Minimizing and Optimizing the Solution Space of Test Data

A MAJOR PROJECT-II THESIS REPORT

SUBMITTED IN PARTIAL FULFILMENT OF REQUIREMENT
FOR THE AWARD OF THE DEGREE
OF

MASTER OF TECHNOLOGY
IN
SOFTWARE ENGINEERING

Submitted By – **Trilochan Yadav**
(Roll No. 2K18/SWE/16)
Under the supervision of **Dr. Ruchika Malhotra (Associate Professor)**
Delhi Technological University

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering) Bawana Road, Delhi-110042

June, 2020

DECLARATION

I hereby declare that the Major Project-II work entitled "**Minimizing and Optimizing the Solution Space of Test Data**" which is being submitted to Delhi Technological University, in partial fulfillment of requirements for the award of the degree of Master of Technology (Computer Science and Engineering) is a bona fide report of Major Project-II carried out by me. I have not submitted the matter embodied in this dissertation for the award of any other degree or diploma

Place: Delhi

Trilochan Yadav **2K18/SWE/16**

CERTIFICATE

This is to certify that the Major II Report entitled “**Minimizing and Optimizing the Solution Space of Test Data"** submitted by **Trilochan Yadav** (2K18/SWE/16) for the partial fulfilment of the requirement for the award of degree of Master of Technology (Software Engineering) is a record of the original work carried out by him under my supervision.

Place: Delhi

Date:

Project Guide Dr. Ruchika Malhotra Associate Professor

Department of Computer Science & Engineering

Delhi Technological University

ACKNOWLEDGEMENT

First of all, I would like to express my deep sense of respect and gratitude to my project supervisor **Dr. Ruchika Malhotra** for providing the opportunity of carrying out this project and being the guiding force behind this work. I am deeply indebted to him for the support, advice and encouragement he provided without which the project could not have been a success.

Secondly, I am grateful to Dr. Rajni Jindal, HOD, Computer Science & Engineering Department, DTU for her immense support. I would also like to acknowledge Delhi Technological University library and staff for providing the right academic resources and environment for this work to be carried out.

Last but not the least I would like to express sincere gratitude to my parents and friends for constantly encouraging me during the completion of work.

Trilochan Yadav

Roll No – 2K17/SWE/19

M. Tech (Software Engineering) Delhi Technological University

ABSTRACT

In the software development lifecycle (SDL), testing of software is the most stressful and exhausting operation which consumes lots of time. Every aspect of software is very hard to test. Consequently, in recent times some automatic data generation research methods were added to reduce the time expended during the software testing. And the solution space of the automated generated test data is very large. It is not easy to check all the test data which is generated because it is time consuming, forces to check whole solution space of automated generated test data. We present in this paper demonstrating the design framework, implementing it and discovering the tool 's capabilities to minimize the test data generated. Our concrete concepts on the test cases for the optimal set is based on the mutation function Specified by the user. The system was implemented in language C++. We introduce mutation function to calculate mutant score with value and path to the test cases generated to minimize the solution space for the tester.

Keywords: SUT (Software Under Testing), Mutant Score, Test Cases, Optimized Test Cases, Optimal Test Suites.

TABLE OF CONTENTS

CONTENT	PAGE NUMBER
Title Page	1
Declaration	2
Certificate	3
Acknowledgment	4
Abstract	5
Table of Contents	6
List of Figures	8
List of Tables	9
List of Abbreviations and Nomenclature	10
1. Introduction 1.1 Overview and Motivation 1.2 Research Objective 1.3 Mutation Testing	11 11 11 12
2. Literature study	13
3. Tool Framework 3.1 Overview 3.2 Formula for Mutant Score (1) 3.3 Formula for Mutant Score (2)	16 16 18 18

4. Resulting and Comparing	20
5. Discussion and Analyze	44
5.1 Change in test suites order.	45
5.2 Change in test cases order within a test suite.	50
5.3 Dividing the Algorithm	56
5.3.1 Algorithm for optimal test cases	57
5.3.2 Algorithm for path coverage of SUT in each test suite	57
6. Conclusion	58
7. References	59

LIST OF FIGURES

CONTENT	PAGE NUMBER
Fig 3.1 Block diagram of Tool Developed	17
Fig 4.1 Program graph (Path)	23

LIST OF TABLES

CONTENT	PAGE NUMBER
Table 4.1 Mutated Statements	23
Table 4.2 Comparing Mutation Score	42
Table 4.3 Other Result Table	43
Table 5.1 Comparing Mutation Score	56

List of Abbreviations and Nomenclature

- SUT – Software Under Testing
- OP – Original code Path
- EO – Excepted Output
- OO – Obtained Output
- TP – Total number of paths in SUT.
- MS (1) - Mutant Score (1)
- MS (2) - Mutant Score (2)

CHAPTER 1

INTRODUCTION

1.1 Overview and Motivation

One of most complicated tasks throughout software development is software testing, and absorbs great amount of time and effort while developing. Testing the software without planning, the software quality can become progressively worst, or extend the software's time and cost. One way to cut down on time is to automate test case generation [1]. The problem with automate generation of test cases is that the solution space of test data generated, which is very large in size. To find the minimum and optimal test suites in the solution space is very important as minimized solution space take less time to check the software correctness. By minimizing the test data, we save the testing cost and time.

1.2 Research Objective

We propose an alternative solution in this paper to minimizing the test data solution space, by referring to the mutation score for each test case of the test suits, based on the actual execution of each path of the program under testing, analysis the flow of each test case. The new test suits are developed using actual input variable values. When executing the program on some input data the execution flow of the path in the program is monitoring. While execution of program, the flow of path is observed and eliminating the test case when the test case is alive and continues when the test case gets killed by the mutant [2]. We select all the test cases with mutant score equal to 1 and test case path is not selected previously. In this approach, we helped the tester to get the optimal solution space.

