases is really important.

- **Defects can be shown through testing**

  Testing to be performed on software product can only expose many defects or bugs existing in the software product. Even after performing the testing on the software application or product exhaustively we cannot say that the product is 100% error or defect free.

- **Absence of Error Fallacy**

  If the system built is unusable and does not fulfil the user's needs and expectations then finding and fixing defects will not help. Also if testing didn't find any defects in the software, it doesn't mean that the software is ready to be used.

1.2 Types of Testing

Software testing is the process of validating an application and its components function as required. This process involves making sure the system does not contain bugs, and that it works as expected. There are many different types of software testing. Each method has a different purpose and provides a unique value to the software development process. In software testing white box and black box testing's major categories and these are divided in various sub categories as shown in given diagram:

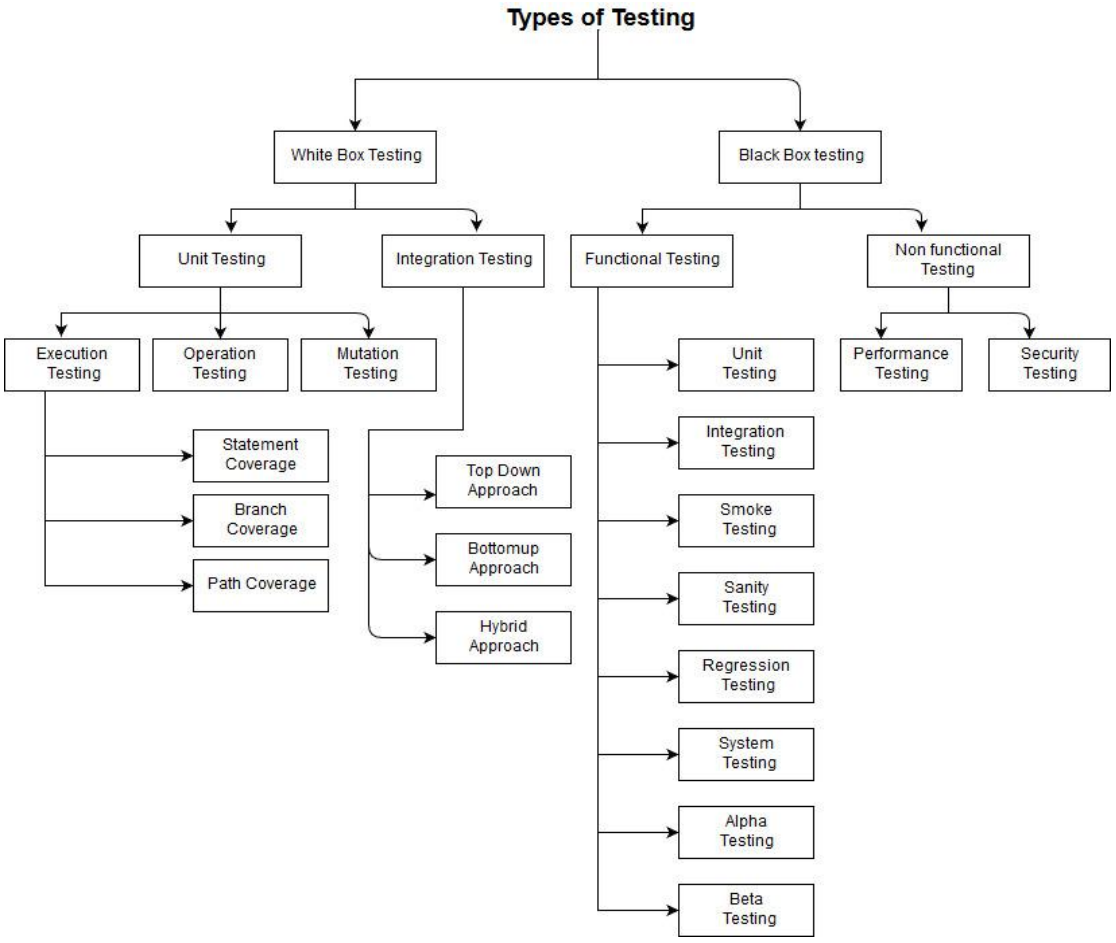


Figure 1.2 Software Testing Types.

1.2.1 White box testing

White Box Testing is the testing of a software solution's internal coding and infrastructure. It focuses primarily on strengthening security, the flow of inputs and outputs through the application, and improving design and usability. White box testing is also called white box analysis, clear box test or clear box analysis. This is a strategy for debugging of software, it is the process of installing and fixing bugs in the computer program code or hardware device see which tester has excellent knowledge about the program components interaction. This method can be used for web service applications, and is rarely practical for debugging in large systems and networks. In addition, the white box test is considered as a safety test method that can be used to verify whether the code implementation is in accordance with this type of design, validates the implemented security functionality, and to highlight exploitable vulnerabilities [8].

1.2.2 Black box testing

Black-box testing is a method of software testing that examines the functionality of an application based on the specifications. It is also known as Specifications based testing. Independent Testing Team usually performs this type of testing during the software testing life cycle. This method of test can be applied to each and every level of software testing such as unit, integration, system and acceptance testing. Black box testing is a software testing techniques in which functionality of the software under test (SUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software. The main purpose of the Black Box is to check whether the software is working as per expected in requirement document & whether it is meeting the user expectations or not [9].

1.3 Mutation Testing

Mutation testing is a kind of structural testing method that focuses upon improving the sufficiency of test suites, to evaluate the count of errors or flaws exists in SUT [10]. The word “mutant” itself means for “change”. Thus the code is tested by making small changes into the original code.

Mutation Testing is also known as fault based testing approach. It delivers a testing standard known as mutation adequacy score. The mutation adequacy score is used to evaluate the efficiency of test set in provisions of its capability for finding the flaws or errors. Such flaws are knowingly, seeded into original program or code by some simple syntactical changes for generating a set of defective programs which are known as mutants, all of them contains various syntactic changes. To enhance the quality of the test set, and mutants have to be execute across the input test set. When the mutants are executed, If the outcomes after executing a mutant are opposed from the outcomes of executing the actual program for any test case in the input test set, it means that the originated fault symbolize that the mutant is discovered. Mutation score is one of the outcomes of the mutation testing which ensures the quality of the input test set. The mutation score is equal to the ratio of number of discovered faults by the sum of number of the originated flaws [11].

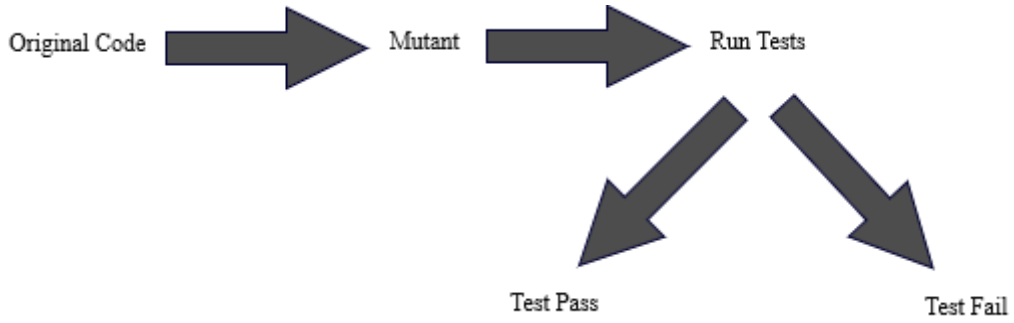


Figure 1.3 Process of Mutation Testing [11].

