CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

Test Case Prioritization Based on Path Complexity

by

Tahseen Afzal

A thesis submitted in partial fulfillment for the degree of Master of Science

in the

Faculty of Computing
Department of Computer Science

2018

Copyright © 2018 by Tahseen Afzal

All rights reserved. No part of this thesis may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, by any information storage and retrieval system without the prior written permission of the author.

DEDICATION

To my BELOVED parents,

The symbol of love and kindness.

CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY
ISLAMABAD

CERTIFICATE OF APPROVAL

Test Case Prioritization Based on Path Complexity

by

Tahseen Afzal

MCS161013

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External Examiner	Dr. Muhammad Uzair Khan	FAST, Islamabad
(b)	Internal Examiner	Dr. Muhammad Azhar Iqbal	CUST, Islamabad
(c)	Supervisor	Dr. Aamer Nadeem	CUST, Islamabad

Dr. Aamer Nadeem
Thesis Supervisor
October, 2018

Dr. Nayyer Masood
Head
Dept. of Computer Science
October, 2018

Dr. Muhammad Abdul Qadir
Dean
Faculty of Computing
October, 2018

Author's Declaration

I, **Tahseen Afzal** hereby state that my MS thesis titled "**Test Case Prioritization Based on Path Complexity**" is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/abroad.

At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my MS Degree.

(Tahseen Afzal)

Registration No: MCS161013

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled "*Test Case Prioritization Based on Path Complexity*" is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been dully acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS Degree, the University reserves the right to withdraw/revoke my MS degree and that HEC and the University have the right to publish my name on the HEC/University website on which names of students are placed who submitted plagiarized work.

(Tahseen Afzal)

Registration No: MCS161013

”There is one thing even more vital to science than intelligent methods;

and that is,

the sincere desire to find out the truth, whatever it may be.”

Charles Sanders Pierce

Acknowledgements

All worship and praise is for ALLAH (S.W.T), the creator of whole worlds. First and leading, I would like to say thanks to Him for providing me the strength, knowledge and blessings to complete this research work. Secondly, special thanks to my respected supervisor Dr. Aamer Nadeem for his assistance, valuable time and guidance. I sincerely thank him for his support, encouragement and advice in the research area. He enabled me to develop an understanding of the subject. He has taught me, both consciously and unconsciously, how good experimental work is carried out. Sir you will always be remembered in my prayers. I would like to thank Hassaan Minhas and Mubashir Kaleem, students of BS(CS), for implementation of mutation testing tool. I would also like to thank all members of CSD research group for their comments and feedback on my research work. I am highly beholden to my parents, for their assistance, support (moral as well as financial) and encouragement throughout the completion of this Master of Science degree. This all is due to love that they shower on me in every moment of my life. No words can ever be sufficient for the gratitude I have for my parents. I hope I have met my parents high expectations. I pray to ALLAH (S.W.T) that may He bestow me with true success in all fields in both worlds and shower His blessed knowledge upon me for the betterment of all Muslims and whole Mankind. Aameen

Tahseen Afzal

Abstract

Software undergoes many modifications after its release. Each time the software is modified, it needs to be re-tested. After modification, regression testing is performed to ensure that the modification has not introduced any errors in the software and the software continues to work correctly. Regression testing is an expensive process, since the test suite might be too large to be executed completely. There are three types of cost reduction techniques used in regression testing, i.e., test case selection, test suite minimization and test case prioritization. These techniques can be used to reduce the cost of regression testing and improve the rate of fault detection. The focus of our research is on test case prioritization. Instead of minimizing test suite or selecting fewer test cases, test case prioritization orders test cases in such a way that the test cases detecting more faults are executed earlier. In case of limited resources; instead of executing complete test suite, only top priority test cases can be executed to ensure the reliability of the software. A number of white box and black box prioritization techniques have been introduced to prioritize test cases.These techniques are mainly based on coverage based prioritization, such as statement coverage, branch coverage, module coverage etc.

In this thesis, we propose an approach which uses path complexity and branch coverage to prioritize test cases. The approach is based on assumption that the complex code is more likely to contain faults. Halstead's metric has been used to calculate the path complexity of the test cases. The test cases with higher path complexity are assigned higher priority. This approach can significantly increase the rate of fault detection as the test cases are prioritized on the basis of path complexity. We have evaluated and compared our approach with branch coverage based prioritization technique using some example programs. The results show that our proposed approach performs better than existing branch coverage based approach in terms of APFD (Average Percentage of Faults Detected).

Contents

Author's Declaration iv

Plagiarism Undertaking v

Acknowledgements vii

Abstract viii

List of Figures xii

List of Tables xiii

List of Abbreviations xiv

1 Introduction 1

1.1 Regression Testing 2
1.1.1 Test Suite Minimization 4
1.1.2 Regression Test Selection (RTS) 5
1.1.3 Test Case Prioritization 5
1.2 Prioritization Criteria 6
1.2.1 Black Box Proritization Approaches 7
1.2.1.1 Interaction Coverage Based Prioritization: 7
1.2.1.2 Requirements Clustering Based Prioritization: . . 8
1.2.1.3 History Based Test Case Prioritization: 8
1.2.1.4 Hierarchical System Test Case Prioritization Technique: 8
1.2.2 White Box Prioritization Approaches 9
1.2.2.1 History Based Prioritization: 9
1.2.2.2 Coverage Based Prioritization Techniques: 10
1.3 Prioritization Algorithms 11
1.3.1 Greedy Algorithm 11
1.3.2 Additional Greedy Algorithm 11
1.3.3 Genetic Algorithm 12
1.3.4 Combined Genetic and Simulated Annealing Algorithm . . . 12
1.3.5 Ant Colony Optimization 12

1.4 Problem Statement 13
1.5 Research Questions 13
1.6 Research Methodology 14
1.7 Thesis Organization 15

2 Literature Review **17**
2.1 Fault Based Prioritization Techniques 19
2.2 Coverage Based Prioritization Approaches 21
2.3 Analysis and Comparison 22
2.4 Gap Analysis 25

3 Proposed Approach **27**
3.1 Code Complexity Metrics 28
3.1.1 Lines of Code Metric 28
3.1.2 Function Point (FP) Analysis 29
3.1.3 McCabe's Cyclomatic Complexity 29
3.1.4 Halstead's Metric 30
3.1.4.1 Operands 30
3.1.4.2 Operators 31
3.1.4.3 Size of The Vocabulary (n) 31
3.1.4.4 Program Length (Program Size N) 32
3.1.4.5 Volume of Program (V) 32
3.1.4.6 Difficulty Level (D) 33
3.1.4.7 Program Level (L) 33
3.1.4.8 Effort to Implement (E) 33
3.1.4.9 Time to Implement (T) 34
3.1.4.10 Estimated Program Length 34
3.1.4.11 Number of Delivered Bugs (B) 34
3.1.5 Why Halstead's Metric? 35
For this example: 37
3.2 Path Complexity Based Prioritization 37
3.2.1 Path Extraction 38
3.2.2 Calculation of Path Complexity 38
3.2.3 Applying Prioritization Algorithm 38
3.3 Example 40

4 Implementation **42**
4.1 Implementation Details 43
4.1.1 Equivalence Class Partitioning 43
4.1.2 Boundary Value Analysis 44
4.2 User Interface 45
4.2.1 Test Case Generation 45
4.2.2 Path Extraction 47
4.2.3 Path Complexity Calculation 48
4.2.4 Test Case Prioritization 49

5 Results and Discussion **50**
5.1 Subject programs 51
5.1.1 Simple Calculator Program: 51
5.1.2 Quadratic Equation Problem: 51
5.1.3 Triangle Problem: 51
5.2 Comparison 55

6 Conclusion and Future Work **62**
6.1 Future Work 64

Bibliography **65**

Appendix A **72**

Appendix B **82**

List of Figures

1.1 Regression Testing Techniques 4

3.1 An illustration of proposed approach 37

4.1 Architecture diagram of tool 43
4.2 Taking Boundary values 45
4.3 Displaying all five values 46
4.4 Displaying test cases 47
4.5 Test Case execution 48
4.6 Halstead measures for test suite 49

5.1 Graphical representation of fault detection of test cases for Quadratic Equation Problem 58
5.2 Graphical representation of fault detection of test cases for Simple Calculator Problem 59
5.3 Graphical representation of fault detection of test cases for Triangle Problem 60
5.4 Graphical Representation of APFD 60

List of Tables

2.1 Overview of State of the art (Fault Based Approaches) 23
2.2 Overview of State of the art (Coverage Based) 25

3.1 Mapping of the test case to entities covered and path complexity . . 40

5.1 Subject Programs summary 52
5.2 Subject Programs' Priority Lists 53
5.3 Subject Programs' APFD 57

List of Abbreviations

SDLC Software Development Life Cycle

NIST National Institute for Standards and Technology

SQA Software Quality Assurance

APFD Average Percentage of Faults Detected

LoC Lines of Code

FP Function Point

Chapter 1

Introduction

Software testing can be defined as a group of activities performed to evaluate some aspect of a piece of software[1]. The process of software testing is carried out for quality evaluation of software under test and to make the software product better. The major objective of software testing is to find out errors in program under test. Moreover, it ensures that the requirements of the customer(s) are fulfilled by the software under test. Software testing is considered an expensive and critically important phase in process of software development [2]. Testing ensures the quality and correctness of software. Testing should utilize minimum resources to reduce the testing cost as design and development phase already consume many resources [3]. Testing is performed in almost every phase of Software Development Life Cycle (SDLC). According to the report of "National Institute for Standards and Technology (NIST)", almost $60 billion per year are utilized on software testing by US economy[4]. In 2016, the cost of testing jumped to $1.1 trillion per annum [5]. Some effective testing approach may decrease the cost of testing to approximately $22 billion. Therefore advanced software testing is required to minimize the cost of software testing process. Software may contain different kinds of errors which include design error, input error, hardware error, statement error, specification error etc. Different testing types are used to identify and fix these errors [6]. There are chances of fault occurrence during any stage of development. Errors or faults are required to be discovered and removed timely so that they may not

further be transmitted to the next phases of software development [7]. Testing identifies the errors in the software under test but it does not guarantee that the software is error free[8].

Software testing holds the most significant importance among all phases of SQA (Software Quality Assurance). On each modification of software, new test suites are generated to test the modified piece of code. Each modification increases the size of test suite[9]. Maintenance is one of the most costly phases of System Development Life Cycle(SDLC). During this phase, software undergoes many modifications and is updated continuously. The modified software needs to be retested to identify and fix the faults in software introduced during modification process. Thus the maximum cost of maintenance phase is consumed on regression testing[10].

1.1 Regression Testing

Whenever modifications are made to the software, it needs to be retested to ensure that the previous functionality of the software is not affected by the change. This type of testing is called regression testing (RT). Basically, the regression testing shows the verification of modified software[11]. Regression testing is a costly process which needs to be performed frequently in order to validate the correctness of the modified software after each modification [12]. Regression testing ensures that the new changes in the software do not affect the functionality of the present part of the software. Regression testing might begin in development phase after detecting errors and their correction by reusing the existing test cases. Modifications in the software can occur at maintenance phase when the software is updated, revised or improved. There are two types of modifications: 'Corrective Maintenance' and 'Adaptive and Progressive maintenance'. In corrective maintenance, specifications of the system are not changed. Adaptive and progressive maintenance involves changes in specifications. On the basis of maintenance, regression testing falls in two categories: Corrective Regression Testing and Progressive regression

testing. Corrective regression testing is carried out when software specifications remain unchanged where as Progressive regression testing is carried out when modifications effect the software specifications with addition of new features [13].After modifying the software, there are five classes of test cases; Reusable, re-testable, obsolete test cases which already exist in test plan T and two classes of test cases i.e., New structural and New Specification, are generated after modifying the software for regression testing. Reusable test cases test the part of the program which remained unchanged, re-testable test cases evaluate the part of software which undergoes some modifications or changes; whereas, obsolete test cases are those which cannot be used any more due to following reasons:

1. They do not test anymore, what they were intended to test.
2. Due to modification, input/output relation no more exists.
3. Modification may result in faults and structural test cases no more provide required structural coverage.