1.3 Mutation Testing

Mutation testing involves smaller modifications of a program. The goal is to help the tester establish successful tests or identify vulnerabilities in the test data used for the program or in parts of the code which are rarely or never accessed during execution. Mutation testing is a form of white-box testing which is mainly used for unit testing. It is a powerful approach to attain high coverage of the source program.

Following are the advantages of Mutation Testing:

- This testing is capable comprehensively testing the mutant program.
- Mutation testing brings a good level of error detection to the software developer.
- This method uncovers 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 a most reliable and stable system.

This is the method which checks for the effectiveness and accuracy of a testing program to detect the faults or errors in the system. And we used mutation to minimize and optimize the test suites.

CHAPTER 2

LITERATURE REVIEW

In this section we present the problem of Optimal Automatic Test Data Generator comes under the research topic of Software testing. It is noted that even after using specified method, the performance of the test data that are generated by Automatic Test Data Generator is not so optimal. So, some researchers performed different techniques to optimize the solution space of test data.

The test data dependents on two things, which type of tool we are using and what Minimization function we are applying on it. There isn't enough research and literature available or conducted on this topic in order to improve the test data generated solution space. The paper also talks about some of the important conclusions that have not been researched yet and some of them are.

The paper by Michael et al., developed a new test data generation software system which could be used for combinatory optimisation in order to obtain condition decision coverage of the program which is developed in C++ or C language [3]. System generated the test knowledge in projects with more complicated interrelationship systems, and the tests showed a major gap between another feature for minimizing problems and the production test results which are dynamic.

The paper by Pargas et al., implemented a Genetic Algorithm (GA) check data generation method for different reporting strategies for example (statement coverage and branch coverage) [4]. The tool employed parallel processing to improve the overall search performance. Another random generator of test data was developed to equate the effects of the Preferred method with

random generator, that revealed the existing one outmatch the latter, as the system complication grew.

The paper by Bin et al., represented the automated test data generation tool that dynamically generated test data by reducing the test data generation process to a single minimisation function [5]. In the proposed algorithm some improvements have been made to improve the computational efficiency of the proposed method.

The paper by Mala, Automated test suite optimisation by applying the honey bees' parallel behavior allowing the algorithm to find the optimal global solution faster [6]. Comparison of the result of the suggested method with other methods, as with GA, sequential ABC and random testing. The test verified that the parallel ABC is proposed worked superior than any of the other methods.

Mala et al. analyzed high-mutation test cases and an updated GA were added to boost the test suite by covering every test case [7]. The suggested method contained two better heuristics that are intake to examine the performance of the test cases produced from the suggested method to the preestablished method and experiment finding revealed that the suggested method yielded higher performing test cases than developed methods.

The paper by Malhotra et al., integrated mutation analysis to create test cases in order to refine the produced test cases by introducing mutation at the time of test data creation, instead of stopping it after creation [8]. To generating the test cases GA is used. The findings of the experiment show that the time taken by the proposed method was substantially saved and the number of test cases created was dramatically reduced.

The paper by Malhotra et al., Using the SBT, developed a test data generation tool. The developed tool is better than the other tools which are designed for the production of test data, because it enables the production of test data using several SBT and allows the user to evaluate

the output of the input information under test (SUT) [9]. This work would decrease the cost and time of testing for a given SUT.

The paper by Ruilian Zhao et al, indicates the DOTSG approach for the EFSM design from GA. The solution involved the creation from GA of test data and of test paths generation. The study took the opinion that the diversity between test paths has a greater effect on disparity among test paths had a greater effect on test case suite diversity in comparison to the disparity among test data in EFSM designs [10].

Fahad M. Almansour et al, represents an evidenced based comparative study with genetic algorithms & adaptive random methods based on four parameters: reduced test suite size, convergence rate, the efficiency of maximizing of all du-pair by coverage ratio criteria and elapsed time. Through these findings, stated that GA algorithms are much more efficient from ORT and ART in the testing of data-flow, and somehow ART has achieved much more coverage ratio [12]. Hence, suggested that the hybridization of ART and GA and the use of a hybrid framework in testing the data-flow.

CHAPTER 3

TOOL FRAMEWORK

3.1 Overview

The tool we had develop is a language-oriented, general purpose programming method integrating some SUT data. The tool 's framework is indicated in Figure 3.1. The first step is to choose the SUT solution space that needs to reduce the test data for. This is done by the SUT Solution Space module. Then we select the test suites from which we have to select the optimal test case.

The Domain Initialiser initializes all the algorithm-related values and produces the SUT control flow graph. The path evaluator initialises the test cases of the 1st test suite and tests the direction that every test case is going along. The test cases where there's no path among the beginning node and every leaving node are considered negligible [9].

Then the tool is instructive the Mutation Calculator (1), for calculating the mutant score for every test case based on the killed mutant by the total mutant. Then the tool is instructive the Mutation Calculator (2), for calculating the mutant score of the same test case again based on the killed mutant by the total mutant. We select only those test cases with both the mutant score equal to 1.

Then, it employs the redundancy checker to determine a single test case 's path that wasn't cover by any other test cases earlier. These test cases are unique as other cases are considered repetitive test cases. Consequently, test cases which are unique added to the optimum test data. And then check the conditions and select the new test suites.

On further refining the test cases and generating new test suite, the Path Checker checks whether the tester's defined total paths are equal to the paths generated, the algorithm stops. The current test case set will then form the final test suite. The main Algorithm stop when all Automated test data checked.