As shown in figure 1.3, initially the mutants are injected into the original code and the test suits are executed. If the test is pass that means mutation is not found. If the test is fail that means that the mutation or change is found. So, mutation must be found which will consider that the original code works as desired. The mutants which are found are known as killed mutants and the mutants which are not found are known as alive or equivalent mutants.

It's logical to consider that the test-set can find out more errors. The test set is considered as more efficient which covers maximum errors. Mutation testing is proposed by DeMillo

[12].The assumption in mutation testing is that there are limited changes made in a module and then mutated modules are compared to the original program results [13].

1.3.1 Basic Steps for Mutation Testing

Mutation testing begins with test data that is designed by various methods, and is the only fault-guided technique to test the completion of test suits. Test cases also need to be generated to input the test data. Mutation testing follows step by step approach. So, there are basic steps that are followed to perform mutation testing as shown in the figure:

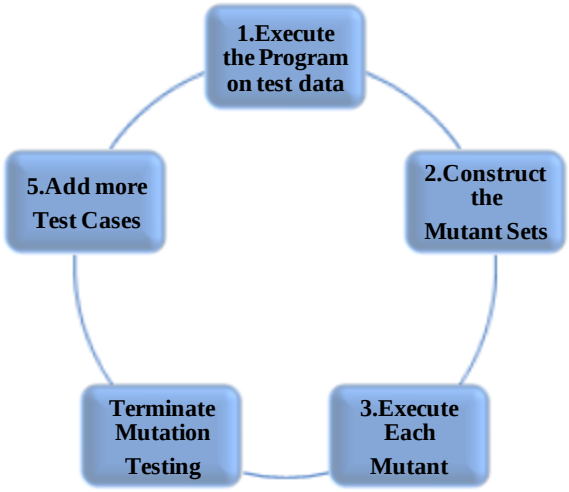


Figure 1.4 Steps for Mutation testing.

If some live mutants exist, then there must be two possible reasons as given below:

- The Mutation is equivalent to the original code.
- The Test Data is not acceptably fine [14].

1.3.2 Types of Mutation operators

- **Value Mutation**

  This mutation includes changing the values of fixed variable or parameter such as loop range is a very common mistake to get started or end up.

- **Decision Mutation**

  This involves modifying the conditions to illustrate the coding errors of potential slips and program conditions.

- **Statement Mutation**

  This includes some strings to corrupt the coding errors or change the sequence of program lines. There are many operations, for *e.g.*, altering tasks in arithmetical expressions .A common mistake might be to skip rise on some variables in a do-while loop [15].

1.3.3 Advantages of Mutation Testing

- It is a powerful approach to accomplish high coverage of the source program.

- This testing is capable brief testing.

- Mutation testing brings a good platform for error detection to the software development team.

- This method uncovers all the ambiguities in the source code, and has the capacity to detect all the faults in the program.

- Customers are benefited from this testing by getting most reliable and steady and efficient system.

1.3.4 Basic Assumptions for mutation testing

- **Competent programmer hypothesis**

  That program which is to be checked or tested is written by a most efficient programmer.

- **The Coupling Effect**

  The Test data, that separates all codes, is just different from the right unit due to common errors, so it is sensitive that they are knowingly more complex errors [14].

1.3.5 Mutation Score

Mutation score (MS) is the ratio of Dead Mutants over the number of Non Equivalent Mutants. The main aim regarding mutation score is to have 100%, score. Which considers that all faults in all mutants have been killed or detected; the more dead mutants the higher the score will be. This method is used for sufficiency testing. The formula for mutation score is given as followed:

$$\mathbf{MS = \mu \backslash \lambda \times 100}$$

In the above equation, λ is the total number of mutants, μ is the killed mutants and MS is mutation score.

1.4 Genetic Algorithm

Genetic algorithm is an optimization algorithm. It is used for finding the optimal solutions or nearly most favorable solution for a given computational problem. A genetic algorithm represents one branch of the field of study called evolutionary computation [17].

The concept of Genetic Algorithm is introduced to replicate procedures in natural system essential for development, explicitly those who follows the fundamentals first laid down by Charles Darwin of continuity of the fittest. It represents an intelligent use of a random search surrounded by certain search space to solve a particular problem [18].

Genetic algorithms are successful in an environment in which there is a large group of candidate's solution and in which the search is uneven and there are several hills and valleys. It is true that genetic algorithms will do well in any environment, but due to the more specific positioning algorithms in simple search places, they will be very bad. Therefore, you should be aware that genetic algorithms are not always the best choice.

Sometimes they can take a lot of time to run and therefore, are not always possible for real time use. However, they are one of the most powerful methods, with who quickly create a high quality solution for a problem. Genetic algorithms are based on the classic view of a chromosome as a string of genes. R.A. Fischer used this scene for mathematical genetics, from which the mathematical formula was specified at which would spread through the population [19].

A genetic or evolutionary algorithm applied to the principles of development found in nature for finding an optimal solution for a particular problem. In "Genetic Algorithm", the problem is encoded in a series of bit strings manipulating algorithms; in an "evolutionary algorithm", decision variables and problem functions are used directly. Most commercial solver products are based on evolutionary algorithms. Genetic Algorithm is useful and works efficiently when search space is considered to be large, complex and poor, when domain knowledge is rare or it is difficult to encode expert knowledge. This is also useful when the search space is required to be reduced and due to the failure of conventional search methods.

1.4.1 Genetic Algorithm Operators

There are three operators through used in Genetic algorithm [20].

- **Selection**

  A selection operator gives preference to the better chromosomes and passes them to the next phases of genetic algorithm and the best chromosomes are chosen by evaluating the fitness function. Fitness function determines the best chromosomes according to their fitness values. There are different selection methods used for selection *viz.* Tournament Selection, Rank based Selection, Roulette wheel selection method *etc.*

- **Crossover**

  The Crossover operator selects genes from parent chromosomes and creates a new child or offspring. Choose appropriate crossover point and everything before this point

copy from a first parent and then everything after a crossover point copy from the second parent. Crossover operation is performed on given chromosomes

Chromosome 1	11011 | 00100110110
Chromosome 2	11011 | 11000011110
Offspring 1	11011 | 11000011110
Offspring 2	11011 | 00100110110

There are many other types of crossover, for example we can choose more crossover points. Crossover can be rather complicated and also depends on encoding of chromosome. Any particular crossover can be used for a specific problem which can improve performance of the genetic algorithm.