New structural and new specification test cases are used to test the modified constructs and changed specifications respectively [14]. Whenever the software is modified, it needs o be tested again to check if the software performs as desired. Regression testing is very expensive process, since it might be costly to run the complete test suite. The most suitable approach, called retest-all, is to run the complete test suite. The expansion of software results in increase in size of test suite; therefore, it becomes difficult to execute the complete test suite. Need of the hour is to take this problem under consideration to reduce the efforts involved in performing regression testing [13].

To lower the cost of regression testing, software testers choose test suite using certain Regression testing techniques. See Figure 1.1

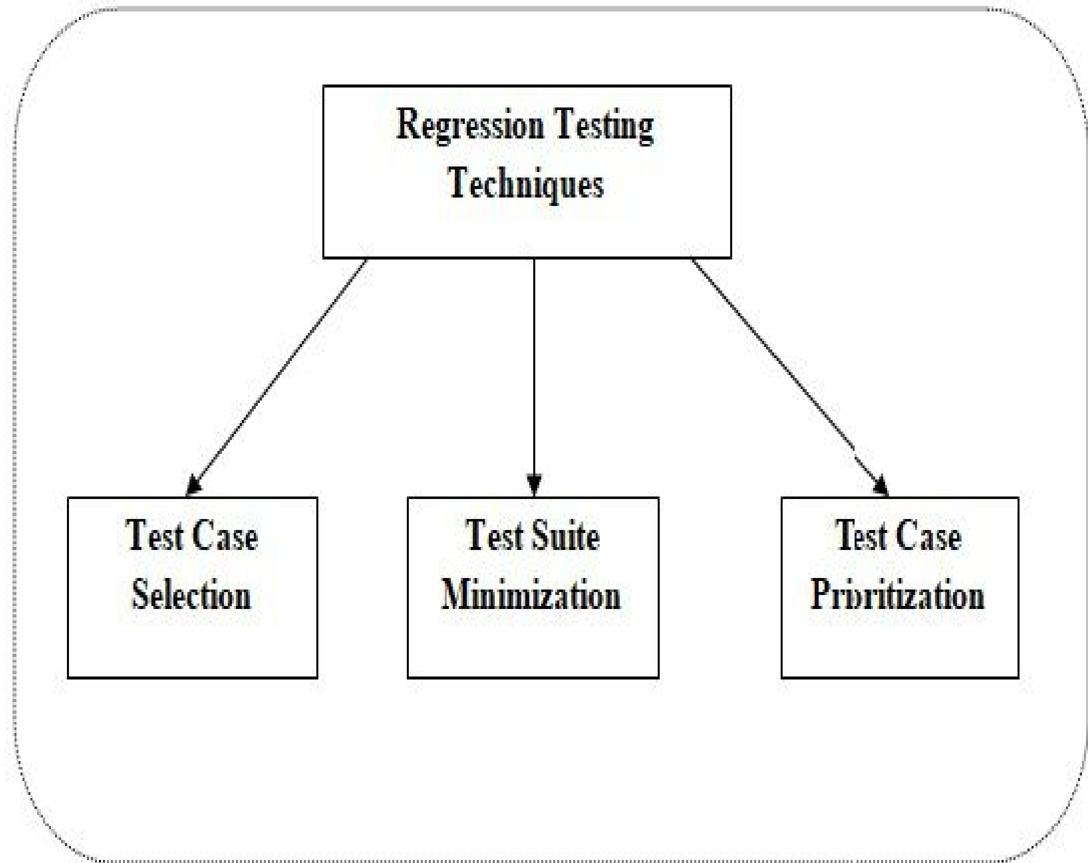

FIGURE 1.1: Regression Testing Techniques

1.1.1 Test Suite Minimization

Test suite minimization technique reduces the size of a test suite by removing redundant test cases from the test suite. This technique removes the test cases which cover the same piece of code. The minimal hitting set stores the selected test cases. [15]. To generate minimal hitting set is NP-Complete problem which has exponential time complexity. Many approaches have been proposed by researchers to solve the minimal hitting set problem [16–18][19]. Different coverage criteria are used for minimization and the test cases that meet the coverage criteria are selected and remaining test cases are discarded [13]. Test suit minimization approach selects the test cases which cover the part of software which undergoes some change, thus reducing or minimizing the overall test suite [12]. This technique may result in elimination of some useful test cases.

1.1.2 Regression Test Selection (RTS)

In this technique we find the modified part of the system and parts affected by modification, then choose the test cases covering the modified sections of the system and discard the remaining test cases [13]. Only the most relevant test cases are selected for execution. The modification can influence the performance of the complete software. Thus, execution of selected tests only may not guarantee that all faults have been detected.

1.1.3 Test Case Prioritization

In Test case prioritization technique, the test cases are ordered in such a way that the important test cases are placed first in prioritized list. The main objective of prioritization is to improve the rate of fault detection and minimize the regression testing cost[13]. As the selected subset of test suite might be too large to be executed fully and certain time and resources constraints make it difficult to execute the complete test suite. Moreover, if testers do not want to eliminate the relevant test cases, they focus on ordering the test cases in a prioritized sequence so that the test cases with higher priority are executed prior in regression testing. In prioritization process, no test case are removed or eliminated, rather it arranges test cases based on certain criteria.

Hence the faults in software under test can be detected earlier which reduces the time and cost of regression testing [20]. Test case prioritization detects a large number of faults by executing only few test cases. The priority list is used to decide when to stop the process of regression testing. Moreover, test case prioritization ensures that if the testing process is halted prematurely then the test cases with higher priority are executed earlier [13]. Test case prioritization was initialy of all introduced by Wong et al., [21] and applied on selected test suite which was selected by RTS technique. Harrold [22] and Rothermel et al., [23] extended this concept and presented it in a more concise manner.

Test case prioritization can be used in combination with regression test selection and test case minimization. In such case, prioritization is applied on a selected or minimized subset of test suite [24].

Formally, test case prioritization can be defined as:

Definition: The Test Case Prioritization Problem:

Given: A test suite 'T' initially designed for original program P; the set of permutations of T, PT; and f: $PT \rightarrow$R , a function from PT to the real numbers.

Problem: To find T' ∈ PT such that (∀T")(T"∈PT) (T"≠T') [f(T') ≥f(T")]

1.2 Prioritization Criteria

The fault detection rate in regression testing is affected by the order in which test cases are executed. This detection is actually the early detection of faults in testing processing [20]. Due to limited time and resources, software testers use a priority list of test cases to decide when to halt the process of testing. Priority list increases the rate of fault detection and debugging at early stages. Moreover; if the testing process is halted due to any reason, it ensures that the important test cases with higher priorities have been executed first which increases the reliability of testing process. [20]. Different prioritization techniques are available including white box and black box techniques. These techniques enable testers to assign priorities to test cases. Prioritization techniques may use one or more than one criteria. Any prioritization criterion assigns the award values to test cases. These values are used for prioritizing the test cases. The test case with higher award value is assigned higher priority and is listed earlier in prioritized list. In other words, based on the selected criteria, test cases with low cost are assigned higher priority. Prioritization techniques can use:

1. The code coverage factor like, statements, branches, spanning entities or functions for assigning award values to test cases [24].

2. Some techniques use the information of specifications, requirements or interaction of different events for ordering the test cases [25].

All the prioritization criteria generate a prioritized list of test cases. The prioritized list is then evaluated. The basic parameter to evaluate a prioritized list is fault detection rate i.e., Average Percentage of Faults Detected (APFD). It is a measure to check how early a particular test suite detects faults by using the particular prioritized list of test cases in test suite [23]. APFD is calculated using following formula:

$$APFD = 1 - (TF_1 + \cdots\ldots TF_m)/nm + 1/2n \tag{1.1}$$

Where T is the test suite with n test cases and F is the set of m faults identified by T. For ordering T', TFi represents the order of the first test case that exposes the ith fault F_i.

Test case case prioritization approaches are classified into two broad categories: Black box prioritization and white box prioritization techniques.

1.2.1 Black Box Proritization Approaches

Black box techniques are based on specifications and requirements. They do not need the access to source code [26]. Thus, they have no knowledge of structure of the software. Some common black box prioritization techniques are briefly discussed below.

1.2.1.1 Interaction Coverage Based Prioritization:

In event driven systems, the number of event combinations and sequence grow exponentially with the number of events. Therefore; in event driven systems, it is very difficult to manage test suite. Bryce and Memon [27]in 2007 proposed

a testing technique which extends the t-way software interaction over sequences of events. The proposed algorithm by Bryce and Memon [27] greedily selects a test case covering the maximum number of t-tuples of event interactions between unique windows which remained uncovered previously. In case of more than one test cases covering same number of event interactions, the tie is broken randomly.

1.2.1.2 Requirements Clustering Based Prioritization:

Software under test may contain many requirements of high priority but all of the requirements are not equally important. A new approach was introduced by Arafeen and Do [28] which uses requirements information to prioritize the test cases. Text mining approach is used to extract words by their proposed approach. On the basis of extracted words, requirements are clustered and later on test cases are prioritized in these requirement clusters. Code complexity is used to prioritize test cases within clusters.

1.2.1.3 History Based Test Case Prioritization:

In black box testing environment, source code of the program is not available. Only limited information is available to prioritize the test cases. In 2007, a black box technique was introduced by Qu et al., [29] for test case prioritization. Their proposed approach uses the run-time and test history information. History information is used to initialize the test suite. Test case relation matrix R is formed by using available information. R matrix indicates the fault detection relationship of test cases. Test cases are ordered using test case relation matrix and run time information.

1.2.1.4 Hierarchical System Test Case Prioritization Technique:

Many approaches have been introduced to prioritize test cases on requirements basis. However; along with requirements, many other factors like implementation

and test case complexity also contribute in test case prioritization. In 2013 Kumar et al., [30] introduced a hierarchical test case prioritization approach based on requirements. They performed prioritization process at three levels. In first step, a priority is assigned to each requirement on the basis of 12 different factors including customer assigned priority, developer assigned priority, requirement volatility, fault proneness, expected fault, implementation complexity, execution frequency, traceability, show stopper requirement, penalty, cost and time. Customer, developer, analyst and tester assign values to those requirement factors. After prioritizing all of the requirements, a mapping between each requirement and its corresponding modules is performed. In case of more than one modules corresponding to any requirement, modules are prioritized using cyclomatic complexity. The last level of prioritization process includes the test case prioritization. Test cases are mapped to corresponding modules. In the last stage, test cases corresponding to modules are then prioritized based on 4 factors including test impact, test case complexity, requirement coverage and dependency.

1.2.2 White Box Prioritization Approaches

White box prioritization techniques are based on source code. The testers need access to the actual source code of the program. Access to the actual code can reveal the code coverage and it can help in early detection of faults in subject program [25]. Different white box techniques are defined below.

1.2.2.1 History Based Prioritization:

In 2002 Kim and Porter [12] proposed history based prioritization approach to prioritize test cases. In this approach, the information is used which is obtained from previous execution cycles of software as criteria for selection of subset of test suite that must be executed for a modified software. RTS technique was applied to test suite T that produced T' in the 1st step. After that in 2nd step, every test in T' was assigned selection probability. In 3rd step, probabilities assigned in

previous step were used to select and execute a test case. The final step includes the repetition of 3rd step until the testing time is finished.

1.2.2.2 Coverage Based Prioritization Techniques:

Rothermel et al., [24] introduced two white box prioritization strategies: "Total" and "Additional" which include following techniques:

1. **Total Function Coverage Prioritization:** In this technique, the criterion used to prioritize test cases is total number of functions covered. Test case which achieves coverage of the maximum number of functions is assigned the highest priority. In case of more than one test cases covering the same number of functions, the test cases are selected randomly.

2. **Additional Function Coverage Prioritization:** This approach assigns priorities to the test cases on the basis of uncovered functions. It selects a test case which achieves coverage of maximum number of functions iteratively, then updates the coverage information of un-prioritized test cases to indicate their coverage of functions which were left uncovered. The process is repeated until no function remains uncovered. If multiple test cases cover the same number of functions, then random ordering is applied.

3. **Total Statement Coverage Prioritization:** The total number of statements covered is used as criterion to order test cases. Test case covering the maximum number of statements is given the highest priority. If same number of statements is covered by multiple test cases then they are ordered randomly.

4. **Additional Statement Coverage Prioritization:** It works the same as additional function coverage but instead of using function, the prioritization is done on the basis of statements remained uncovered by previous selection.