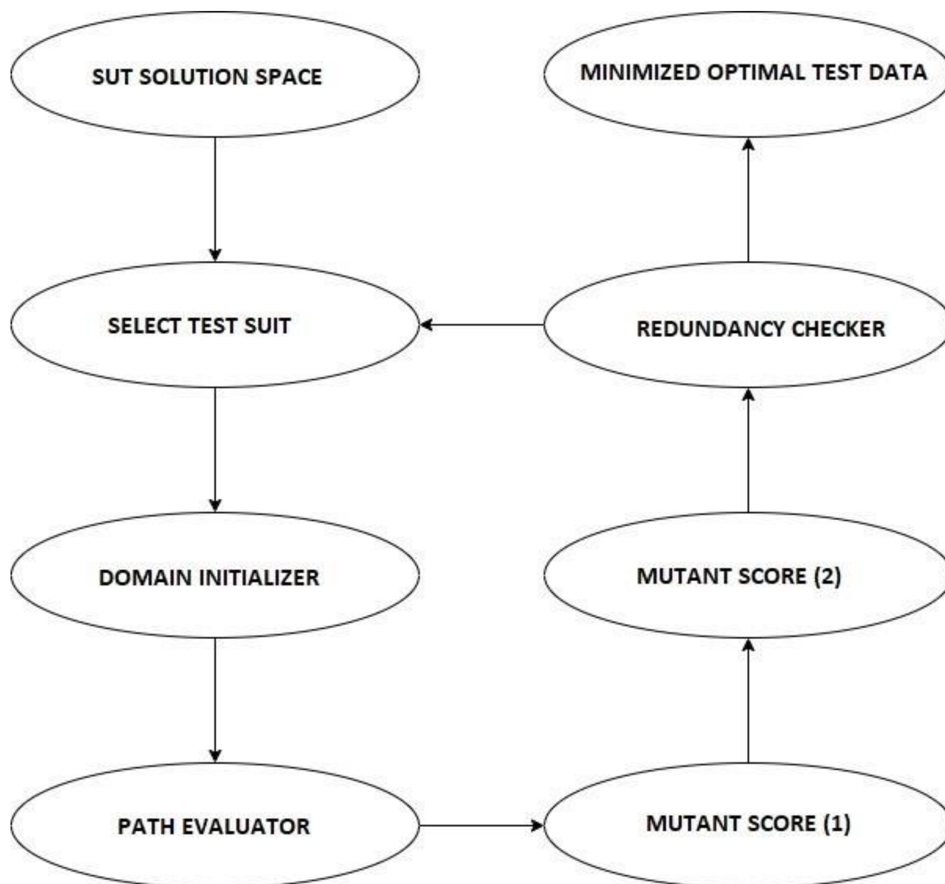

Fig 3.1 Block diagram of Tool Developed

3.2 Formula for Mutant Score (1)

Mutant score (1) is calculated for conditional changes in the mutant code.

Line number in mutant code where change is occurred = L

The original code path of the test case = OP

The Mutant code path of the test case = MP

The expected output of the test case = EO

The obtain output of the test case = OO

So,

MS (1) = (L+1 belongs to OP) && (OP equal to MP)

Mutant Alive = MS (1) is True.

Mutant Kill = MS (1) is False.

Total Mutant = Mutant Alive + Mutant kill

Mutant Score (1) = Mutant Kill / Total Mutant

3.3 Formula for Mutant Score (2)

Mutant score (2) is calculated for operation changes in the mutant code.
Line number in mutant code where change is occurred = L

The original code path of the test case = OP

The Mutant code path of the test case = MP

The expected output of the test case = EO

The obtain output of the test case = OO

So,

MS (2) = (L belongs to OP) && (OP equal to MP) && (OO equal to EO).

Mutant Alive = MS (2) is True.

Mutant Kill = MS (2) is False.

Total Mutant = Mutant Alive + Mutant kill

Mutant Score (2) = Mutant Kill / Total Mutant

To add the test case to the optimal Test data both the Mutant score (1) and Mutant Score (2) must be equal to 1.

CHAPTER 4

RESULT AND COMPARISON

Now we run the tool and analyze the result of the optimal test data. We minimize the solution space of the SUT (Software under Test). The important part in the tool is to calculate both the mutant score. Mutation score is the main key in minimization tool. This helps us to minimize the solution space and lead us to the optimal test data. We calculate the both mutant score and if they equal to 1 then we select them as the part of optimal test data. Then we analyze the mutant score of test suites and then compare it with the mutant score of the obtain test suite. If the mutation score of the obtain test suite is better, than we consider that test suite as the optimal test suite for the test data.

We are performing the tool on the program to find the largest among three number.

Program to find the largest among three numbers

```
#include<stdio.h>

#include<conio.h>

1. void main()

2. {

3. float A,B,C;

4. clrscr();

5. printf("Enter number 1:\n");

6. scanf("%f", &A);

7. printf("Enter number 2:\n");
```

```
8. scanf("%f", &B);
9. printf("Enter number 3:\n");
10. scanf("%f", &C); /*Check for greatest of three numbers*/
11. if(A>B) {
12. if(A>C) {
13. printf("The largest number is: %f\n",A);
14. }
15. else  {
16. printf("The largest number is: %f\n",C);
17. }
18. }
19. else {
20. if(C>B) {
21. printf("The largest number is: %f\n",C);
22. }
23. else  {
24. printf("The largest number is: %f\n",B);
25. }
26. }
27. getch();
28. }
```

We have test suite of the SUT we perform over algorithm and see how it minimize and give an optimal solution.

Test Suite A.

S. No.	A	B	C	Expected Output
1	6	10	2	10
2	10	6	2	10
3	6	2	10	10
4	6	10	20	20

Test Suite B.