- **Mutation**

  After performing crossover, mutation takes place. This is to prevent falling all solutions in population into a local optimum of solved problem. Mutation is used to make changes randomly in the new offspring. For binary encoding, we can exchange a few randomly chosen bits from 1 to 0 or from 0 to 1.

  Mutation is performed on given chromosomes as follows:

  Original offspring 2: 1101100100110110

  Mutated offspring 2: 1101101100110110

Chapter 2

Literature survey

In this research work, the surveys on Mutation Testing for cost reduction, Test data generation techniques, Test Case generation techniques, various evolutionary algorithm Strategies have been performed. The following is the summary of the review performed by various research fellows:

2.1 Analysis of Mutation Testing

Mishra *et al.* [21] proposed a new technique for generating the efficient test data in the context of mutation testing. They described the mutation testing and mutation analysis as fault based testing techniques and used in testing to defeat the limitations of other testing strategies that are followed in software testing. Mutation analysis is the part of mutation testing. In Mutation analysis it is analyzed that which mutation technique or strategy will be used to perform mutation testing .Instead of that mutation testing the sufficiency criteria is tested on the basis of mutation analysis. From the study of past years, mutation testing can be found in a student's paper in 1971 by R. Lipton. Mutation testing can be applied on any phase of testing, *i.e.*, Unit level, Integration level, and System level. The Mutation testing can be used for effective detection of defects. It has also been proved with experience that the mutation test is stronger than the statement and the branch coverage. The mutation testing is more efficient in finding errors as compared to data flow testing.

Dave *et al.* [22] represented the review of the strategies that have been applied with mutation testing for automatically generating the test data which is optimized in context of time, cost and code coverage. This paper is an up-to-date review of the technologies that have been applied with mutation testing for automatic generation of test data which is optimized regarding time, cost and code coverage. The survey reveals an increase in interest regarding the meta-heuristic techniques along with mutation testing. It is also discussed major aim and challenge is to identify the appropriate test data which can kill the maximum mutants as much as possible. Another big challenge is the good mutation

score for generation of test data and selection. Mutation testing is among the strongest and efficient techniques in detecting maximum bugs or faults in any software. Because of higher cost of mutation testing or expenses mutation testing is not generally used practically. Automatic test case generation is the area which has been researched with minimum practical employment and guidance. The major issue in Mutation testing is that it is lacking automated tools for generation of tests, which are the limitation for its practical use in software testing.

2.2 Cost Reduction Techniques in Mutation Testing

Mutation testing is considered as an expensive testing technique. In order apply the Mutation Testing in a Practical Testing approach several cost-cutting techniques have been offered. There are Cost reduction approaches which are separated into three types "Decrease", "Do Fast", and "Do the Smart." The prepared subdivision is separated into two types, less no of mutants generated and decrease in Execution cost [23].

2.2.1 Techniques for Reducing the Execution Cost

Along with deducting the number of prepared mutants Compatibility can also be minimized by adjusting cost of process of mutant execution. The way we decide whether to analyze it during the execution process, if the mutant is killed, an integration checking strategies could be divided in three types. These are: Strong Mutation, Weak Mutation and Firm Mutation.

Howden *et al.* [24] proposed an approach by considering an example of a the program p, the program is called an interactive meter of the p' mutant. If It gives distinct outcome from the meter so it can be killed in Real program p. To implement Strong Mutation, weak Mutation was proposed by Howden. In the weak mutation, a program p is considered to be composed of concept in which there are components C=c1,c2…etc and Cm is said to be changing component is made by adding mutant on component It is said that the mutant should be encountered if any Component Cm Running is separate from the mutant m. The execution of the whole program cannot be waited. The execution point would be tested exactly after the execution point Mutated part.

Offut *et al.* [25] described that the benefit of weak mutation is that every mutant does not require an appropriate implementation process. After the mutated components gets executed we could examine the continuity. Apart from this, it may not be essential to generate each mutant, because obstacles of test data could frequently be set in advance.

2.2.2 Techniques for Reducing the Runtime Cost

Jia *et al.* [12] presented that the Interpreter-Based Techniques are among the various optimization techniques which were used in first generation of tools for mutation testing. In conventional approaches based on Interpreter, the outcome of a mutant gets interpret from the source code straightway. The price for this approach is demonstrated by the price of interpretation. To improve the conventional Interpreter-Based approach, Offutt and King translated the original program into transitional form. The Mutation and interpretation are implemented at the intermediate code level. The tools based on Interpreter ensure advance elasticity and are adequately effective for mutating the small codes.

Byoungju *et al.* [26] represented the techniques basis on the compiler, If every mutant is initially compiled into an executable code, then compiled mutant testing is executed by many test cases. This approach is much faster as in comparison to interpretation techniques of source code because when binary code gets executed then it takes less time as compared to interpretation. After all, also there is speed limitation evolved, called as compilation congestion, because of the high cost of compilation for program code whose runtime is longer as compared to the time of compilation .

Jatana *et al.* [27] represented the opportunities and challenges that are faced when the Search-based techniques applied on the mutation testing. In Past Years, huge amount of research has been covered in this area. The storage and computational cost for a large set of mutants is the major limitation. There are latest researches, available tools and updates to Search Based Mutation Testing (SBMT) are also mentioned in the research work.

2.3 Mutation testing Using Genetic algorithm

Langdon *et al.* [28] defined a multi-objective Pareto technique is adopted by using Monte Carlo sampling, genetic algorithm and genetic programming for searching higher order mutants, which both are really difficult-to-encounter and reasonable. The space of complex flaws or higher order mutants is greater as compared to conventional first order mutations that are corresponding to easy flaws or defects, however search based approaches makes it extensible. The complications of non-determinism and effectiveness are blown away. They also have discussed that 90% of faults which goes through manufacturer's testing processes or procedures are complicated. Because of that, the correspondent bugs fix evolves various kinds of mutants. The complex or higher order mutation testing is used for learning about the bugs' intercourse and their buffet on testing software for detecting the faults.

Haga *et al.* [29] proposed a methodology which generates the test-cases automatically for the software on the basis of mutation analysis and genetic algorithm. They used the technique of collaborating random generation and enhancement. Every test-case gets generated randomly in the initial step, and then a group of test-cases is filtered through the genetic algorithm. For evaluating the sufficiency of the test cases, mutation score is being used, that are on the basis of mutation investigation of software testing. In proposed strategy, that has been performed on C programming language program, automatically generating test case sets with 100% branch coverage and boundary coverage. The time spent for generation of one test-case set was almost 130 ms.

Khan *et al.* [30] proposed an approach for test cases have been automatically generated through genetic algorithm and mutation analysis. The Software Engineers waste mainly there time upon the procedure of software testing. It is also has been observed that in the industries most of money is exhausting on the software process. In process of software testing test cases are applied as input and are checked for the final outcome. Here, the initial concern is to choose those test cases that are effective for the process of software testing. For the correct or effective output, it is really hard to select effective test cases..