5. **Total Branch Coverage Prioritization:** Test cases are assigned priorities using the maximum number of branches covered. The branch coverage

is defined as coverage of each possible outcome (true and false) of the condition. If the function does not contain any branch, then the function itself is considered as a branch entry and branch is said to be covered by each test case that invokes that function.

6. **Additional Branch Coverage Prioritization:** It is same as additional statement coverage prioritization but the prioritization is done using uncovered branches.

1.3 Prioritization Algorithms

Regression testing is very important and a lot of research has been done on different approaches of prioritization. Many prioritization algorithms have been introduced which use different criteria to prioritize test cases. Few commonly known prioritization algorithms are discussed below.

1.3.1 Greedy Algorithm

Greedy algorithm is a straight forward approach of prioritization. The test cases are assigned priority on the basis of total number of entities covered. Test case covering the maximum number of entities is assigned the highest priority in ordered list and those covering lesser number of entities are assigned lower priorities according to the number of covered entities [20].

1.3.2 Additional Greedy Algorithm

Additional greedy algorithm assigns priorities to the entities which remain uncovered by previously selected test case(s). Higher priority is assigned to the test case which covers maximum number of entities not covered so far [20].

1.3.3 Genetic Algorithm

Genetic algorithm selects random population from a given set. The population is replaced by a new population using fitness function which is based on total code coverage [31].

1.3.4 Combined Genetic and Simulated Annealing Algorithm

Combined Genetic and Simulated Annealing Algorithm is combination of two algorithms i.e., Genetic Algorithm(GA) and Stimulated Annealing (SA). Thus, it is also called GASA. It takes the advantages of both of the algorithms. The quick processing quality of GA and effective solution feature of SA is combinely used to prioritize test cases. In the first step, solution is generated using GA, Later on, it is refined and made more effective by using SA [32].

Algorithms, discussed above, use some coverage data for assigning priorities to test cases either by using total strategy or additional strategy. The algorithm assigns highest priority to the test case covering maximum entities.

1.3.5 Ant Colony Optimization

Ant Colony Optimization is an optimal path searching approach based on the natural behavior of ants searching for food. During the search of food, the ant leaves behind a chemical substance called 'pheromone'. The ants following that path, smell the odor of chemical traces left behind by the leading ant and thus follow the same path. The most optimal path is found out by teamwork and evaporation process of 'pheromone' [33].

Path Prioritization Ant Colony Optimization (PP-ACO) is an algorithm which follows the Ants' foraging behavior to generate optimized path sequence of decision to decision (DD) paths of a graph. In path testing, the said algorithm takes the

coverage of all paths and gives the most suitable and optimal path sequences. The sequences are then prioritized according to the strength of paths. PP-ACO algorithm uses forward move of ant from source (i.e. Nest) to destination (i.e. Food) and vice versa in backward move. Ants take probabilistic decisions in forward PP-ACO for their next move among the available nodes. Ants keep the track of edges visited during forward movement and the cost of visiting each node is recorded. The solution cost is built according to the weight (cost) of visiting each node [34].

1.4 Problem Statement

Most of the existing prioritization techniques are based on code coverage [35]. Coverage alone cannot improve the effectiveness of fault detection rate of test suite [36]. In such techniques, different coverage criteria are used to prioritize test cases to improve rate of fault detection. The techniques using structural complexity of the software can detect faults faster as compare to code coverage based approaches [37]. Coverage based approaches consider entities covered by the test cases in software under test.

While using coverage criteria, the test cases with greater coverage are given higher priorities, whereas, there might be higher complexity value of test cases with smaller coverage. The program complexity can affect the fault detection rate. Thus, in this thesis two parameters for test case prioritization; the structural complexity and branch coverage are taken into account to improve the fault detection rate of test suite.

1.5 Research Questions

In this research work, we have used path complexity based on computational complexity by Halstead's Metric as a criterion to generate a priority list of test

cases to improve rate of fault detection. However, the following questions are taken into account:

RQ. 1: What are the gaps in existing white box prioritization techniques?

To answer this research question, a literature survey is conducted through which we have identified the gaps in existing and most commonly used techniques.

RQ. 2: How well does the proposed prioritization technique compare with the well studied white box prioritization techniques with respect to fault detection rate?

Our research is focused on to find out answer to the above mentioned research questions with reference to the prioritization algorithm.

1.6 Research Methodology

1. First of all we have done literature review to identify the most relevant and most commonly used white box prioritization techniques. After studying various prioritization techniques, we have reached the conclusion that these techniques are coverage based. There are only a few fault proneness based prioritization techniques and very few techniques have considered the structural complexity of the code in prioritization [37].

2. To overcome the gaps in existing techniques, we have proposed a new approach that will assign priority to each test case on the basis of its path complexity and branch coverage.

3. The implementation of our approach has been performed in following steps:

   (a) In the first phase, we collected all data including subject programs. After collecting the data, our code analyze extracts the path executed by each test case.

   (b) The extracted path is passed to Halstead's function to calculate the path complexity (i.e. error proneness) of test cases for the complete

test suite of the respective programs. The Halstead's metric calculates the complexity of entire program. We have made some modifications in implementation of Halstead metric in such a way that it calculates the complexity of extracted path(i.e. piece of code) rather than complete program.

(c) The extracted path is also passed to our coverage analyzer which analyzes the extracted path and traces the branches covered by each test case.

(d) In the next step, we generate the prioritized list of test cases based on path complexity by assigning the higher priority to the test case with higher path complexity. The tie among test cases is broken by using branch coverage as secondary criterion.

(e) Next stage is about generating another prioritized list for same program and its corresponding set of test cases by using additional branch coverage technique, since additional branch coverage techniques is considered as one of the strongest approach in order to improve APFD of test suite [25]. Additional Branch coverage is considered one of the best prioritization method [38]. Therefore, we have used 'Additional Branch Coverage' white box technique for comparison and evaluation of our proposed approach.

4. After creating both prioritized lists (additional branch coverage based and path complexity combined with branch coverage), we perform a comparison between both techniques. We have used the Average Percentage of Faults Detected (APFD) as the main parameter for comparison.

1.7 Thesis Organization

Rest of the thesis is organized as follows:

The Chapter 2 is about literature review. We have discussed different research studies about fault based prioritization techniques and coverage based prioritization techniques. Chapter 3 is based on proposed solution. In Chapter 4, implementation details are presented. Chapter 5 is about results and discussions. Chapter 6 is about conclusion we have made after comparing our proposed approach with existing approach. Future work has also been discussed in Chapter 6 i.e., how this work can further be extended.

Chapter 2

Literature Review

Test case prioritization orders test cases on the basis of some prioritization criterion. The ordered list provides maximum benefits to software testers. Each test case is assigned a priority. Software testers make sure that the test cases with higher priorities run earlier during testing process [13]. The order of test cases in which they are executed has great influence on test suite's rate of fault detection [20]. In case of limited availability of resources, software testers use priority list to decide when to stop the testing process by executing few top priority test cases rather than executing the complete test suite. Prioritized list increases the probability that if the testing process is halted due to some constraints, the most important test cases with higher priorities have been executed. It enables debugging at early stages of testing and increases the rate of fault detection as well [20].

The test case prioritization techniques have been separated into White box and Black box prioritization. Test case prioritization techniques prioritize test cases in such a way that it helps to improve the rate of fault detection and meet the goals earlier. Different prioritization techniques use different criteria to improve the process of regression testing. The test cases are given priority on the basis of selected criteria. Test cases with higher priority are executed earlier, increasing the rate of fault detection and debugging at early stages of testing.

Two main white box prioritization strategies were introduced by Elbaum et al., [20] and Elbaum et al., [39]: "Total" and "Additional", which have already been discussed in Chapter 1.

Depending on these strategies, prioritization approach can be single criterion based or multicriteria based. Prioritization techniques using single criterion, order test cases on the basis of a single criterion. Whereas; multicriteria based prioritization approaches use more than one criterion to prioritize test cases. The criteria can be white box or black box.

Black box prioritization techniques have been less well studied as compared to white box techniques; however recent advances have developed some black box prioritization techniques which focus on promoting diversity among test cases. Although, black box prioritization techniques are highly competitive but they are not commonly used given that very less information is provided, i.e. no structural information is provided to use as prioritization base [25].

White box criteria consider elements of source code, e.g., statements, branches, functions etc for assigning priorities to test cases. In white box prioritization techniques, code complexity has great importance. There are different software complexity metrics to measure the complexity of code. Some commonly known complexity metrics will be discussed in Chapter 3. The measure of the cost of software development, maintenance and usage is known as Software Complexity Metric [40]. Software complexity metrics are closely related to error distribution in subject code [41]. Many testers use code coverage for prioritization. Only few researchers have considered code complexity for test case prioritization [37]. Chances of fault occurrence are higher in complex code as compare to simple statements. Code coverage based approaches assume that there are more chances of fault occurrence in the test case covering maximum elements [36]. Whereas; complexity based approaches assume that fault occurrence rate is higher in complex piece of code [37]. Since large software systems can be used more than 15 years, computer

scientists and researchers have put great efforts to measure the complexity of software in previous several years. Some estimates show that 40 to 70% expenses are consumed on maintenance of existing software systems [42].

Focus of our research is on prioritization criteria. In this chapter, different fault based and code coverage based prioritization techniques are being discussed.

2.1 Fault Based Prioritization Techniques

Test case prioritization techniques help in improving the rate fault detection in regression testing. In most of the existing techniques, it is assumed that all faults have equal severity; however, it is not practically correct. Zengkai et al.,[37] proposed an approach on the basis of program structure analysis. It is claimed by the authors that their proposed approach detects severe faults earlier. Moreover, the technique can be applied both for regression testing and non –regression testing. The main idea of their approach is to compute the testing importance of each module. Testing importance of module considers two factors; fault proneness and importance of module from system perspective and user perspective. Test cases are prioritized on the basis of testing importance of module. To evaluate the effectiveness of proposed approach, authors implemented it using Apros, a test case prioritization tool. Authors also introduced a metric APMC (Average of the Percentage of fault-affected Modules Cleared per test case) which measures the effectiveness of various prioritization techniques. Improved APFDC metric has been proposed, which takes fault severity and test cost in account, is used to compute the effectiveness of proposed approach based on module level complexity.

Prakash and Rangaswamy [43] proposed a modular based test case prioritization technique. The proposed technique performs regression testing in two stages. First stage comprises of the prioritization of test cases based on modules coverage. In second stage, modular based ordered test suites are merged together for further prioritization. The study was based on fault coverage. This empirical study used three standard applications to validate the proposed approach. The algorithm

was compared with Greedy Algorithm and Additional Greedy Algorithm. APFD values show that modular based prioritization based approach is better than overall program test case prioritization. However, it considers fault coverage only whereas the nature of fault has not been taken in account.

Ahmed et al.,[44] proposed an approach of test case prioritization in which Genetic Algorithm with Multi-Criteria fitness Function is used. Fitness function considers weight of test cases, fault severity, fault rates and number of structural coverage items covered by each test case. Their proposed technique used total strategy to prioritize test cases. Authors have compared their approach with Condition Coverage, Multiple Condition Coverage and Statement Coverage using standard metric of APFD.

Tyagi and Malhotra [45] proposed an approach for test case prioritization on the basis of three factors, i.e., rate of fault detection, percentage of fault detected and risk detection ability. Authors have made comparison of proposed approach with different prioritization techniques such as un-prioritized test suite, reverse prioritization, random prioritization. The results were compared with those of Kavitha and Sureshkumar [46]. The approach proposed by Kavitha and Sureshkumar [46] detects severe faults earlier. The comparison of technique proposed by Kavitha and Sureshkumar [46] was made with un-prioritized test suite on the basis of APFD. The comparison of technique proposed by Tyagi and Malhotra, [45] with that of R. Kavitha and N. Sureshkumar[46] was made on the basis of APFD (Average Percentage of Faults Detected) value for each prioritization technique. Results show that the proposed technique outperforms other techniques compared with. Authors are simply using the values of three factors mentioned earlier. The fault and coverage data was taken from initial testing of software.