S. No.	A	B	C	Expected Output
1	10	10	10	10
2	10	5	5	10
3	6	10	5	10
4	20	40	30	40

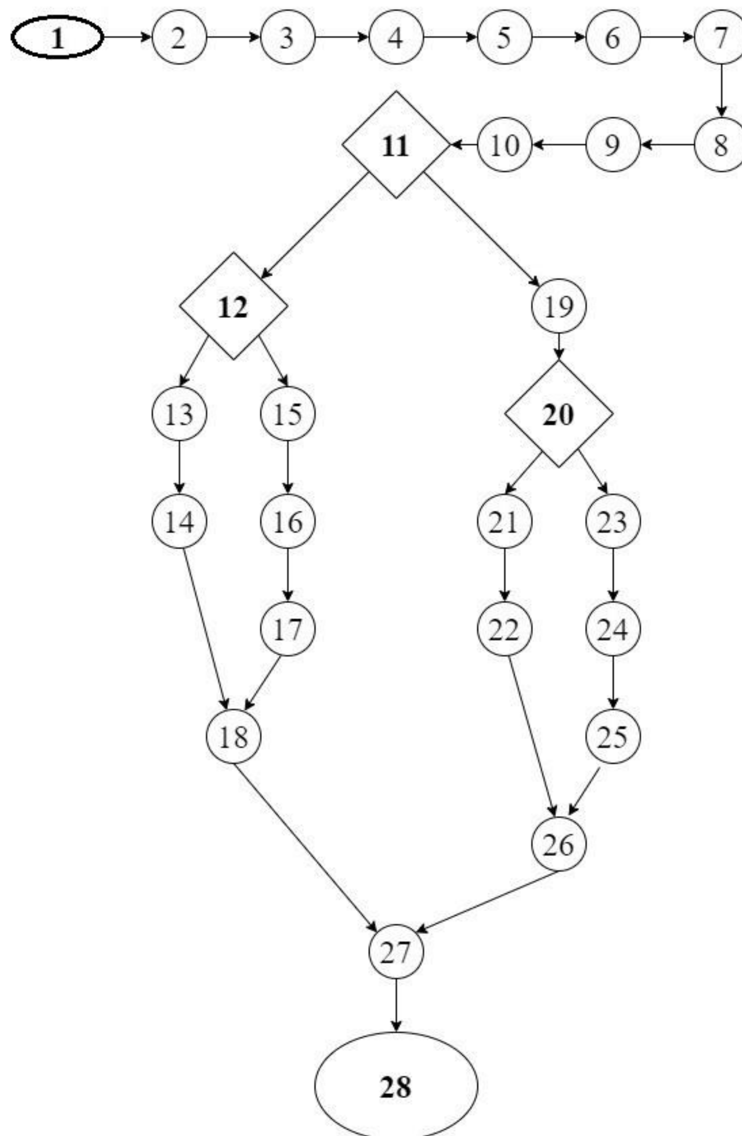

Fig 4.1 Program graph to find the largest number amongst three numbers (Path)

Mutant No.	**Line No.**	**Original Line**	**Modified Line**
M1	11	If(A>B)	If(A<B)
M2	11	If(A>B)	If(A>(B+C))
M3	12	If(A>C)	If(A<C)
M4	20	If(C>B)	If(C==B)
M5	16	Print (“The Largest number is:%f\n”,C);	Print (“The Largest number is:%f\n”,B);

Table 4.1 Mutated Statements

Mutant live = Expected Output is equal to Obtain Output.

Mutant kill = Expected Output is not equal to Obtain Output.

For Test Suite A.

Mutant M1.

S. No.	A	B	C	EO	OO
1	6	10	2	10	6
2	10	6	2	10	6
3	6	2	10	10	10
4	6	10	20	20	20

Mutant M1 is killed

Mutant M2.

S. No.	A	B	C	EO	OO
1	6	10	2	10	10
2	10	6	2	10	10
3	6	2	10	10	10
4	6	10	20	20	20

Mutant M2 is alive

Mutant M3.

S. No.	A	B	C	EO	OO
1	6	10	2	10	6
2	10	6	2	10	2
3	6	2	10	10	6
4	6	10	20	20	20

Mutant M3 is killed

Mutant M4.

S. No.	A	B	C	EO	OO
1	6	10	2	10	10
2	10	6	2	10	10
3	6	2	10	10	10
4	6	10	20	20	10

Mutant M4 is killed

Mutant M5.

S. No.	A	B	C	EO	OO
1	6	10	2	10	10
2	10	6	2	10	10
3	6	2	10	10	2
4	6	10	20	20	20

Mutant M5 is killed

For Test Suite A

Mutation Score = Number of mutants killed/ Total number of mutants.

Mutant killed = 4.

Total Mutant is 5.

Mutant live = 1 (M2).

So,

Mutation Score for Test Suite A is **0.8**

For Test Suite B.

Mutant M1.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	10
3	6	10	5	10	6
4	20	40	30	40	30

Mutant M1 is killed

Mutant M2.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	5
3	6	10	5	10	10
4	20	40	30	40	40

Mutant M2 is killed

Mutant M3.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	5
3	6	10	5	10	10
4	20	40	30	40	40

Mutant M3 is killed

Mutant M4.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	10
3	6	10	5	10	10
4	20	40	30	40	40

Mutant M4 is alive

Mutant M5.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	10
3	6	10	5	10	10
4	20	40	30	40	40

Mutant M5 is alive

For Test Suite B

Mutation Score = Number of mutants killed/ Total number of mutants.

Mutant killed = 3.

Total Mutant is 5.

Mutant live = 2 (M4, M5).

So,

Mutation Score for Test Suite A is **0.6**

As we can see the Mutant score is not equal to 1, which means the test suites are not optimal but test suite A is near to optimal.