For generating the test cases automatically many nature inspired optimization algorithms have been used.

Louzada, Jailton, *et al.* [31] They have proposed an approach of using for Genetic Algorithm's elitist approach as a tool for generating and selecting the test-data that is implemented in Mutation Testing for various benchmarks. The outcomes indicated a fine completion of the algorithm that is used over various benchmarks. They described that the elaboration of an effectual methodology for generation of test data is really a provoking mechanism that straightway smashes the time which is spent on the activities which are related to software testing. Thus, many of the researches that are affiliated to this area are discussed in this paper. Through many of the techniques which are being used for generating test data automatically, meta-heuristics is being highlighted, which comes under optimistic area known as Search-Based Software Testing.

Langdon *et al.* [32] discussed that they are able to find examples that cause challenges to testing in the higher order space that cannot be represented in the first order space. Also, it is the mutation testing has been proved as dominant approach for detecting faults or defects from the software application or product. However, it comes with a limitation because it is very costly and some of the important traditional mutants that are similar to actual defects are blended with many others that reflect impractical flaws. The complications of expenditure and realism have been an important barrier to the technical upliftment of mutation testing. Genetic programming has been used for replacing high order mutants. Space is very large, from conventional first order mutation space of easy defects to higher order mutation space. Anyhow, the use of a search-based strategy makes it extensible, it simply seeks mutants that challenge the tester, whereas this idea of difficult flaws resolves the problem of reality; it has been studied that 90% of the actual defects are complicate or difficult.

Liu *et al.* [33] proposed Modified Genetic Algorithm for this problem of suffering from a large run-time and low sufficiency in Test Case Generation. In this paper the algorithm carries the principle of real number coding and logic coverage, while improving the fitness function. In Apart from this, it is called genetic-oriented control..The premature

convergence program shows experimental results that MGA is fast convergence motion and higher test data generation capacity than conventional genetic algorithms.

Fraser *et al.* [34] represented that the automatically done software testing, without full formal specification manual check is needed to detect errors, it makes production desirable by keeping the amount of tests as petite as feasible, like the strongest possible test set. Generally applicable coverage criteria as branch coverage are probably faint, mutations. Although the test has been introduced as a strong determinant factor, mutation-based tests are influenced by generation because generally there are number of mutants, and lots of them are moreover trivial or correspondent, on these kind of mutants, any attempt to test case generation for each definition will be exhausted for each definition To defeat this problem, our search-based test generation by EVOSUITE equipment having two novel optimizations Micro co-ordinates *viz.* First, unnecessary testing performance was avoided and the state on mutants was monitored, and secondly they had used the entire trial suite Generation to customize test suits to kill the uppermost amount of mutants, somewhat than by choosing particular mutants, these optimizations permitted them to implement EVOSUITE. A unsystematic trial of hundred's open source projects, which included the sum of 8, 963 classes and further Compared to 20 million Lines of Code (LOC), in which there are total 1,380,302 mutants which shows that the well-scales of their proposed approach, mutation testing is a feasible test decisive factor .Automatic test case generation for the equipment, and allowed them to examine the association Branch coverage and mutation testing in detail.

Jia *et al.* [35] investigated the Higher Order Mutants (HOM).As in conventional mutation testing only consider the first order Mutants formed by injection of a mistake frequently these first order mutants reflect the insignificant flaws which can be simply encountered. Then they investigated HOM. The paper reports the results of an empirical study into subsuming HOMs, using six benchmark programs. HOM Benchmark program is the largest study of mutation. For removing exponential explosion considered in the number of mutants, the paper introduced a search based approach for identification of subsuming HOMs. The outcomes are represented for the genetic algorithm, greedy algorithms and mountain climbing algorithm.

Mala *et al.* [36] represented that in the software testing process, selection and generation of test cases, requires a lot of human interference. Such manual test procedure leads to Ineffective software which in turns requires more time and cost .To make the software Zero-defect Software the number of test cases will be required for testing which will be infite.Since in software testing studies have shown that exhaustive testing of a Software is not feasible. So for testing the test cases needs to be reduced and only effective or optimal test cases should be executed which will be more efficient in finding out the errors will be unlimited, since complete testing is not feasible, they have to decrease the amount of test cases by choosing There is a high potential to identify optimal test cases Under the Test Errors in Software Under (SUT). In our approach, we proposed a that can generate optimal test cases by using Hybrid Genetic Algorithm (HGA) based novel testing methodology from the set of test cases based on the mutation score that achieves high statement coverage criterion by finding more injected bugs in the SUT. At last, they compared the simple genetic algorithm (SGA) based strategy for Test case optimization and proven that, the HGA-based strategy is generating better optimized results.

Fraser *et al.* [37] described for evaluating the adequacy of the test suite, mutation analysis is seeds in artificial faults programs. The non-detected Mutation demonstrates a weak point in the test suite. So, they proposed an approach for unit testing that can detect the mutants for object oriented classes. It has two advantages for: First one is, the resulting test suite is optimized towards finding the defects modeled by mutation operators rather than covering code. Second, the state change caused by mutations induces oracles that precisely detect the Mutants. Evaluation is done on 10 open source libraries.

Chapter 3

Gap analysis and Problem statement

3.1 Gap analysis in existing work

Based on the literature review of Mutation testing, Genetic Algorithm, Test case generation techniques and Anomalies present in Mutation Testing, following gaps have been identified.

- Mutation testing is cost and time consuming for testing the software [29], and domain can be extended to reduce the testing cost and to improve the effectiveness of test cases.

- Little work has been reported to improve the test suits, in conjunction with mutation analysis [34].

3.2 Problem statement

In SDLC, software testing consumes 80% of the total cost involved; hence testing is the most important phase. In Software engineering exhaustive testing is not possible. Mutation testing is the most comprehensive testing technique, but it is extremely costly and time consuming, since there are many mutant programs that are needed to be generated. So, we need to reduce the cost of mutation testing. Thus, for reducing the cost we are reducing the test cases. If only effective test gets selected and executed then it will give more effective results and to get the optimal test cases we are using modification in the genetic algorithm. In our research work we are refining the test cases for a particular intermediate sample source code.

The test cases are generated through the path crawler tool which is a structural testing tool and generates the test cases automatically by covering all possible paths. Applying genetic algorithm we are refining the test cases by considering their weight-age factor related to length of path. We have executed the code by using two mutation operators in genetic algorithm and analyzed the results as best test cases along with their fitness. After

getting the best test cases the mutants will be injected and then the mutation score will consider the adequacy.

3.2 Objectives

- To Study and explore research in the field role of mutation testing in test generation related to test.
- To propose a framework to optimize test cases using mutants in genetic algorithm.
- To implement the proposed framework.

Chapter 4

Proposed Methodology

We proposed a test case selection strategy for analysis and optimization of the selected test cases in Mutation Testing using the GA. We are using this strategy to reduce the number of test cases which will automatically result in reducing the cost of mutation testing. In our proposed approach, we have generated test cases by using the Path Crawler tool. Since, a large number of test cases arise; we have refined those test cases according to their weight-age. Then the improved GA is applied on the generated chromosomes and fitness and time of each chromosome is calculated.