Only based on coverage criteria; when there is tie between multiple test cases, random test selection is made which may lead to decrease the rate of fault detection. It also leads to increase the time and budget index. Therefore, Wang et al.,[47] used fault severity as another criterion for test case prioritization. On the basis of severity of faults' effect on the software, faults are divided into four types; fatal,

serious, general and minor faults. Fault severity ranges from 2^0 to 2^3 (from minor to fatal faults). On the basis of fault severity, importance of test case is calculated and finally test cases are prioritized. Though authors have effectively described their approach; however, random selection of test cases with equal fault severity may cause important test cases to be prioritized at the end of the list rather than early execution.

Alves et al., [48] proposed a Refactoring based approach (RBA) to improve test case prioritization. Authors used five different refactoring faults models (RFMs) i.e., rename method, pull up field, move method, pull up method and add parameter. Authors created five versions of programs under test by seeding refactoring faults in programs. Authors gained APFD of 92% by RBA. The limitation of the proposed approach was, it required execution of complete test suite at least once which is very costly. Moreover, it deals with refactoring based modifications (i.e. The modifications which do not affect the behavior of the software.)

Tahvili et al., [49] used multi-criteria decision making technique TOPSIS in combination with fuzzy logic to prioritize test case. Authors used fault detection probability and execution time for prioritization. Authors used fault failure rate as a measure to detect the potential of test cases in test suite to detect faults. The probability of fault detection and execution time of test cases are provided by the testers which may prone to human error, thus this approach may not provide desired outcome.

2.2 Coverage Based Prioritization Approaches

Hla et al., [50] proposed a multiple criteria based prioritization approach using three criteria; Statement, Branch, Function coverage. This empirical study uses the PSO (Particle Swarm Optimization) algorithm for test case prioritization. Particle Swarm Optimization algorithm is a multi-object optimization technique used to set the ordering of objects. The main objective of this approach is to order the test cases to achieve high rate of fault detection. The empirical results

of this study show that Particle Swarm Optimization improves the performance of regression testing.

Kaur et al., [51] proposed an approaches which uses more than one criterion to generate a prioritized list of test cases. Authors used bank application for experimentation of their proposed approach. The main objective of the proposed approach was to find out the effectiveness of prioritized and un-prioritized test cases in terms of APFD (Average Percentage of Faults Detected), APSD (Average Percentage of Statement Detection) , APBD (Average Percentage of Branch Detection) and APPD (Average Percentage of Path Detection). The results of this study show that proposed method is outperforms the existing method.

Henard et al., [25] presented a comprehensive comparison of different white box strategies and newly introduced black box approaches. They found a little difference approximately 4% in performance of white box and black box approaches. They found hat the 'additional' coverage based approaches outperform 'total' coverage based approaches. Authors used 5 different open source programs and six versions of each program to perform and evaluate their proposed approach. They used the initial version to generate code coverage information and test suite was prioritized. Later 5 versions were used to execute the prioritized test suite and to find out the average percentage of fault detection. The results of their comparison of 10 white box approaches to 10 black box approaches show that white box approaches outperform black box approaches.

2.3 Analysis and Comparison

Very few studies have considered the structural complexity of the software under test [37]. The techniques using fault severity has not clearly discussed how to categorize faults or how to calculate fault severity [44].

The structural complexity in combination with coverage criteria has rarely been considered. The rate of fault detection is affected by structural complexity of the software [52]. The test cases giving larger code coverage may have smaller complexity as compare to those with smaller coverage. Thus, overlooking the

structural complexity may decrease the rate of fault detection which results in high cost of regression testing.

The studies which are considering multi-criteria based on code coverage, they overlook the structural complexity of the code. Such studies consider the code coverage, they do not consider the fault proneness of the code covered. Rather than an assumption is made that the test cases covering maximum entities can detect maximum number of faults, which is not always correct.

Table 2.1 provides an overview of some literature with respect to fault based prioritization techniques.

TABLE 2.1: Overview of State of the art (Fault Based Approaches)

Year	Author(s)	Prioritization Criteria	Algorithm	APFD	Limitations
2008	Zengkai Ma and Jianjun Zhao[37]	Fault severity	Program structure analysis.	N/A	•Module level approach •Needs info from system perspective as well as user perspective so difficult to find module imp. fault proned.
2012	N. Prakash and T.R. Rangaswamy [43]	Module coverage, fault coverage	Modular based test case prioritization	Greedy Algo: 83.33%, Add. Greedy Algo: 86.51% New Algo: 86.51%.	•Module based • Considers fault coverage only • Does not consider nature of fault
2012	Amr Abdel Fatah Ahmed, Dr.Mohamed Shaheen,Dr.Essam Kosba [44]	Control-flow coverage, Statement coverage, Fault severity	Genetic Algorithm	86.60%	• Uses existing coverage and fault data •Does not consider fault proneness or path complexity of test cases

Year	Author(s)	Criteria Covered	Algorithm	APFD	Limitations
2015	Manika Tyagi and Sona Malhotra [45]	rate of fault detection, percentage of fault detection, risk detection	Greedy Algorithm	85.5%	• Values from original software testing are used
2015	Yiting Wang, Xiaomin Zhao and Xiaoming Ding [47]	Additional statement coverage, fault severity.	Optimized Results	77.5%	• Random selection of test cases with equal fault severity may cause important test cases to be prioritized at the end of the list rather than early execution.
2016	Everton L. G. Alves, Patrcia D. L. Machado, Tiago Massoni, Miryung Kim [48]	Faults coverage	RBA	92%	• Deals with refactoring based modifications only
2016	Tahvili, S., Afzal, W., Saadatmand, M., Bohlin, M., Sundmark, D. and Larsson, S., [49]	Fault detection probability, execution time, or complexity	TOPSIS	N/A	• Testers' provided execution time may affect the fault detection rate.

Table 2.2 provides an overview of some literature with respect to coverage based prioritization techniques.

Table 2.2: Overview of State of the art (Coverage Based)

Year	Author(s)	prioritization Criteria	Algorithm	APFD	Limitations
2008	Khin Haymar Saw Hla,YoungSik Choi,Jong Sou Park [50]	Statement coverage, Branch coverage, Function coverage	PSO(Particle Swarm Optimization)	N/A	• Test cases with higher coverage may detect lesser faults
2014	Navleen Kaur, Manish Mahajan [51]	Statement coverage, Fault coverage, Path coverage, Branch coverage	Optimized Results	84.87%	• Not necessarily, the test cases with high coverage have higher path complexity
2016	Henard C., Papadakis M., Harman M., Jia Y., Traon Y.L. [25]	TS: Total Statement, AS:Additional Statement, TB: Total Branch, AB: Ad. Branch, AM: Ad. Method	White Box Approaches	TS:70%, AS:87%, TB:70% , AB:87% , TM: 70%	•Test cases with higher coverage may not detect higher number of faults

2.4 Gap Analysis

As we have studied here the literature of two different types of prioritization approaches i.e., fault based approaches and coverage based approaches. After the thorough study of above mentioned approaches, we have identified the following gaps in existing white box prioritization approaches.

In fault based prioritization approaches, the data of faults and test cases is taken from testing of original software. When software goes under some modification, new faults possibly be introduced during modification. The modification of a

particular part of the software may also affect the other parts of software. Therefore; in regression testing, the previous data of faults may not be useful due the following reasons:

- The faults occurred in initial (original) version are already been fixed so possibly, those faults may not occur again.
- Modification may introduce new faults in software. Thus, previous fault data becomes invalid for the testing of modified software.

In fault based approaches, the test cases are assigned priority on the basis of their potential to detect fault on the basis of availability of fault data by testing of original software. Those prioritized test cases may not work well for the modified software due to possibility of occurrence of new faults.

In coverage based approaches, only the number of entities (such as statements, branches, modules) covered are considered to prioritize test cases. No fault proneness is considered. As, there is a possibility that a test case covering less number of entities may be more vulnerable to faults due to some structural complexity and vice versa. Moreover; modification may cause the change of number of statements or branches in some cases. Thus the structure of the software can also affect the rate of fault detection. If the structural complexity is combined with any one of the coverage criteria for test case prioritization, it may prove a useful milestone in regression testing.

Chapter 3

Proposed Approach

From state of the art, we have observed that the existing prioritization techniques are either based upon coverage criteria or rate of fault severity. In case of coverage criteria based techniques, there exists a possibility that the test cases providing greater coverage are not structurally complex, but they are given higher priority based on coverage, thus the test cases covering complex part of the program may be given least priority. Another observation is that some existing techniques use fault severity as a criterion for test case prioritization. In these approaches, it is assumed that the data of fault coverage and fault severity already exists. There are only few techniques which take the complexity of program under consideration. Those techniques are based on modular level complexity. The techniques based on the complexity of the path covered by each test case lead to higher fault detection rate. We are going to propose test case prioritization approach based on path complexity combined with branch coverage.

Due to the use of path complexity of test cases rather than coverage, it is likely that path complexity based prioritization will produce better prioritization in terms of APFD as compared to only coverage based prioritization techniques [52].

3.1 Code Complexity Metrics

Complexity metrics help to locate complex portion of the code. Since past several years, computer scientists have been putting their efforts to measure the complexity of computer program. Program maintenance is affected by complexity therefore complexity measurement is of great significance in software maintenance [42]. There are many metrics used to measure software complexity, e.g., McCabe's Cyclomatic complexity number, Halstead's metric, Lines of code, Henry and Kafura's Information Flow Metric, McClure's Control Flow Metric, Woodfield's Syntactic Interconnection Measure etc. Among these metrics, McCabe's, Lines of code and Halstead's metrics are code metrics and the others are structure metrics [53]. Software complexity is an important feature which has been broadly discussed in literature. In software structure complexity measurement, Lines of Code (LOC), McCabe's Cyclomatic Complexity and Halstead's Volume have been commonly used [54].

3.1.1 Lines of Code Metric

The Lines of Code (LOC) metric was proposed around 1960. It was used for economic, productivity and quality studies. The simplest way to quantify the complexity of program is Lines of code. It literally counts the number of lines/text in a file of code. It is easy to count and understand [55]. LOC does not take intelligent content of the code into account. It characterizes only one aspect of the code, i.e., the length only. The functionality or complexity of the code is not taken into account. Moreover the complexity of the code may vary as different programmer may write the same code in different number of lines. Someone may write a very long and simple code and the same code may be implemented by someone else in complex but less number of lines. Thus it cannot be considered as a suitable measure of program complexity. Keeping above discussion in view, we do not consider LOC suitable for our prioritization approach.

3.1.2 Function Point (FP) Analysis

Function point analysis is an approach introduced by Allan Albrecht (1979) [56] to compute the size of a computerized business information system. The technical complexity factor takes technical and other factors during development and processing of information into account. Function point analysis was widely accepted for measuring the size of functional units, system development and enhancement [57]. The limitation of function point analysis is; it needs great effort to compute and only trained people can use this. Moreover, it is highly correlated to Lines of Code [58]. There exist no standards, only 35 different methods are there. Thus FP is not feasible for modern type of systems. FP analysis requires high cost and great effort to implement. It is time consuming method.

3.1.3 McCabe's Cyclomatic Complexity

Thomas McCabe introduced cyclomatic complexity v(G) in 1976. This metric is used to measure complexity of linearly-independent paths (control flow) of the code [59]. V(G) is number of conditional branches.

$$V(G) = e – n + p \tag{3.1}$$

Where
e: the number of edges,
n: the number of nodes and
p: connected components.

The program having sequential statements only will have v(G) = 1 because there exists only one path. McCabe's Cyclomatic Complexity v(G) does not consider unconditional branches like break-statements, goto statements, return statements etc. Although, these statements also increase the complexity of the code but McCabe's cyclomatic complexity metric overlooks these statements. McCabe's cyclomatic complexity is the measure of program's control complexity, not the

data complexity. Moreover, same weight is assigned to nested and non-nested loops whereas if there is increase in the nesting level, the complexity of the program is also increased. The deeply nested structures are difficult and complex to understand as compared to non-nested structures. Thus, McCabe's cyclomatic complexity metric is inadequate for measuring the complexity of the code [60]. Keeping these weaknesses of Cyclomatic complexity metric in view, we do not consider it suitable for our complexity based prioritization approach.

3.1.4 Halstead's Metric

Halstead's metric is structural complexity based metric. It was proposed by Maurice Halstead in his theory of software science [61]. Halstead's metric interprets the source code on the basis of tokens and classifies each token either as an operand or an operator. Halstead's metric measures following different properties of software:

n_1: number of unique operators

n_2: number of unique operands

N_1: total occurrences of operators

N_2: total occurrences of operands

All other measures of Halstead's metric are based on these four quantities with certain fixed formulas which will be explained later. In Halstead complexity metric, following are the operators and operands.