Now we apply our algorithm to improve the test suite and minimize the optimal solution. We use our new Mutant Score formula to obtain the optimal solution space by minimizing and optimizing the test suites.

Our Mutant Score must be equal to 1 of each test cases to add in minimized optimal solution space. So, our method selects the optimal test case from test suite and gets the optimal and test suites.

Let now we start our method of selecting the test case from the given test suite.

For Test Suite A.

Mutant score (1) is calculated for conditional changes in the mutant code.

Line number in mutant code where change is occurred = L

The original code path of the test case = OP

The Mutant code path of the test case = MP

So,

MS (1) = (L+1 belongs to OP) && (OP equal to MP)

Mutant Alive = MS (1) is True.

Mutant Kill = MS (1) is False.

Total Mutant = Mutant Alive + Mutant kill

Mutant Score (1) = Mutant Kill / Total Mutant

Mutant M1

line no. 11

L+1=12.

S. No.	A	B	C	OP	L+1 ε OP	OP=MP	Mutant
1	6	10	2	1-11, 19-20, 23-28	False	-	kill
2	10	6	2	1-14, 18, 27 ,28	True	False	kill
3	6	2	10	1-12, 15-18, 27,28	True	False	kill
4	6	10	20	1-11, 19-22, 26-28	False	-	kill

Mutant M2

line no. 11

L+1=12.

S. No.	A	B	C	OP	L+1 ε OP	OP=MP	Mutant
1	6	10	2	1-11, 19-20, 23-28	False	-	kill
2	10	6	2	1-14, 18, 27 ,28	True	True	alive
3	6	2	10	1-12, 15-18, 27,28	True	False	kill
4	6	10	20	1-11, 19-22, 26-28	False	-	kill

2nd number test case is alive. So, it must not be in optimal test suites.

Mutant M3

line no. 12

L+1=13.

S. No.	A	B	C	OP	L+1 ε OP	OP=MP	Mutant
1	6	10	2	1-11, 19-20, 23-28	False	-	kill
2	10	6	2	1-14, 18, 27 ,28	True	False	alive
3	6	2	10	1-12, 15-18, 27,28	False	-	kill
4	6	10	20	1-11, 19-22, 26-28	False	-	kill

Mutant M4

line no. 20

L+1=21.

S. No.	A	B	C	OP	L+1 ε OP	OP=MP	Mutant
1	6	10	2	1-11, 19-20, 23-28	False	-	kill
2	10	6	2	1-14, 18, 27 ,28	False	-	alive
3	6	2	10	1-12, 15-18, 27,28	False	-	kill
4	6	10	20	1-11, 19-22, 26-28	True	False	Kill

As we calculated the mutant score (1) and Now we calculate Mutant Score (2)

Mutant score (2) is calculated for operation changes in the mutant code.

Line number in mutant code where change is occurred = L

The original code path of the test case = OP

The Mutant code path of the test case = MP

The expected output of the test case = EO

The obtain output of the test case = OO

So,

MS (2) = (L belongs to OP) && (OP equal to MP) && (OO equal to EO).

Mutant Alive = MS (2) is True.

Mutant Kill = MS (2) is False.

Total Mutant = Mutant Alive + Mutant kill

Mutant Score (2) = Mutant Kill / Total Mutant

Mutant M5

line no. 16

L=16.

S. No.	A	B	C	OP	L ε OP	OP=MP	OO=EO	Mutant
1	6	10	2	1-11, 19-20, 23-28	False	-	-	kill
2	10	6	2	1-14, 18, 27 ,28	False	-	-	kill
3	6	2	10	1-12, 15-18, 27,28	True	True	False	kill
4	6	10	20	1-11, 19-22, 26-28	False	-	-	kill

We calculated both the Mutant Score (1) and Mutant Score (2).

The Mutant Score Table.

Test Case No.	Mutant Score (1)	Mutant Score (2)	OP ε Path	Selected
1	**1**	**1**	**No**	**Yes**
2	0.8	1	No	No
3	**1**	**1**	**No**	**Yes**
4	**1**	**1**	**No**	**Yes**

Path is the set of paths of test cases which are previously selected for the present optimal test suites.

We select those test cases for minimized optimal solution which have both mutant score equal to 1 and OP ε Path is no.

For Test Suite B.

Mutant score (1) is calculated for conditional changes in the mutant code.

Line number in mutant code where change is occurred = L

The original code path of the test case = OP

The Mutant code path of the test case = MP

So,

MS (1) = (L+1 belongs to OP) && (OP equal to MP)

Mutant Alive = MS (1) is True.

Mutant Kill = MS (1) is False.

Total Mutant = Mutant Alive + Mutant kill

Mutant Score (1) = Mutant Kill / Total Mutant

Mutant M1

line no. 11

L+1=12.

S. No.	A	B	C	OP	L+1 ε OP	OP=MP	Mutant
1	10	10	10	1-11, 19-20, 23-28	False	-	kill
2	10	5	5	1-14, 18, 27 ,28	True	False	alive
3	6	10	5	1-11, 19-20, 23-28	False	-	kill
4	20	40	30	1-11, 19-20, 23-28	False	-	kill

Mutant M2

line no. 11

L+1=12.

S. No.	A	B	C	OP	L+1 ε OP	OP=MP	Mutant
1	10	10	10	1-11, 19-20, 23-28	False	-	kill
2	10	5	5	1-14, 18, 27 ,28	True	False	alive
3	6	10	5	1-11, 19-20, 23-28	False	-	kill
4	20	40	30	1-11, 19-20, 23-28	False	-	kill

Mutant M3

line no. 12

L+1=13.