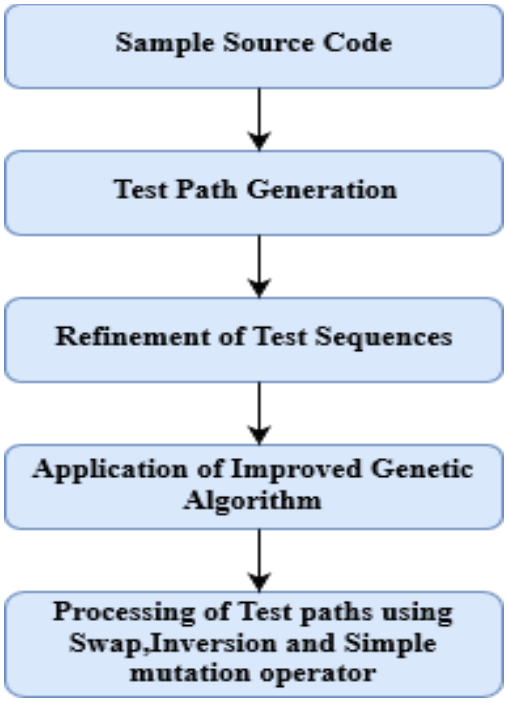


Figure 4.1 Proposed Methodology.

4.1 Model Description

In the proposed methodology, source code of c program has been given as input to test case generator tool to generate test cases. Mutants have been injected along with input source program. After this weight-age factor is combined for selecting test cases for

initial population in the genetic algorithm. Then the improved version of Genetic algorithm is applied on chromosomes in initial population set and fitness of each chromosome is calculated.

After performing selection, crossover and mutation the results are analyzed by identifying the best test case. Results are analyzed by using different mutation operators at different Mutation Rates (MR).

4.1.1 Sample Source Code

The source code can be any program in C language for which the test cases need to be generated. We have picked a simple code and input that code to the Path crawler Tool for path generation.

4.1.2 Test Path Generation

Test cases are designed on the basis of the functions of a specific application; hence it differs from application to application. In the case of testing, the original results are validated against the expected results. On the basis of final results, test case is either modified or kept as it is. Test case is modified in the case if does not works well. Test case generation is the practice of creating test suits to detect system errors. A test suite is a group of relevant test-cases collected together. Test case generation is the most important and fundamental process of software testing.

4.1.3 Refinement of Test Sequences

After the generation of test cases, there will be large number of test cases generated. We cannot take them entirely as the initial population in genetic algorithm as it will result in more complications. Thus we are refining a set of test cases according to the weight-age assigned. We have given the weight-age according to the length of test paths .The test cases which are having test path of maximum length will be refined and passed to the next phase for execution i.e. making them initial population in genetic algorithm. Thus by refining the test cases we will be having some of the effective test cases which will result in optimized results.

4.1.4 Application of Improved Genetic Algorithm

The genetic algorithm is an evolutionary algorithm and known for its optimized results. So in our approach we are using genetic algorithm for generating effective set of test cases. After refinement of the test cases the resultant *i.e.* refined test cases will be taken as initial population in the genetic algorithm. We will take those refined test cases as chromosomes. Following genetic algorithm there are different operations *i.e.* Selection, Crossover and Mutation are performed on the population set. In selection section of genetic algorithm, we are using random selection, and taken the crossover point as 1, with operators *viz* Inversion, Swap and Simple.

The GA has been executed with different mutation operator at varying mutation rates. The GA is executed on program under consideration 10 times or we can say 10 iterations are executed for all the given operators on different mutation rates.

4.1.5 Processing of Test Path using Swap, Inversion and Simple Mutation operator

Test paths are processed through the mutation operators' *viz.* Swap, Inversion and Simple in genetic algorithm. After evaluating the results through the given mutation operators we are comparing their results to analyze that which works operator better for our approach. These given operators are described as follows:

- **Swap Mutation**

In swap mutation, we select two positions on the chromosome at random, and interchange the values. This is common in permutation based encodings.

- **Inversion Mutation**

In inversion mutation, we select a subset of genes like in scramble mutation, but instead of shuffling the subset, we merely invert the entire string in the subset.

- **Simple Mutation**

In Simple mutation, we select one or more random bits and flip them. This is used for binary encoded GA.

4.2 Proposed Algorithm

Step 1: Initialize the population.

Step 2: Calculate the value of fitness Function.

Step 3: While (fitness value! =completion criteria)

Step 4: Select Chromosome.

Step 5: Apply Crossover on Chromosome.

Step 6: Apply Mutation on Chromosomes.

Step 7: Calculate Fitness function.

Step 8: END.

Chapter 5

Implementation

5.1 Implementation of Improved Genetic Algorithm on Test Cases

We are reducing the test cases by refining effective test cases from the total count of test cases with the help of modifications in genetic algorithm. As genetic algorithm helps to generates the optimized results. This will result in reduced test cases. The steps that are followed for the implementation of proposed methodology are as follow:

1. First step is to input the Program to the Path crawler tool for the automatic generation of test cases.

2. After that, the test cases are generated as given below:

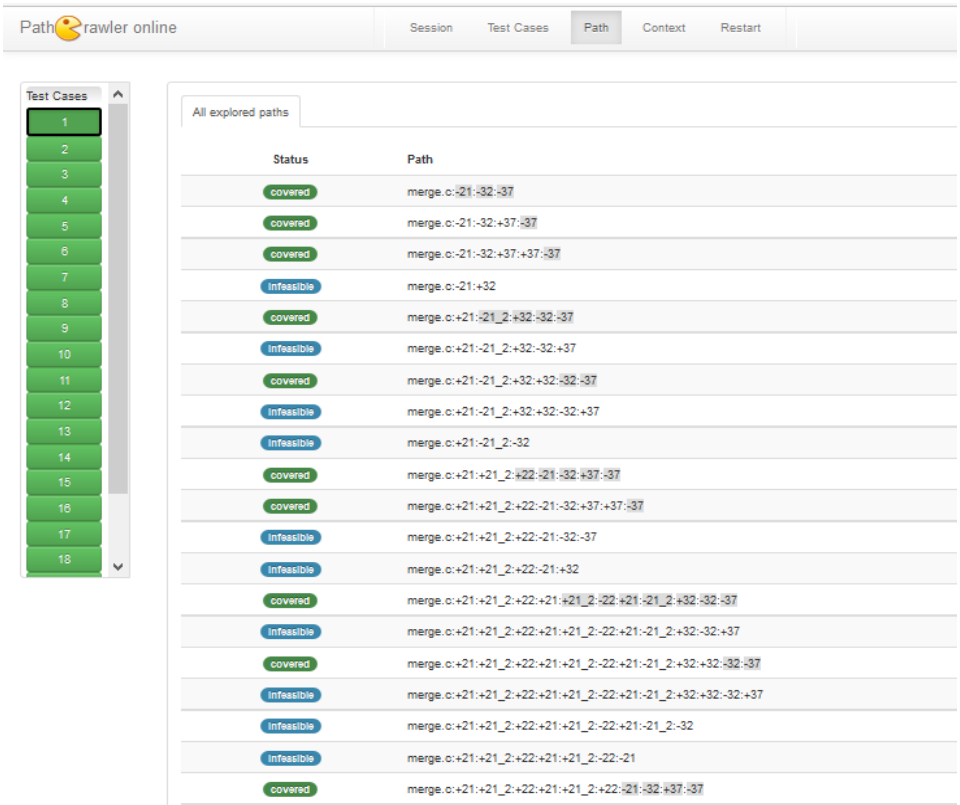


Figure 5.1 Test Cases Generated through Path crawler tool.