3.1.4.1 Operands

Following entities are treated as operands in Halstead's metric.

- All the identifiers are considered as operands in Halstead's metric.

All the TYPENAME

TYPESPEC (Type specifiers): Keywords used to specify the type are also operands e.g., char, long, int, float, bool, double, short, signed, unsigned, void etc.

CONSTANT: All the numeric, string or character constants.

3.1.4.2 Operators

Below are given operators of Halstead.

Storage class specifiers: All the keywords which are used to reserve storage space are considered as operators, i.e., register, typedef, inline, auto, extern, virtual, mutable etc.

Type qualifiers which qualify type like friend volatile, class etc.

Keywords: All other reserved words like if, else, enum, do, while, break, case, default, switch, struct, sizeof, union, namespace, const, true, false etc.

All operators e.g., ! , !=, % , && , —— , ( ) , *, *=, + , ++ , +=, , - , -- , -= , >, / , /= , <, <=, =, == , > , >= , ?, [ ] , ˆ , ˆ= , { } etc In case of control structures for(), if(), switch, while etc the colon and parentheses are treated as part of the structure and counted together.

Halstead's metric is program flow control based structure complexity metric, therefore, in this research; we are using Halstead's metric to measure the structure complexity of flow of test cases. The other measures of Halstead's metric are given below with their derivation formulas.

3.1.4.3 Size of The Vocabulary (n)

Program vocabulary is sum of total number of unique operators and total number of unique operands.

Size of vocabulary: $n = n_1 + n_2$

where n_1: number of unique operators

n_2: number of unique operands

3.1.4.4 Program Length (Program Size N)

Program vocabulary is used to measure program length N by following formula:

program length: $N = N_1 + N_2$

where N_1: total number of operators

N_2: total number of operands

Program length is also known as size of program. A very straight forward approach is to use program size for test case prioritization. As it is assumed that larger is the size, there is large number of faults or problems in the program. However, there are different opinions about considering the lines of code either all the statements including declaration statements or only executing statement should be considered. Size of the program measure can be used for test case prioritization.

3.1.4.5 Volume of Program (V)

Program volume is actually the information content of the software. It represents the space required to store the program. This parameter depends on specific algorithm implementation. Halstead computes program volume by following formula.

$$Volume = N log_2 n \tag{3.2}$$

Halstead's volume V is the size of implementation of an algorithm. It uses program vocabulary and size of the program.

3.1.4.6 Difficulty Level (D)

Difficulty is also known as error proneness of the program. Difficulty is proportional to the ratio of total number of operands to the total number of unique operands.

$$DifficultyD = (n_1/2) * (N_2/n2) \tag{3.3}$$

It means that if same operands are used multiple times in program, the program is more prone to faults. This is very important measure of Halstead's metric. In our research, we are going to use this measure i.e., error proneness or difficulty for test case prioritization.

3.1.4.7 Program Level (L)

Halstead's metric also measures the program level (L).

$$L = 1/D \tag{3.4}$$

Program level (L) is the inverse of fault proneness. It implies that a low level program is more vulnerable to faults than a high level program.

3.1.4.8 Effort to Implement (E)

Halstead's metric also measures effort (amount of mental activity) to implement and comprehend. Effort is considered good for software complexity measure. This measure is preferred in software maintenance. This measure can be used for test case prioritization as well. Effort is measured using following formula.

$$programmingeffort : E = V * D \tag{3.5}$$

3.1.4.9 Time to Implement (T)

Halstead's metric measures the time to implement the program. The time is directly proportional to effort. It is measured in seconds. Dividing the effort by 18 gives the approximate time to implement the program. This formula gives the predicted time to develop the software.

$$Estimatedprogrammingtime : T = E/18 \tag{3.6}$$

Where 18 is constant which is Stroud number S.

A psychologist John Stroud developed the concept of processing rate of human brain.

By him,a moment was defined as: "the time required by the human brain to carry out the most elementary decision".

Stroud number S = 18 moments / second

3.1.4.10 Estimated Program Length

This is the predicted program length based on vocabulary. It is also a result of Halstead metric.

Estimate of N is:

$$N\text{'} = n_1 log_2 n_1 + n_2 log_2 n_2 \tag{3.7}$$

3.1.4.11 Number of Delivered Bugs (B)

The overall complexity of the program is correlated with the Number of delivered bugs (B).

$$B = \frac{(E_3^2)}{3000} \tag{3.8}$$

It is an important measure for dynamic testing. The number of errors in the source code is approximately equal to the number of bugs delivered in the source code.

3.1.5 Why Halstead's Metric?

As in the previous discussion, it is observed that Halstead's metric is very comprehensive metric to measure different aspects of software. Halstead's metric does not require in-depth analysis of the code. It can measure the overall quality of the code. It is very simple to compute and can be used for any programming language[62].

In LOC, merely the number of lines of the code is taken in account. The LOC may vary depending upon the language used for writing the code. In case of programming languages allowing multiple statements per line and similar cases are not considered by LOC metric[62]. Whereas, in McCabe's complexity metric, it ignores the unconditional transfer of control statements like goto, break statements and return statements which also have considerable influence on complexity of the program [60]. FPA is also difficult and costly method to implement [58]. Therefore, Halstead's complexity Metric is better for measuring the software complexity. Keeping these important characteristics of Halstead's metric, we are using it in our research. The difficulty D (Error proneness) is used in our thesis to prioritize test cases.

In this research work, path complexity for each test case is calculated using Halstead function of our proposed approach. Basically, Halstead's metric is used to compute the complexity of complete program. In this thesis, we have made Halstead's function to compute the complexity of a path (extracted piece of code) executed by a particular test case. Halstead takes each path as input and calculates n, n_1, n_2, N, N_1, N_2, V, D, E, and T of the path executed by each test case using Halstead's metric. Among these calculated values, we will use Difficulty D for our test case prioritization.

For example: For following piece of code:

```
public static void sort(int p [])

{ for (int a=0; a < p.length-1; a++)

{ for (int b=a+1; b < x.length; b++)

{ if (p[a] > p[b])

{ int save=p[a]; p[a]=p[b]; p[b]=save

}

}

}

}
```

Using Halstead's metric:

Operators	**No. of occurrences**
public	1
sort( )	1
int	4
[ ]	7
{ }	4
for {; ;}	2
if ( )	1
=	5
¡	2
$n_1 = 17$	$N_1 = 39$

Operand	No of occurrences
p	9
length	2
a	7
b	6
save	2
0	1
1	2
$n_2 = 7$	$N_2 = 29$

For this example: N = 68, N ' = 89, V = 311.777, E = 10979.02, D = 35.21, L = 0.028, B = 0.165

3.2 Path Complexity Based Prioritization

In complexity based prioritization approach, the test cases are prioritized on the basis of path complexity. Path is actually the statements which are executed by a particular test case. The path complexity is calculated using Halstead's Metric. Each test case executes a particular path of the software under test (SUT), that path is extracted from the program and its complexity is calculated using Halstead's metric. Test case with the higher value of path complexity is given higher priority as compare to those with lower value of path complexity.

Proposed solution context diagram is shown in Figure 3.1

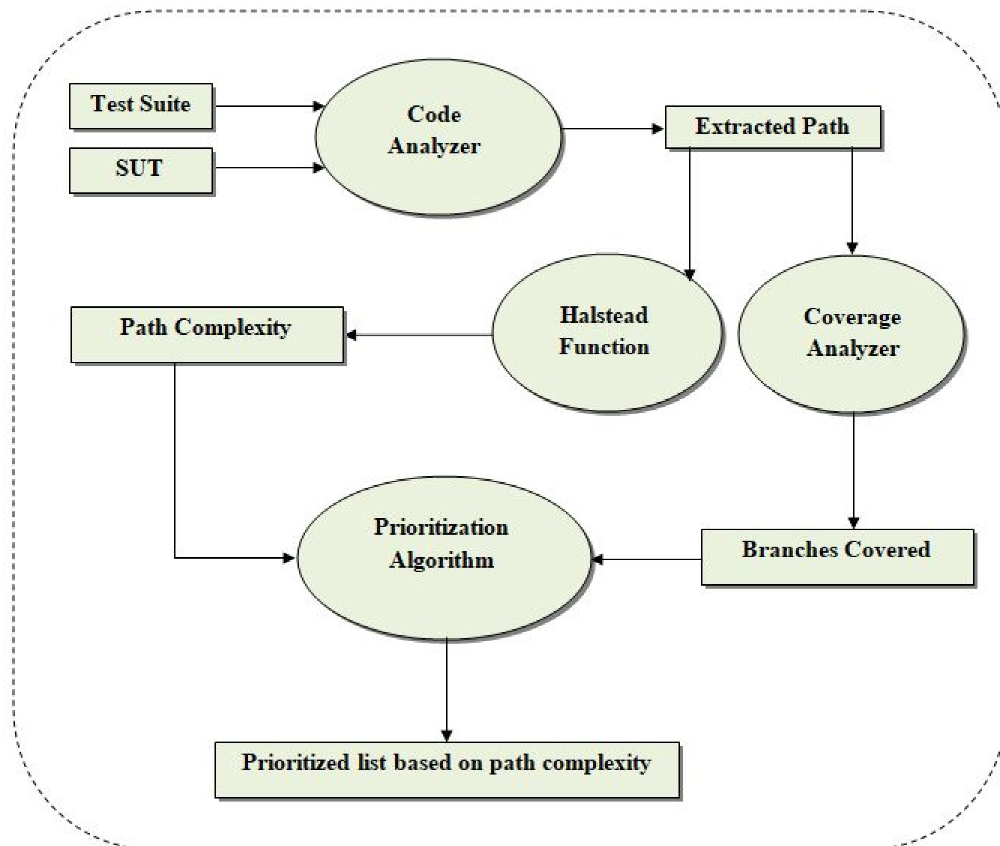

FIGURE 3.1: An illustration of proposed approach

Following are important steps involved in our approach to achieve goal.

- Path extraction for each test case.
- Calculation of path complexity.
- Applying prioritization algorithm to generate priority lists.

3.2.1 Path Extraction

The basic function of code analyzer is to use a defined unit of measurement to asses a code-base. In our approach, we give source code and test suite as input to our algorithm. The code analyzer module of the algorithm takes each test case from the test suite and executes it on software under test (SUT). Code analyzer extracts that path (piece of code) which is executed by the particular test case and saves the extracted code into a file. It takes each test case one by one, executes it on the source code, extracts path for that particular test case and stores it into a separate file. Later on, the extracted path is used by Halstead function of the algorithm.

3.2.2 Calculation of Path Complexity

Halstead's function is used to calculate the path complexity for each test case. It takes the path as input and calculates the number of unique operators, number of unique operands, total number of occurrences of operators and total number of operands. These measures are used to calculate the error proneness(difficulty) i.e. path complexity of the extracted path.

3.2.3 Applying Prioritization Algorithm

After collecting the above mentioned data, it will be used as input for prioritization algorithm. The algorithm generates priority list of test cases based on path

complexity and branch coverage as well. In first step, the algorithm will follow the greedy approach and find the test case t which has the highest path complexity N(t).

In next step the selected test case will be appended to complexity based priority list and removed from test suite. It will not further be considered. If there is a tie on complexity, then number of branches covered $Cov(t)$ is considered. The test case covering greater number of branches not covered so far is assigned higher priority. In case of multiple test cases with same number of branches covered, random selection is made. The process is repeated until no test case remains un-prioritized in the test suite.