S. No.	A	B	C	OP	L+1 ε OP	OP=MP	Mutant
1	10	10	10	1-11, 19-20, 23-28	False	-	kill
2	10	5	5	1-14, 18, 27 ,28	True	False	alive
3	6	10	5	1-11, 19-20, 23-28	False	-	kill
4	20	40	30	1-11, 19-20, 23-28	False	-	kill

Mutant M4
line no. 20

L+1=21.

S. No.	A	B	C	OP	L+1 ε OP	OP=MP	Mutant
1	10	10	10	1-11, 19-20, 23-28	False	-	kill
2	10	5	5	1-14, 18, 27 ,28	False	-	alive
3	6	10	5	1-11, 19-20, 23-28	False	-	kill
4	20	40	30	1-11, 19-20, 23-28	False	-	kill

As we calculated the mutant score (1) and Now we calculate Mutant Score (2)

Mutant score (2) is calculated for operation changes in the mutant code.

Line number in mutant code where change is occurred = L

The original code path of the test case = OP

The Mutant code path of the test case = MP

The expected output of the test case = EO

The obtain output of the test case = OO

So,

MS (2) = (L belongs to OP) && (OP equal to MP) && (OO equal to EO).

Mutant Alive = MS (2) is True.

Mutant Kill = MS (2) is False.

Total Mutant = Mutant Alive + Mutant kill

Mutant Score (2) = Mutant Kill / Total Mutant

Mutant M5

line no. 16

L=16.

S. No.	A	B	C	OP	L ε OP	OP=MP	OO=EO	Mutant
1	10	10	10	1-11, 19-20, 23-28	False	-	-	kill
2	10	5	5	1-14, 18, 27 ,28	False	-	-	kill
3	6	10	5	1-11, 19-20, 23-28	True	True	False	kill
4	20	40	30	1-11, 19-20, 23-28	False	-	-	kill

We calculated both the Mutant Score (1) and Mutant Score (2).

The Mutant Score Table.

Test Case No.	Mutant Score (1)	Mutant Score (2)	OP ε Path	Selected
1	1	1	Yes	No
2	**1**	**1**	**No**	**Yes**
3	1	1	Yes	No
4	1	1	Yes	No

Path is the set of paths of test cases which are previously selected for the present optimal test suites.

We select those test cases for minimized optimal solution which have both mutant score equal to 1 and OP ε Path is no.

From Both Test Suites we selected optimal test cases for new test suites. Now we again calculate the mutation score of new test suite and compare the mutation score with old test suites.

Our new test suite selected 4 test cases from test suit A and one test case from test suite B.

3+1=4 test cases are selected. Number of test cases selected is equal to the TP. So, these selected test cases form our new test suite.

New Test Suite

S. No.	A	B	C	Expected Output
1	6	10	2	10
2	6	2	10	10
3	6	10	20	20
4	10	10	5	10

We calculate the mutation score of new test suite and see is this test suit is better than other or not.

For New Test Suite.

Mutant M1.

S. No.	A	B	C	EO	OO
1	6	10	2	10	6
2	6	2	10	10	10
3	6	10	20	20	20
4	10	10	5	10	5

Mutant M1 is killed

Mutant M2.

S. No.	A	B	C	EO	OO
1	6	10	2	10	10
2	6	2	10	10	10
3	6	10	20	20	20
4	10	10	5	10	5

Mutant M2 is killed

Mutant M3.

S. No.	A	B	C	EO	OO
1	6	10	2	10	10
2	6	2	10	10	6
3	6	10	20	20	20
4	10	10	5	10	5

Mutant M3 is killed

Mutant M4.

S. No.	A	B	C	EO	OO
1	6	10	2	10	10
2	6	2	10	10	10
3	6	10	20	20	10
4	10	10	5	10	10

Mutant M4 is killed

Mutant M5.

S. No.	A	B	C	EO	OO
1	6	10	2	10	10
2	6	2	10	10	2
3	6	10	20	20	20
4	10	10	5	10	10

Mutant M5 is killed

For New Test Suite

Mutation Score = Number of mutants killed/ Total mutants.
Mutant killed = 5.

Total Mutant is 5.

Mutant live = 0

So, Mutation Score for Test Suite A is **1.**

Compare test suites Mutation Score

S. No.	Test Suit Name	Mutation Score	Optimality
1	Test Suit A	0.8	Nearly Optimal
2	Test Suit B	0.6	Not Optimal
3	New Test Suit	1	Optimal

Table 4.2 Comparing Mutation Score

As we can see that the mutation score of new test suite is better than the other two test suites and from two test suites, we minimize to one test suite. So, we can say that our new algorithm of optimizing and minimizing the solution space of test data and performance are better than others and it help the tester to test the SUT in low cost and less time.

Other Result Table

AGTD = Automated Generated Test Data

TTS = Total Test Suites

MTC = Minimal Test Cases

MTS = Minimal Test Suites

OMTS = Optimal Minimal Test Suites

Program Name	AGTD	TTS	MTC	MTS	OMTS
Max no.	100	**25**	20	**5**	**5**
Max no.	200	**50**	48	**12**	**12**
Marks Grade	100	**20**	10	**2**	**2**
Marks Grade	200	**40**	15	**3**	**3**

Table 4.3 Other Result Table

As we can see in all Automated Generated Test Data is reduced or say minimized to less test data for testing. By reducing the Automated Generated Test Data, we save the testing time and cost of the software under test (SUT).

From the result we can see that Automated Generated Test Data has less average of optimal test suites than from our minimal test suites, as all the minimal test suites are optimal means minimal test data is better than the Automated Generated Test Data.

Therefore, we can say that our algorithm generates Minimal and Optimal Test data from the Automated Generated Test Data.

CHAPTER 5

DISCUSSION AND ANALYZE

In this section, we discuss and examine the majority of aspects that affect our algorithm and test data.