There are 20 test cases generated for the sample program and the status shows whether the given path is under the set covered (visited) or uncovered. So, infeasible paths are un-visited. And the test paths are showing the line no which are visited.

3. The next figure is the sample that how particular test paths of test cases are generated corresponding to the source code.

4. As the test cases are generated we took only 6 sample test cases and consider these as initial population in the genetic algorithm for performing operations.

```
Y.append ("-21-32-37")
Y.append ("-21-32+37-37");
Y.append ("-21-32+37+37-37");
Y.append ("-21+32");
Y.append ("+21-21_2+32-32-37");
Y.append ("+21-21_2+32+32-32+37");
```

5. For refinement of test case we applied a condition related to the length of test path. When length of test path is more than 12 (*i.e.* total count of the whole string ("-21-32-37") is greater than 12) only that test case is allowed for next level, otherwise it is eliminated.

6. After the refinement of test cases. These test cases are taken as genes and new chromosomes are generated from them.

7. Then Fitness and time of each chromosome is calculated along with fitness of genes also.

8. The selection method is taken as Random, Crossover type is 1 point crossover, and three Mutation operators *i.e.* swap operator, inversion operators and Simple have been used. Results are analyzed on different mutation operators and different mutation rates i.e. 0.2, 0.4, 0.6, 0.8.

9. The code is to be executed at least 10 times or more than that to get the better results.

10. Then the best chromosomes along with best test cases, their best fitness and best time come across as our results.

11. Then the mutants are injected on the best test cases. Then the code is executed if mutants get killed or not.

12. Mutation score is to be calculated for the best test cases.

5.2 Hardware and Software requirements

Table 5.1 H/W and S/W Requirement.

1.	RAM	4 GB
2.	System Type	64 bit/ operating system
3.	Hard Disk	500GB
4.	Operating System	Windows 8.1
5.	Programming Language	Microsoft Excel, Python

Chapter 6

Experimental Results

In order to conduct the simulation and experimentation, we have selected the 6 test cases from the set generated by Path Crawler Tool, which are pruned further on the basis of their weight-age (and sub-set includes 4 test cases). We have used permutation encoding as one of the techniques to encode the initial solution. We have a set $T` = \{t_2\ t_3\ t_5, t_6\}$, as selected test cases. In this process we have used the iteration method to select the best fitness values for the given test cases. Here, we have applied single crossover rate 0.8 on different mutation rates such as 0.2, 0.4, 0.6 and 0.8. For each iteration the fitness values and time are varying for given operators. We have evaluated the results using three Mutation operators *viz.* Swap, Inversion and Simple. Each mutation rate has been iterated 10 times to get the best solution.

In the iteration we have calculated the average of the fitness values and average of the time for each mutation rate.

After this the results for different test cases that are identified through execution of given mutation operators have been compared. Following the specified iterations, we have computed the average of fitness values for each test case. Table 6.1 shows the results in context to the best test case on the basis of fitness by using different mutation operators.

Table 6.1 Fitness of different operators for best test cases.

Best Test Case	Average of Fitness		
	Simple Mutation operator	Inversion Mutation operator	Swap Mutation Operator
6	266.264	278.083	254.910
5	301.104	269.871	258.643
2	382.122	291.828	271.997
3	296.468	255.983	252.875

Now, again by following the specified iterations, we have computed the average of time for each test case. Table 6.2 shows the results in context to the best test case on the basis of time by using different mutation operators.

Table 6.2 Time of different operators for best test cases.

Best Test Case	Average of Time		
	Simple Mutation operator	Inversion Mutation operator	Swap Mutation operator
6	3.644	2.788	2.549
5	2.864	2.652	2.561
2	3.182	2.966	2.743
3	3.536	3.012	2.935

Figure 6.1 gives comparison between the mutation operators distinctly on the basis of fitness value. In this graph Y-axis shows the Fitness value for all the mutation operators used, whereas X-axis shows the best test case. This graph shows the average of fitness for different test cases. We are comparing the average of best fitness of the given mutation operators in context to best test case.

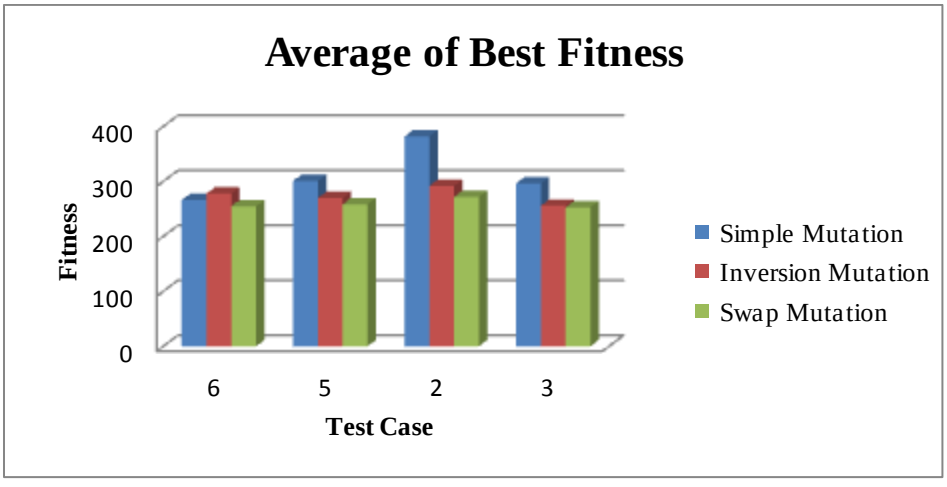


Figure 6.1 Comparing the Results of Different operators on Best Test Cases.

Figure 6.2 gives comparison between the mutation operators distinctly on the basis of time. In this graph Y-axis shows the Time for all the mutation operators used, whereas X-axis shows the best test case. This graph shows the time for different test cases. We are comparing the average of time of the given mutation operators in context to best test case.