The algorithm 1 is given below:

Algorithm 1 Procedure for prioritizing regression tests

Input: T : Set of Regression tests for P, N : Set of Path Complexity value for each test case in T, Cov : Set of entities covered by executing P against t, X' : A temporary set of regression tests for calculations

Output: PrT : A sequence of tests based on path complexity N

```
1:  X' = T
2:  while X' ≠ ∅ & Cov≠ ∅ do
3:      for all dot ∈ X'
4:          if N(t)>N(u) then
5:              PrT = PrT + t
6:              X' = X' − {t}
7:          else if N(t)=N(u) then
8:              if Cov(t) ≥ Cov(u) then
9:                  PrT = PrT + t
10:                 X' = X' − {t}
11:             else
12:                 PrT = PrT + t
13:             end if
14:         end if
15:     end for
16: end while
```

3.3 Example

To further elaborate the algorithm, consider the information given as an input to the path complexity based prioritization algorithm:

T = {T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13}

TABLE 3.1: Mapping of the test case to entities covered and path complexity

Test Cases	**Branches Covered**	**No. of Br. Covered**	**Path Complexity**
T1	2,4,6,7,9	5	4
T2	2,4,6,7,10,12,14	7	4
T3	2,4,6,7,10,12,14	7	4
T4	1,4,6,8,15,18	6	6
T5	2,3,6,8,16,19,22	7	6
T6	2,4,6,7,9	5	4
T7	1,4,6,8,15,18	6	6
T8	2,4,6,7,10,12,14	7	4
T9	2,4,6,7,10,12,14	7	4
T10	2,3,6,8,16,19,22	7	6
T11	2,4,5,8,16,20,23,26	8	9
T12	2,4,6,7,9	5	4
T13	2,4,6,7,9	5	4

Now, we will explain this example for our proposed path complexity based prioritization algorithm. Initially test case T11 is selected by considering the path complexity since it has maximum value of complexity measure. The next selection is T5 with second highest path complexity. Test cases with same path complexity are selected randomly. Following the same procedure, we prioritize all test cases and final prioritization list becomes:

PrT : { T11, T5, T10, T4, T2, T8, T3, T1, T7, T12, T9, T13, T6 }

Same test cases will be prioritized on the basis of branches covered. The branches covered by each test case are recorded. Test case achieving maximum branch coverage is assigned the highest priority. Therefore, T11 is selected since it covers maximum entities. In next step, the test case covering second highest number of entities. So, T2 is selected. In case, there exist mote than one test cases which have same number of branch coverage, one test case is selected randomly for that branch coverage. Considering this definition T3, T4, T5, T8 and T9 are selected randomly. By repeating the same process, we will prioritize all the test cases. Final prioritization list becomes as follows:

PrT : {T11, T2, T3, T5, T8, T9, T10, T4, T7, T1, T6, T12, T13}

In the end, we will compare our prioritization technique with already existing coverage based, white box prioritization technique.

Chapter 4

Implementation

This chapter includes implementation details of our proposed approach. In order to automate the process, we have developed a tool in Python 2.7.14 and Windows 10 operating system. Python is very powerful and easy to learn programming language. Python has high level and efficient data structures which help to develop applications for any platform. Our tool has three main components; the first component takes test cases and source code as inputs and executes test cases on source code; this component consists of code analyzer, it extracts the path executed by each test case and stores it in a separate text file. The second component is based on Halstead's function. It calculates different measures of Halstead e.g., Number of operators n1 and operands n2 , length of the program N, Effort E to implement the program, Volume V of the program, Difficulty D or error proneness and Estimated Time T. The third component of our tool prioritizes test cases on the basis of difficulty D and generates an ordered list of test cases. The other list is generated on the basis of branch coverage by using 'total' strategy.

Test cases are generated using a combination of Equivalence Class Partition (ECP) and worst case Boundary Value Analysis (BVA).

The architecture of tool is given below in Figure 4.1

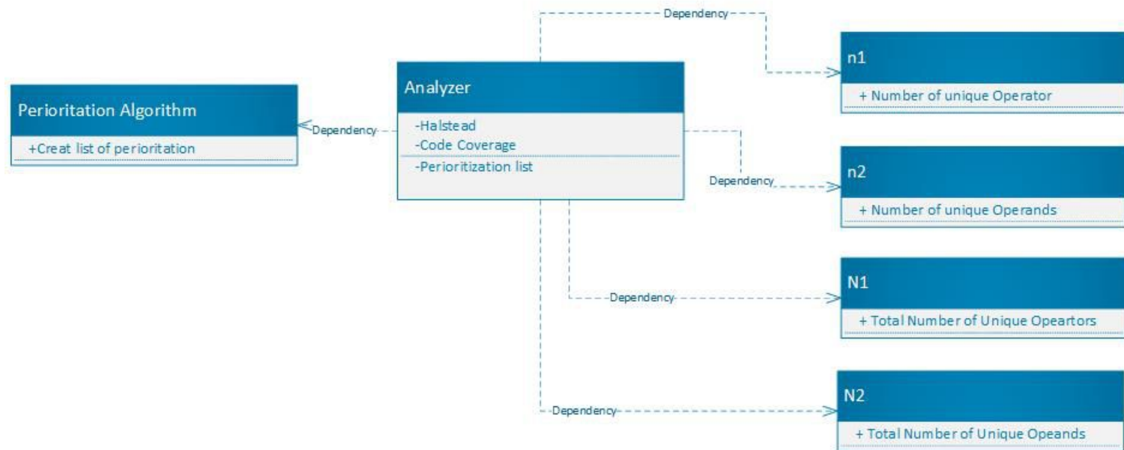

FIGURE 4.1: Architecture diagram of tool

4.1 Implementation Details

Here is given the complete detail of the implementation of our proposed approach.

the tool is developed by using python. it consists of three components. the first component is test case generation. There are many approaches to generate test cases, i.e., Equivalence Class Partitioning (ECP), Boundary value Analysis (BVA), Decision Table etc [63].

4.1.1 Equivalence Class Partitioning

In ECP, it is assumed that program's input and output can be classified or partitioned into multiple finite number of classes. some of them are valid (for valid input and output), some are invalid (for dealing exceptions). Thus; for each partition, only one test is required.In this way, the number of test cases to achieve the goal of testing is reduced.

ECP enables the tester to cover large domain of input and output by using a limited and smaller subset chosen from an equivalence class. another advantage of ECP is: it enables testers to select a subset of test cases with high chances of identifying defects [63].

4.1.2 Boundary Value Analysis

Boundary value analysis uses edges of input and output classes to generate test case. Boundaries of equivalence class can also be used to derive test cases. The errors occurring at the boundaries of equivalence class is called boundary value analysis [63].

In our proposed approach, we are using a combination of ECP and BVA to generate the test cases for our source programs. First of all, we have identified the classes on the basis of input and output of the program. Later on, the test cases for each class are generated on the boundaries of the class by using boundary value analysis. The source code of software under test and test cases are passed as input to our tool. The first component; code analyzer of the tool, takes each test case one by one and executes it on the source code. The code analyzer function of the too extracts code (path) executed by the particular test case and saves it to a separate file.

After completion of the path extraction process, the extracted path is passed as input to the second component, The Halstead function. The Halstead function computes different metrics of the path (code) and the complexity value is calculated for each test case. The value of error proneness (difficulty) is used as path complexity to prioritized the test cases.

The third component of the tool is mainly involved in prioritization of test cases according to our proposed approach. The test cases' complexity value and code coverage is given as input to the prioritization algorithm. Algorithm gives the higher priority to the test case with higher complexity value. If there are multiple test cases having same complexity measure, the test case which covers the entities not covered yet is chosen by following the additional strategy. In our proposed approached, if there is a tie among multiple test cases on code coverage, then test case is selected randomly. Finally, the prioritized list is generated.

4.2 User Interface

Here is the detail given how our proposed system works.

4.2.1 Test Case Generation

First of all, the test cases are generated using a combination of equivalence class partitioning (ECP) and boundary value analysis (BVA). The test case generation is automated by using mutation testing tool developed by Hassaan Minhas and Mubashir Kaleem, students of BS (CS). First of all, for generating test suite, system takes the min and max values for each variable from user through a user interface which is shown in Figure 4.2.

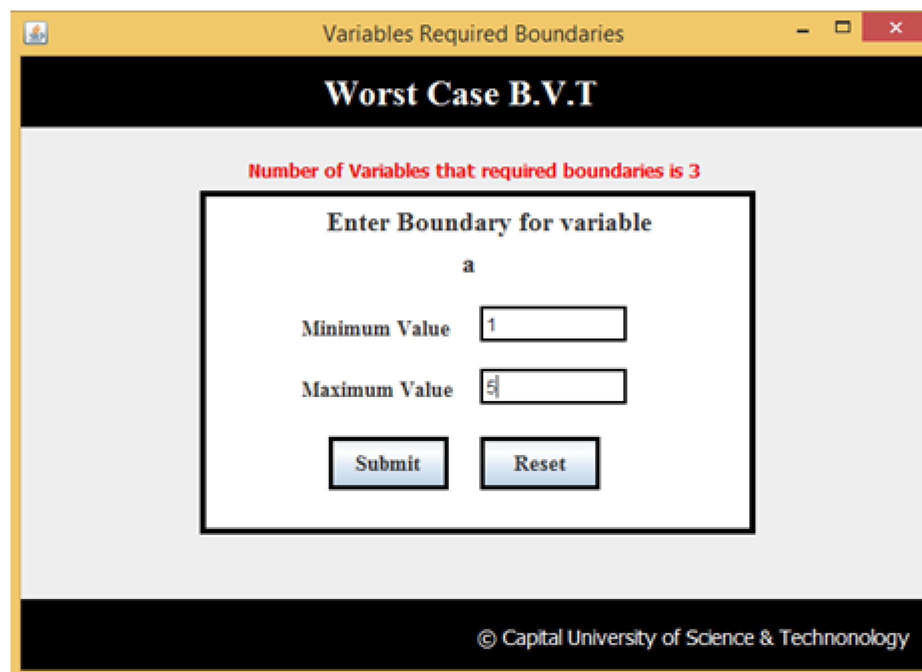

FIGURE 4.2: Taking Boundary values

The boundary values for the variables in the source code are analyzed and system asks for confirmation of boundary values as shown in the figure 4.3.

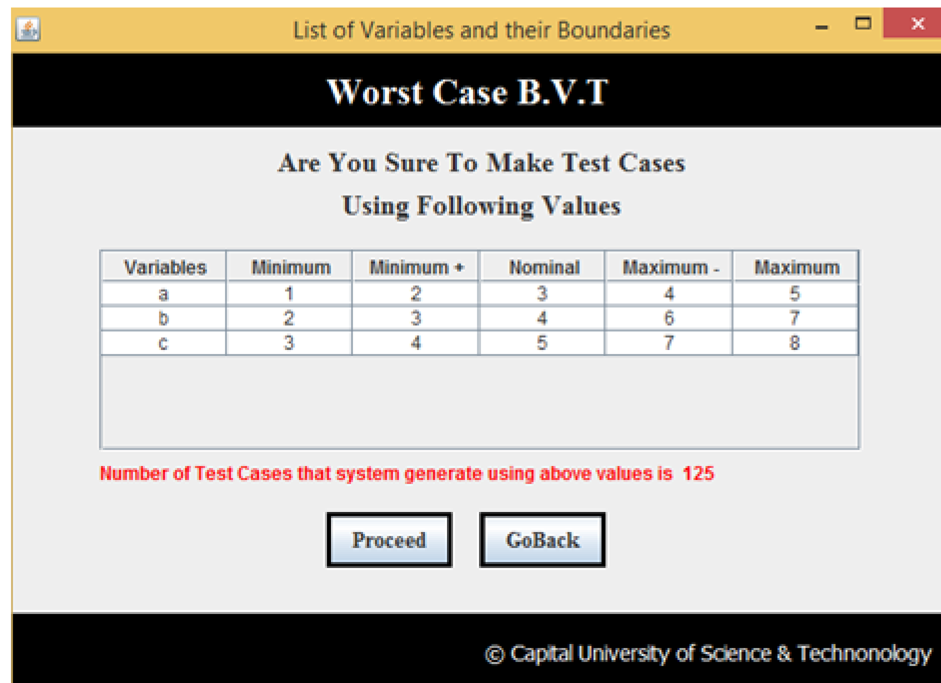

FIGURE 4.3: Displaying all five values

After confirmation from the user, by using worst case boundary value analysis, test cases are generated. Worst case boundary value analysis uses all the possible combinations of boundary values to generate test cases.

The system will display test cases with an option of test cases execution as shown in Figure 4.4.