Main Algorithm

```
//Calculate Mutant Score 1
If (! ((L+1 ε OP) && (OP=MP)) )
{
   flag=1; // for the calculation of mutant score 2
}
//Calculate Mutant Score 2
If (flag==1)
{
    If (! ((L ε OP) && (OP=MP) && (OO=EO)) )
    {
         If (! (OP ε Path)) //redundance check
         {
              //Add Test Case to Optimal Test Data
         }
    }
 }
```

5.1 Change in test suites order.

We have seen the result of test suites A and B, now we see, if we change the order of test suites in our solution space.

Let test suite B is before test suite A. Now we apply our Algorithm, then our new test suit.

For test suite B

We calculated both the Mutant Score (1) and Mutant Score (2).

The Mutant Score Table.

Test Case No.	Mutant Score (1)	Mutant Score (2)	OP ε Path	Selected
1	**1**	**1**	**No**	**Yes**
2	**1**	**1**	**No**	**Yes**
3	1	1	Yes	No
4	1	1	Yes	No

We select those test cases for minimized optimal solution which have both mutant score equal to 1 and OP ε Path is no.

For test suite A

We calculated both the Mutant Score (1) and Mutant Score (2).

The Mutant Score Table.

Test Case No.	Mutant Score (1)	Mutant Score (2)	OP ε Path	Selected
1	1	1	Yes	No
2	0.8	1	No	No
3	**1**	**1**	**No**	**Yes**
4	**1**	**1**	**No**	**Yes**

Path is the set of paths of test cases which are previously selected for the present optimal test suites.

We select those test cases for minimized optimal solution which have both mutant score equal to 1 and OP ε Path is no.

From Both Test Suites we selected optimal test cases for new test suite 1. Now we again calculate the mutation score of new test suite and compare the mutation score with old test suites.

New Test Suite 1

S. No.	A	B	C	Expected Output
1	10	10	10	10
2	10	5	5	10
3	6	2	10	10
4	6	10	20	20

We calculate the mutation score of new test suite 1.

For New Test Suite.

Mutant M1.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	5
3	6	2	10	10	10
4	6	10	20	20	20

Mutant M1 is killed

Mutant M2.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	5
3	6	2	10	10	10
4	6	10	20	20	20

Mutant M2 is killed

Mutant M3.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	5
3	6	2	10	10	6
4	6	10	20	20	20

Mutant M3 is killed

Mutant M4.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	10
3	6	2	10	10	10
4	6	10	20	20	10

Mutant M4 is killed

Mutant M5.

S. No.	A	B	C	EO	OO
1	10	10	10	10	10
2	10	5	5	10	10
3	6	2	10	10	2
4	6	10	20	20	20

Mutant M5 is killed

Mutation Score = Number of mutants killed/ Total number of mutants.

Total mutant = 5

Mutant kill = 5

Mutant alive =0

So,

Mutation Score for New Test Suite 1 is **1.**

As we can see that the new test suite 1 is also optimal with mutation score 1, so we can say that our algorithm is independent of test suites order in aspect of obtaining the optimal test suits. But we observe that there is change in test cases of new test suite 1. As in new test suite 1, test case (6, 10, 2) which is in new test suite is replaced by test case (10 ,10, 10). Therefore, test cases in optimal test suits are dependent of test suites order.

5.2 Change in test cases order within a test suite.

We have now changed the order of test cases in test suite B (test case 1 to test case 4) and test suite B comes before test suite A. Now we apply our Algorithm, then our new test suit.

For test suite B

We calculated both the Mutant Score (1) and Mutant Score (2).

The Mutant Score Table.

Test Case No.	Mutant Score (1)	Mutant Score (2)	OP ε Path	Selected
4	**1**	**1**	**No**	**Yes**
2	**1**	**1**	**No**	**Yes**
3	1	1	Yes	No
1	1	1	Yes	No

We select those test cases for minimized optimal solution which have both mutant score equal to 1 and OP ε Path is no.

For test suite A

We calculated both the Mutant Score (1) and Mutant Score (2).

The Mutant Score Table.

Test Case No.	Mutant Score (1)	Mutant Score (2)	OP ε Path	Selected
1	1	1	Yes	No
2	0.8	1	No	No
3	**1**	**1**	**No**	**Yes**
4	**1**	**1**	**No**	**Yes**

Path is the set of paths of test cases which are previously selected for the present optimal test suites.

We select those test cases for minimized optimal solution which have both mutant score equal to 1 and OP ε Path is no.

From Both Test Suites we selected optimal test cases for new test suite 1. Now we again calculate the mutation score of new test suite and compare the mutation score with old test suites.

New Test Suite 2

S. No.	A	B	C	Expected Output
1	20	30	40	40
2	10	5	5	10
3	6	2	10	10
4	6	10	20	20

We calculate the mutation score of new test suite 2.

For New Test Suite.

Mutant M1.

S. No.	A	B	C	EO	OO
1	20	30	40	40	30
2	10	5	5	10	5
3	6	2	10	10	10
4	6	10	20	20	20

Mutant M1 is killed

Mutant M2.

S. No.	A	B	C	EO	OO
1	20	30	40	40	40
2	10	5	5	10	5
3	6	2	10	10	10
4	6	10	20	20	20

Mutant M2 is killed

Mutant M3.

S. No.	A	B	C	EO	OO
1	20	30	40	40	40
2	10	5	5	10	5
3	6	2	10	10	6
4	6	10	20	20	20

Mutant M3 is killed

Mutant M4.

S. No.	A	B	C	EO	OO
1	20	30	40	40	40
2	10	5	5	10	10
3	6	2	10	10	10
4	6	10	20	20	10

Mutant M4 is killed

Mutant M5.