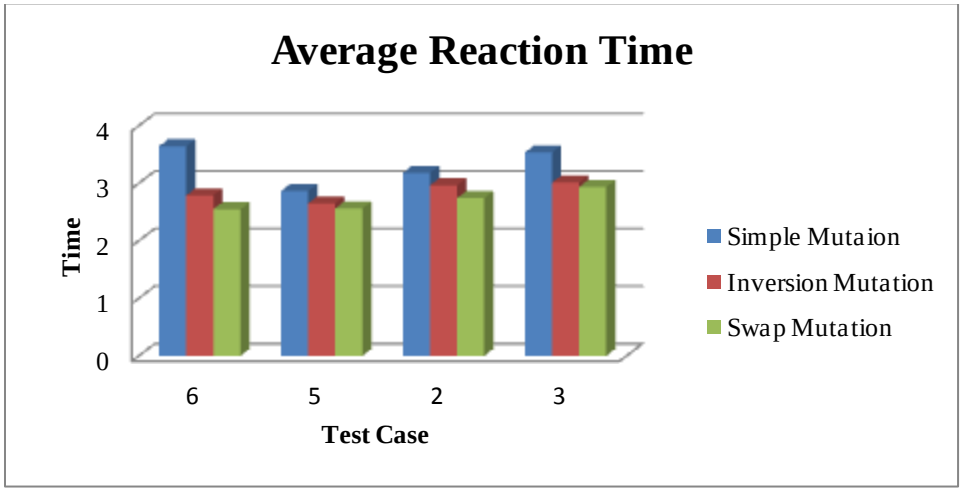


Figure 6.2 Comparison between Mutation operators on the basis of time.

The Swap Mutation operator generated better fitness values and takes less time as compared to the inversion, and simple mutation operator used in GA. Application of mutation operator on test cases resulting in smaller fitness values and less time as compared to its peers has been considered as best one, which is coming through swap mutation operator here. Now, the mutants have been injected into the best entries as obtained in the previous step.

The mutants are injected only upon the conditional statements such as if else, While loop *etc*. The injected mutants are shown in Table 6.2. We have generated the mutants for the selected test paths {2, 3, 5, 6}.

Table 6.2 Mutants generation for the best test cases.

S. No.	Best Test Case	Mutants
1	2	0>= L1 1<L2 2>=L2
2	3	0>=L1 1<L2 2>=L2
3	5	0>=L2 2>L1
4	6	1>=L1 1>=L2

The mutants given in the table have been injected into the original source code. Here, the total mutants Injected are 10. Among all the total mutants, the killed mutants are 8. So the Mutant Score is calculated and results as 80%. The mutation score considers the coverage for detecting the errors.

Chapter 7

Conclusion and Future scope

7.1 Conclusion

In our research work, we proposed an approach for the refinement of test cases having better path coverage as compare to the simple GA. The test cases have been generated for the sample source code through the path crawler tool. In our research work we identified that swap mutation operator works best as compared to inversion and the simple mutation operator. The quality was evaluated by Fitness value and mutation scores.

7.2 Future Scope

In future improved GA can be extended for applying to multithreaded program having concurrent and parallel execution.

References

[1] R. Khan and M. Amjad, "Automatic test case generation for unit software testing using genetic algorithm and mutation analysis," 2015 IEEE UP Section Conference on Electrical Computer and Electronics (UPCON), Allahabad, 2015, pp. 1-5.

[2] P. Jorgensen, Software testing: a craftsman's approach. Boca Raton: CRC Press, Taylor & Francis Group, 2014.

[3] A. P. Mathur, Foundations of software testing, PEARSON Education India, pp.502-593, 2013.

[4] B. Beizer, Software testing techniques, 2nd Edition, Van Nostrand Reinhold, 1990.

[5] A. Paul, J. Offutt, Introduction to software testing, Cambridge University Press, pp.170-212, 2008.

[6] J. Offutt, A. Lee, G. Rothermel, H. Untch, Z. Christian , "An experimental determination of sufficient mutant operators," ACM Transactions on Software Engineering, vol.5, no. 2, pp.99-118, 1996.

[7] M. Pezzè and M. Young, Software testing and analysis: process, principles, and techniques. Hoboken, NJ: Wiley, 2008.

[8] R. S. Pressman, "Software engineering: a practitioner's approach," Palgrave Macmillan, 2005.

[9] K. Singh, S. Kumar, and G. Shrivastava. "A Strategic Approach to Software Testing."

[10] D. Millo, A. Richard, J. Lipton, and G. Sayward. "Hints on test data selection: Help for the practicing programmer." vol.5, no.4, pp.34-41, 1978.

[11] J. Ofutt Y.S. Ma and Y.R. Kwon. Mujava : An automated class mutation system. Journal of Software Testing, Veri_cation and Reliability, vol.15no.2 pp.97-133, 2005.

[12] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," in *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649-678, Sept.-Oct.2011.doi: 10.1109/TSE.2010.62.

[13] M. P. Usaola and P. R. Mateo, "Mutation Testing Cost Reduction Techniques: A Survey," in *IEEE Software, vol.27, no.3, pp.80*-86, May-June2010. doi: 10.1109/MS.2010.79.

[14] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic Mutation Test Input Data Generation via Ant Colony," Proc. Ninth Ann. Conf. Genetic and Evolutionary Computation, pp. 1074-1081, 2007.

[15] "Mutation Testing in Software Development Tutorial," *TestingBrain*. [Online]. Available: http://www.testingbrain.com/whitebox/mutation-testing.html. [Accessed: 10-Feb-2017].

[16] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" Proceedings of the 27th international conference on Software engineering (ICSE '05), New York, pp. 402-411, 2005.

[17] K. Srivastava, P. Ranjan, and K.Tai-hoon . "Application of genetic algorithm in software testing." International Journal of software Engineering and its Applications vol.3 no.4, pp.87-96, 2009.

[18] W. Banzhaf, P. Nordin, R. Keller, F. Francone, "Genetic programming – An introduction", San Francisco: Morgan Kaufmann, vol. 1, 1998.

[19] A. Kaur, and S. Goyal, "A genetic algorithm for fault based regression test case prioritization," International Journal of Computer Applications, vol. 32, no. 8, 2011.

[20] G. Dhiman, and V. Kumar, "Spotted hyena optimizer: A novel bio-inspired based metaheuristic technique for engineering applications," Advances in Engineering Software, 2017.

[21] K. K. Mishra, S. Tiwari, A. Kumar and A. K. Misra, "An approach for mutation testing using elitist genetic algorithm," *3rd International Conference on Computer Science and Information Technology*, Chengdu, vol.5 , pp. 426-429, *2010*.

[22] M. Dave and R. Agrawal, "Search based techniques and mutation analysis in automatic test case generation: A survey," *IEEE International Advance Computing Conference (IACC)*, Banglore, pp. 795-799, 2015,

[23] A.J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," Mutation testing for the new century, Springer US, pp. 34-44, 2001.

[24] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," in *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 371-379, July 1982. doi: 10.1109/TSE.1982.235571.