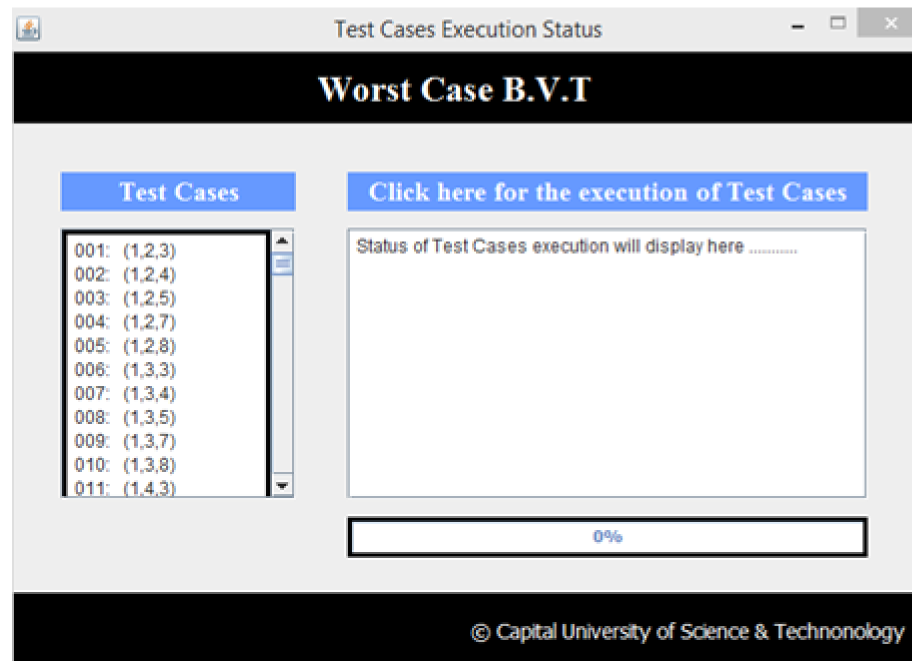

FIGURE 4.4: Displaying test cases

The test cases are copied and stored into a text file and later on along with the source code are passed as input to the algorithm.

4.2.2 Path Extraction

The proposed algorithm reads test cases one by one and executes on the source code as shown in the figure 4.5.

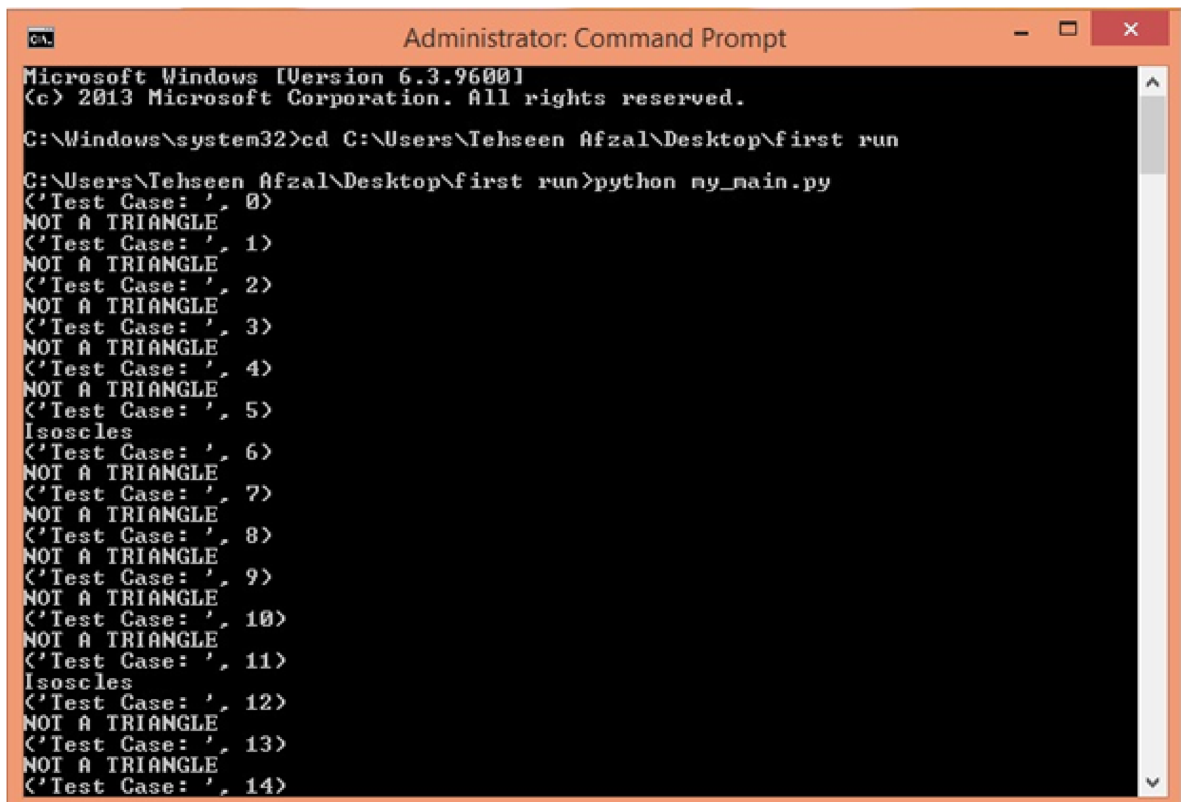

FIGURE 4.5: Test Case execution

The code analyzer analyses the path (code) followed by the particular test case, extracts the path and stores in a separate text file. For each test case, extracted path is stored in a separate file.

4.2.3 Path Complexity Calculation

The Halstead function of the algorithm takes the extracted path as input and calculates the path complexity for each test case. Normally, Halstead's metric is used to calculate the complexity of the whole source code but in this research thesis, we have made it to calculate the complexity of a path (a small chunk of code) executed by a particular test case. Following figure 4.6 shows the Halstead measures for each test case about a particular path. These measures are stored in file with .csv extension.

Test case	a	b	c	n1	N1	n2	N2	N	n	V	D	E	T
T1	1	2	3	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T2	1	2	4	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T3	1	2	5	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T4	1	2	7	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T5	1	2	8	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T6	1	3	3	13	51	10	31	82	23	370.9321	18	6676.777	370.9321
T7	1	3	4	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T8	1	3	5	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T9	1	3	7	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T10	1	3	8	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T11	1	4	3	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T12	1	4	4	13	51	10	31	82	23	370.9321	18	6676.777	370.9321
T13	1	4	5	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T14	1	4	7	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T15	1	4	8	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T16	1	6	3	11	33	9	18	51	20	220.4183	10	2204.183	122.4546
T17	1	6	4	11	33	9	18	51	20	220.4183	10	2204.183	122.4546

FIGURE 4.6: Halstead measures for test suite

4.2.4 Test Case Prioritization

The prioritization algorithm, takes error proneness (difficulty D) to prioritize test cases. The test cases with higher value of D are given higher priority and placed earlier in the prioritized list. For the test cases having same value of D, branch coverage is taken into account. Then the test case covering additional branches (not covered so far) is selected. If still tie remains there on coverage, test cases are chosen randomly.

Chapter 5

Results and Discussion

In this chapter we have discussed the results of our experiments which we have performed on subject programs. by using Halstead metric, we have calculated the path complexity of test cases for the respective test suites of subject programs. We have generated path complexity based prioritization list for each program. The existing criterion which we have used for comparison is total branch coverage. Both the techniques, i.e., path complexity and branch coverage based prioritization and additional branch coverage based prioritization are compared by using APFD.

For the evaluation of our technique, we have used three different programs. Our proposed approach is applied on method level. Methods usually consist of few lines of code. Therefore, the case studies used for evaluation are not very large in terms of lines of code. Source code of the program and test suite are given as input to our tool which executes each test case on the source code, extracts the path executed by particular test case and calculates the path complexity for each test case. The data generated is then given to prioritization algorithm for generating prioritization list. Branch coverage information is also collected for each program using its respective test suite and branch coverage based prioritization lists are also generated.

5.1 Subject programs

We have used a Simple Calculator program, Quadratic Equation problem and Date problem as subject programs to evaluate our approach. A brief description of each program is given below.

5.1.1 Simple Calculator Program:

The source code of simple calculator is taken from MYCPLUS [1]. This program takes three inputs i.e., two operands and one operator.The operator is used to decide what type of arithmetic operation is to be performed on operands. There are different types of basic arithmetic operations like addition, subtraction, multiplication, division, power of one number raised to another given number etc.

5.1.2 Quadratic Equation Problem:

The source code of quadratic equation problem is taken from Sanfoundary [2]. It take s three inputs, i.e., a, b and c. On the basis of these inputs, the program calculates the roots of quadratic equation.

5.1.3 Triangle Problem:

Triangle program takes three input variable a, b and c which represents the sides of a triangle. The input variables must satisfy the following conditions:

C1. $a < b + c$

C2. $b < a + c$

C3. $c < a + b$

[1]http://www.mycplus.com
[2]http://www.sanfoundry.com

There are three types of triangle which are equilateral, Isosceles and Scalene. Triangle problem returns the type of the triangle on the basis of input variables values if the above conditions are satisfied. If these values do not meet any of the above conditions then the program returns Not a Triangle as an output.

The source codes of these programs are given in appendix A. Different characteristics of these subjects programs are given below in Table 5.1.

TABLE 5.1: Subject Programs summary

Program	**LOC**	**No. of inputs**	**No. of branches**
Simple Calculator Program	223	3	10
Quadratic Equation Problem	41	3	04
Triangle Problem	80	3	40

Path complexity is calculated for each program and two priority lists are generated. One is path complexity based and the other is branch coverage based using the data given in appendix B where each program's data is given in tabular form. First column gives the path complexity, the second column lists all the branches covered and the third column shows the total number of branches covered.

Table 5.2 shows both priority lists for all 3 programs.

Table 5.2: Subject Programs' Priority Lists

Program	Path complexity based priority list	Branch coverage based priority list
Quadratic Equation problem	[t31, t36, t41, t47, t61, t91, t1, t3, t5, t7, t11, t19, t26, t48, t27, t29, t30, t40, t60, t75, t42, t66, t111, t46, t2, t25, t6, t10, t18, t15, t37, t101, t28, t34, t52, t73, t114, t35, t55, t105, t100, t56, t96, t86, t71, t4, t8, t16, t21, t51, t32, t49, t69, t112, t89, t45, t90, t120, t72, t116, t12, t13, t24, t76, t38, t64, t107, t79, t67, t65, t110, t50, t121, t81, t20, t9, t54, t97, t57, t108, t88, t115, t70, t106, t23, t17, t77, t98, t122, t44, t59, t80, t95, t14, t117, t93, t68, t43, t85, t22, t99, t118, t94, t124, t84, t62, t82, t39, t78, t125, t83, t53, t113, t119, t109, t123, t58, t102, t104, t87, t74, t33, t103, t92, t63]	[t1, t5, t26, t30, t2, t3, t4, t6, t7, t8, t9, t10, t11, t12, t13, t14, t15, t16, t17, t18, t19, t20, t21, t22, t23, t24, t25, t27, t28, t29, t31, t32, t33, t34, t35, t36, t37, t38, t39, t40, t41, t42, t43, t44, t45, t46, t47, t48, t49, t50, t51, t52, t53, t54, t55, t56, t57, t58, t59, t60, t61, t62, t63, t64, t65, t66, t67, t68, t69, t70, t71, t72, t73, t74, t75, t76, t77, t78, t79, t80, t81, t82, t83, t84, t85, t86, t87, t88, t89, t90, t91, t92, t93, t94, t95, t96, t97, t98, t99, t100, t101, t102, t103, t104, t105, t106, t107, t108, t109, t110, t111, t112, t113, t114, t115, t116, t117, t118, t119, t120, t121, t122, t123, t124, t125]