S. No.	A	B	C	EO	OO
1	20	30	40	40	40
2	10	5	5	10	10
3	6	2	10	10	2
4	6	10	20	20	20

Mutant M5 is killed

Mutation Score = Number of mutants killed/ Total number of mutants.

Total mutant = 5

Mutant kill = 5

Mutant alive =0

So,

Mutation Score for New Test Suite 2 is **1.**

As we can see that the new test suite 2 is also optimal with mutation score 1, so we can say that our algorithm is independent of test cases order within a test suite in aspect of obtaining the optimal test suits. But we observe that there is change in test cases of new test suite 2. As in new test suite 2, test case (10, 10, 10) which is in new test suite 1 is replaced by test case (20 ,30, 40). Therefore, test cases in optimal test suits are dependent of test cases order within a test suite.

Compare test suites Mutation Score

S. No.	Test Suit Name	Mutation Score	Optimality
1	Test Suite A	0.8	Nearly Optimal
2	Test Suite B	0.6	Not Optimal
3	New Test Suite	1	Optimal
4	New Test suite 1	1	Optimal
5	New Test Suite 2	1	Optimal

Table 5.1 Comparing Mutation Score

As we can see that the mutation score of new test suite, new test suite 1and new test suite 2 are better than the other two test suites (A and B). The new test suite 1 and new test suite 2 is also optimal with mutation score 1, so we ca say that our algorithm is independent of test suites order and test cases order within a test suite in aspect of obtaining the optimal test suits.

5.3 Dividing the Algorithm.

Now we have divided our algorithm in two algorithms and we have seen how they affect the test data.

5.3.1 Algorithm for optimal test cases.

In this algorithm we have just use the half part of our algorithm. We just use that part of algorithm which has optimized the test cases by calculating the mutant score (1) and mutant score (2). After that we select only those test cases for new test suit which have both the mutant score (1) and mutant score (2) are equal to 1. We observed that this algorithm only minimized the test data but not optimized the test data.

As in test suite B, we can see that all the test cases are optimal with both mutant score equal to 1 but the mutation score is 0.6 which is not optimal.

5.3.2 Algorithm for path coverage of SUT in each test suite.

In this algorithm we have just use the half part of our algorithm. We just use that part of algorithm which help us to covered all the path of SUT in each test suite. In this algorithm, we select test cases with different path for new test suit. We observed that this algorithm only minimized the test data but not optimized the test data.

As in test suite A, we can see that all the test cases belong with different path but the mutation score is 0.8 which is not optimal.

CHAPTER 6

CONCLUSION

It is concluded from the above discussion that the tool for Minimization of Automated Generated Test Data perform better than other tool developed. There are many tools and algorithms for automated generated test data with large solution space. But there are some other algorithms which minimize the solution space of the SUT for testing the software. But they are not so optimal and the tool which we developed, provides the optimal or near to optimal solution. Our developed tool is better than the other tools which were developed for the generation of test data, because it enables the test data to eliminate the not optimal test cases, which make our test suites more optimal than any other tools developed. This work would decrease the cost and time of testing for a given SUT considerably.

CHAPTER 7
REFERENCES

[1] Y. Singh, “Software Testing”, Cambridge University Press, UK, 2010.

[2] B. Korel, "Automated software test data generation," in *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870-879, Aug. 1990.

[3] C. Michael and G. McGraw, “Automated software test data generation for complex programs,” A Technical Report, Reliable Software Technologies, 1998.

[4] R. P. Pargas, M. J. Harrold, R. R. Peck. “Test-data generation using genetic algorithms” in Software Testing Verification Reliability, 9(4): 263---282, 1999.

[5] L. Bin, L. Zhi-Shu, C. Yan-Hong and L. Bao-Lin, "Automatic Test Data Generation Tool Based on Genetic Simulated Annealing Algorithm," Computational Intelligence and Security Workshops, 2007. CISW 2007. International Conference on, Heilongjiang, pp. 183-186, 2007.

[6] D. Jeya Mala, V. Mohan and M. Kamalapriya, "Automated software test optimisation framework – an artificial bee colony optimisation-based approach", IEEE, 10.1109/ICRITO.2016.7785020, 19 December 2016.

[7] D.J. Mala, and V. Mohan, “Hybrid Tester - An Automated Hybrid Genetic Algorithm Based Test Case Optimization Framework for Effective Software Testing,” International Journal of Computational Intelligence: Theory and Practice, vol.3, no.2, pp 81-94, 2008.

[8] R. Malhotra and M. Garg. "An adequacy based test data generation technique using genetic algorithms." In Journal of information processing systems, vol. 7, no. 2. pp. 363-384. 2011.

[9] R. Malhotra, Poornima and N. Kumar, "Automatic test data generator: A tool based on search-based techniques," 2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, 2016, pp. 570-576.

[10] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Yves Le Traon. "Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing", 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, 2013

[11] Feras A. Batarseh, Avelino J. Gonzalez. "Predicting failures in agile software development through data analytics", Software Quality Journal, 2015

[12] Lili Pan, Junyi Li, Beiji Zou, Hao Chen. "Bi-Objective Model for Test-Suite Reduction Based on Modified Condition/Decision Coverage" , 11th Pacific Rim International Symposium on Dependable Computing (PRDC'05), 2005

[13] J.A. Jones, M.J. Harrold. "Test-suite reduction and prioritization for modified condition/decision coverage", IEEE Transactions on Software Engineering, 2003

[14] Hanh, Le Thi My, Nguyen Thanh Binh, and Khuat Thanh Tung. "A Novel Fitness function of metaheuristic algorithms for test data generation for simulink models based on mutation analysis" , Journal of Systems and Software, 2016.

[15] Subramanian. "A Tool for Generation and Minimization of Test Suite by Mutant Gene Algorithm", Journal of Computer Science, 2011