[25] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," in *IEEE Transactions on Software Engineering*, vol.20, no.5, pp. 337-344, May 1994. doi: 10.1109/32.286422.

[26] C. Byoungju, and A. P. Mathur, "High-performance mutation testing," Journal of Systems and Software, vol.20, no.2, pp.135-152, 1993.

[27] N. Jatana, S. Rani and B. Suri, "State of art in the field of Search-based Mutation Testing,"*4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*,Noida, 2015, pp. 1-6. doi: 10.1109/ICRITO.2015.7359256.

[28] W. B. Langdon, M. Harman and Y. Jia, "Multi Objective Higher Order Mutation Testing with Genetic Programming," *Testing: Academic and Industrial Conference - Practice and Research Techniques*, Windsor, 2009, pp.21-29.doi: 10.1109/TAICPART.2009.18.

[29] H. Haga and A. Suehiro, "Automatic test case generation based on genetic algorithm and mutation analysis," *2012 IEEE International Conference on Control System, Computing and Engineering*, Penang, and pp.119-123.doi: 10.1109/ICCSCE.2012.6487127.

[30] R. Khan and M. Amjad, "Automatic test case generation for unit software testing using genetic algorithm and mutation analysis," *2015 IEEE UP Section Conference on Electrical Computer and Electronics (UPCON)*, Allahabad, 2015, pp. 1-5.doi: 10.1109/UPCON.2015.7456734.

[31] J. Louzada, C. G. Camilo-Junior, A. Vincenzi and C. Rodrigues, "An elitist evolutionary algorithm for automatically generating test data," *2012 IEEE Congress on Evolutionary Computation*, Brisbane, QLD, and pp.1-8.doi: 10.1109/CEC.2012.6256516.

[32] W. B. Langdon, M. Harman and Y. Jia, "Multi Objective Higher Order Mutation Testing with Genetic Programming," *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, Windsor, pp.21-29.doi: 10.1109/TAICPART.2009.18.

[33] D. Liu, X. Wang, and J. Wang. "Automatic test case generation based on genetic algorithm," Journal of Theoretical & Applied Information Technology vol.48, no.1, 2013.

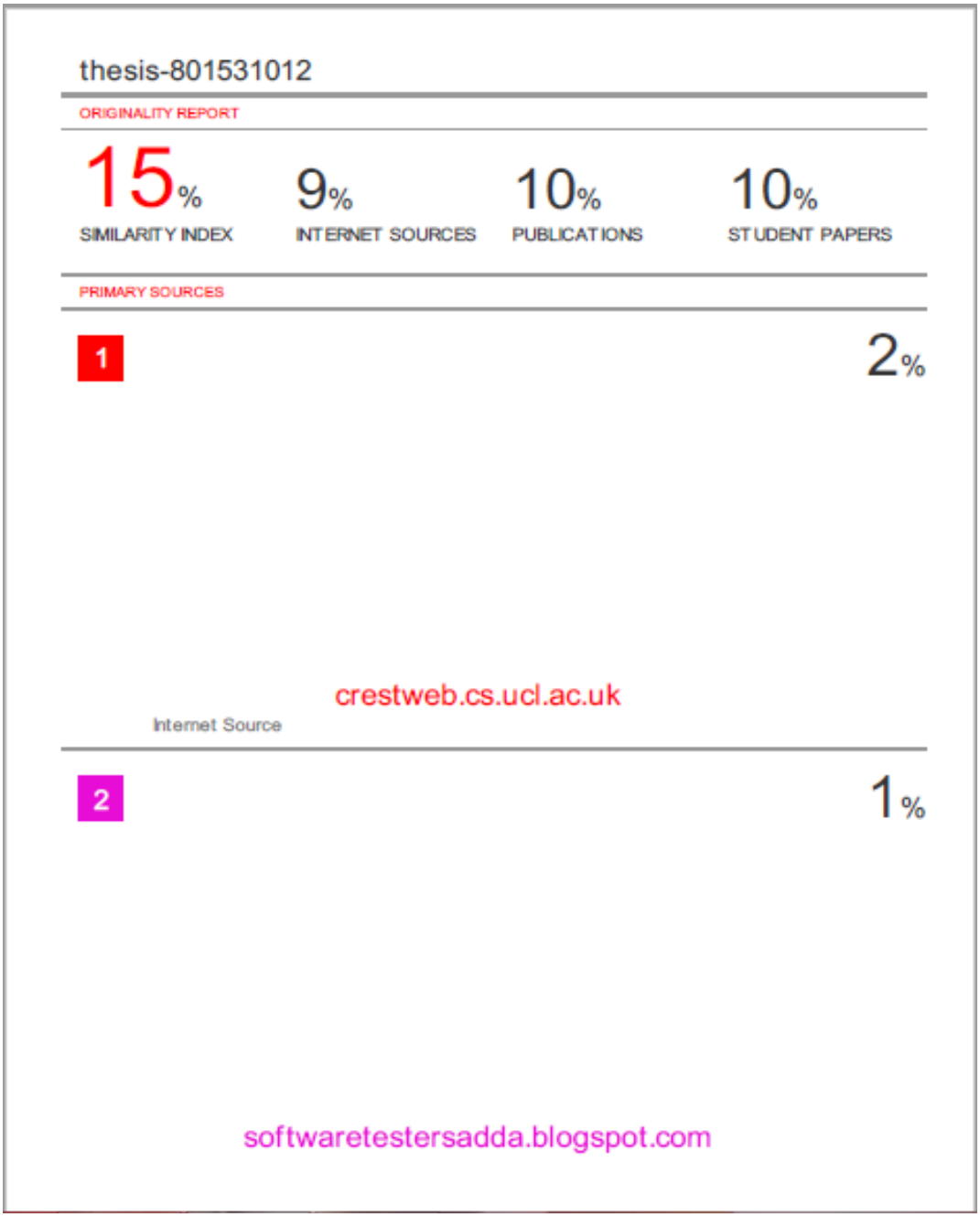
[34] G. Fraser, and A. Arcuri. "Achieving scalable mutation-based generation of whole test suites," Empirical Software Engineering, vol.20, no.3, pp.783-812, 2015.

[35] Y. Jia and M. Harman, "Constructing Subtle Faults Using Higher Order Mutation Testing," *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, Beijing, 2008, pp.249-258. doi: 10.1109/SCAM. 2008.36.

[36] D. J. Mala, S.R. Elizabeth, and V. Mohan, "Intelligent Test Case Optimizer-An automated Hybrid Genetic Algorithm based test case optimization framework," International Journal of Computer Science and Applications, vol.1 no.1, pp.51-55, 2008.

[37] G. Fraser and A. Zeller, "Mutation-Driven Generation of Unit Tests and Oracles," in *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278-292, March-April 2012.doi: 10.1109/TSE.2011.93.

Plagiarism Report



thesis-801531012

ORIGINALITY REPORT

15%	9%	10%	10%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	crestweb.cs.ucl.ac.uk Internet Source	2%
2	softwaretestersadda.blogspot.com	1%