Simple Calculator Program	[t25, t27, t31, t39, t75, t77, t81, t89, t131, t133, t1, t3, t7, t15, t105, t107, t111, t119, t129, t135, t137, t50, t52, t56, t64, t26, t30, t38, t37, t76, t80, t88, t104, t132, t2, t6, t14, t13, t106, t110, t118, t117, t130, t136, t140, t51, t55, t63, t62, t28, t36, t33, t47, t78, t86, t102, t92, t134, t4, t12, t9, t22, t108, t116, t113, t126, t138, t53, t61, t58, t71, t32, t48, t45, t82, t98, t84, t83, t8, t24, t21, t112, t128, t125, t139, t57, t73, t70, t40, t42, t43, t90, t93, t97, t16, t18, t23, t120, t122, t127, t65, t67, t72, t41, t35, t34, t79, t94, t85, t17, t10, t121, t114, t66, t59, t46, t49, t100, t87, t101, t19, t5, t123, t109, t68, t54, t44, t91, t95, t20, t124, t69, t29, t96, t103, t11, t115, t60, t99, t74]	[t1, t25, t50, t75, t99, t105, t127, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, t14, t15, t16, t17, t18, t19, t20, t21, t22, t23, t24, t26, t27, t28, t29, t30, t31, t32, t33, t34, t35, t36, t37, t38, t39, t40, t41, t42, t43, t44, t45, t46, t47, t48, t49, t51, t52, t53, t54, t55, t56, t57, t58, t59, t60, t61, t62, t63, t64, t65, t66, t67, t68, t69, t70, t71, t72, t73, t74, t76, t77, t78, t79, t80, t81, t82, t83, t84, t85, t86, t87, t88, t89, t90, t91, t92, t93, t94, t95, t96, t97, t98, t100, t101, t102, t103, t104, t106, t107, t108, t109, t110, t111, t112, t113, t114, t115, t116, t117, t118, t119, t120, t121, t122, t123, t124, t125, t126, t128, t129, t130, t131, t132, t133, t134, t135, t136, t137, t138, t139, t140]

Triangle problem	[t56, t6, t124, t87, t12, t123, t113, t118, t92, t71, t26, t27, t29, t59, t89, t122, t121, t119, t114, t46, t18, t22, t47, t75, t101, t11, t1, t5, t20, t64, t24, t62, t31, t49, t108, t82, t51, t66, t28, t58, t57, t72, t41, t48, t67, t43, t94, t16, t117, t2, t13, t53, t65, t37, t112, t97, t77, t30, t90, t76, t100, t111, t38, t83, t23, t42, t4, t35, t105, t81, t106, t61, t60, t88, t96, t21, t98, t120, t102, t73, t9, t3, t34, t74, t103, t86, t36, t78, t115, t52, t68, t125, t44, t91, t107, t25, t63, t95, t32, t69, t80, t99, t116, t109, t70, t17, t93, t14, t50, t19, t104, t84, t33, t15, t55, t39, t10, t45, t7, t79, t40, t110, t8, t85, t54]	[t6, t11, t26, t51, t1, t32, t71, t101, t2, t3, t4, t5, t7, t8, t9, t10, t12, t13, t14, t15, t16, t17, t18, t19, t20, t21, t22, t23, t24, t25, t27, t28, t29, t30, t31, t33, t34, t35, t36, t37, t38, t39, t40, t41, t42, t43, t44, t45, t46, t47, t48, t49, t50, t52, t53, t54, t55, t56, t57, t58, t59, t60, t61, t62, t63, t64, t65, t66, t67, t68, t69, t70, t72, t73, t74, t75, t76, t77, t78, t79, t80, t81, t82, t83, t84, t85, t86, t87, t88, t89, t90, t91, t92, t93, t94, t95, t96, t97, t98, t99, t100, t102, t103, t104, t105, t106, t107, t108, t109, t110, t111, t112, t113, t114, t115, t116, t117, t118, t119, t120, t121, t122, t123, t124, t125]

5.2 Comparison

Both of the prioritization techniques are compared by using APFD; which is standard criterion for evaluation of prioritization techniques. For APFD computation, faults are seeded in the original program to generate its mutants(i.e. the faulty

versions). The mutation faults are representative of real faults. Hand seeded faults can be problematic for validity of results [64].

In this research work, AORB (Arithmetic Operator Replacement Binary) mutation operator has been used to generate mutants of the original program. AORB replaces every occurrence of one of the arithmetic operators +, -, /, * and % with each of the remaining operators. We tried different combinations, tested with test cases and it was observed that the program is working properly and ensured the 100% coverage of the program codes by using AORB mutation operator.

Errors detected by each test case are shown in column 4 of each program's table given in appendix B. APFD is calculated by using following formula.

$$APFD = 1 - (TF_1 + \cdots \ldots .TF_m)/nm + 1/2n \tag{5.1}$$

Where T is the test suite containing n test cases and F is the set of m faults revealed by T. For prioritizing T', let TF_i be the order of the first test case that reveals the ith fault. For APFD calculation, only those faults which are detected by the test suite are considered and undetected faults are ignored.

For example, consider the Quadratic Equation problem where:

F= {1,2,3,4,5, 6, 7, 8, 9,10,12,13,14,15,16,17,18,19,20,21,22,23,24,26,27,28,29,30,31,32}

T(Path Complexity based): [t31, t36, t41, t47, t61, t91, t1, t3, t5, t7, t11, t19, t26, t48, t27, t29, t30, t40, t60, t75, t42, t66, t111, t46, t2, t25, t6, t10, t18, t15, t37, t101, t28, t34, t52, t73, t114, t35, t55, t105, t100, t56, t96, t86, t71, t4, t8, t16, t21, t51, t32, t49, t69, t112, t89, t45, t90, t120, t72, t116, t12, t13, t24, t76, t38, t64, t107, t79, t67, t65, t110, t50, t121, t81, t20, t9, t54, t97, t57, t108, t88, t115, t70, t106, t23, t17, t77, t98, t122, t44, t59, t80, t95, t14, t117, t93, t68, t43, t85, t22, t99, t118, t94, t124, t84, t62, t82, t39, t78, t125, t83, t53, t113, t119, t109, t123, t58, t102, t104, t87, t74, t33, t103, t92, t63]

n=125, m=30

$$\text{APFD} = 1 - \frac{1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+2+2+3+4+5+......+14+59}{125(30)} + \frac{1}{2(125)}$$

$$\text{APFD} = 1\text{-}\frac{148}{3750} + \frac{1}{250}$$

$$\text{APFD} = 96.50\%$$

Using this data APFD for path complexity based prioritization for Quadratic equation problem is calculated.

APFDs of all three programs for both prioritization techniques are given below in 5.3.

TABLE 5.3: Subject Programs' APFD

Program	No. of test cases	No. of faults seeded	No. of faults detected	APFD for Path complexity based prioritization	APFD for branch coverage prioritization
Quadratic Equation Problem	125	32	30	96.50%	87.40%
Simple Calculator Problem	140	24	24	96.10%	52.80%
Triangle Problem	125	48	36	95.30%	92.51%

For Quadratic Equation problem, there is a remarkable difference of 9.10% between APFD of both priority lists. The test case t31 detects maximum faults (i.e. 18 faults) out of 30 faults which is prioritized at position 1 due to highest path complexity value, where as the same test case t31 is placed at position 10 in branch coverage prioritization which results in remarkable difference of APFD in both prioritization techniques. The difference between APFD shows that path

complexity based prioritization technique can detect faults earlier as compared to simple branch coverage based prioritization. The graphical representation of fault detection of test cases for Quadratic Equation problem is given below in Figure 5.1

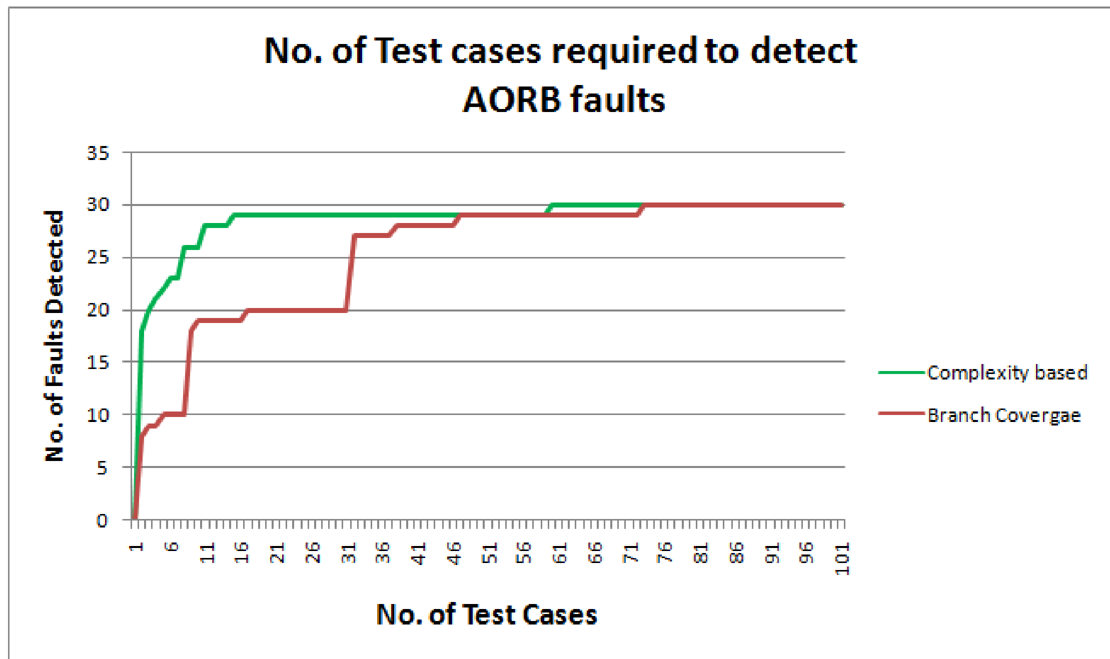

FIGURE 5.1: Graphical representation of fault detection of test cases for Quadratic Equation Problem

In simple calculator problem, the number of mutants are less than the other two example programs. Only two test cases are sufficient to kill all mutants. The test case t131, which covers 50% faults is positioned at rank 10 in path complexity based priority list, where as the same test case i.e. t131 is placed at position 131 in branch coverage priority list. this considerable difference in priority list position of test case results in a big difference of 43.30% in APFD of both priority lists. The graphical representation of fault detection of test cases for Simple Calculator Problem in Figure 5.2

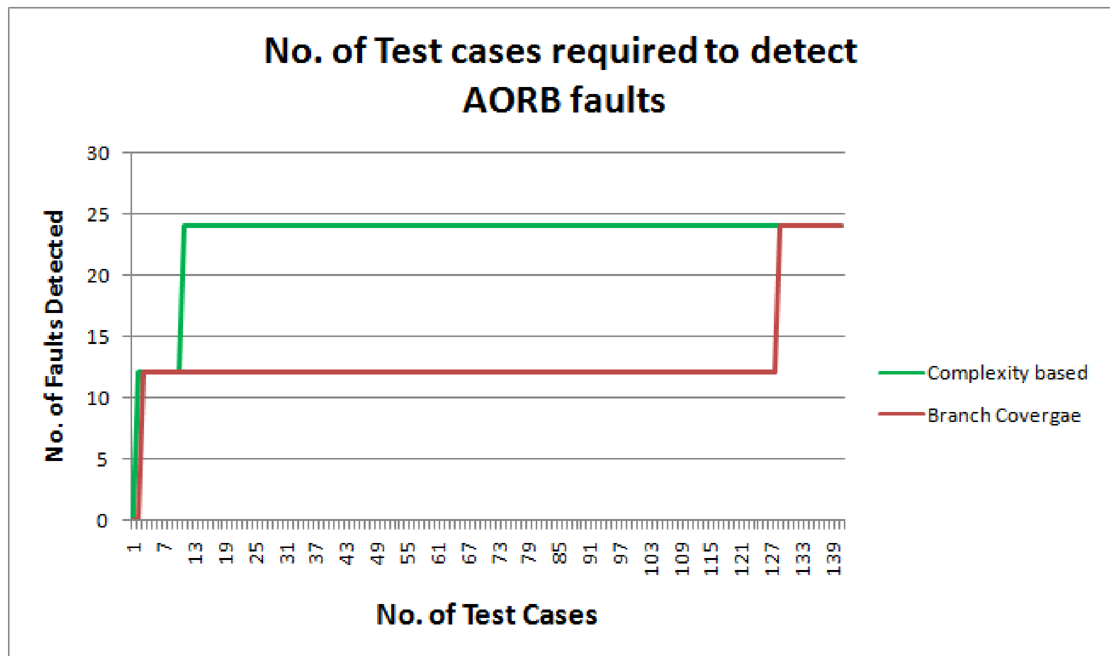

FIGURE 5.2: Graphical representation of fault detection of test cases for Simple Calculator Problem

For triangle problem, the difference between APFDs is only 2.79% because both priority lists include almost identical test cases with their positions varying.In path complexity based priority list the test case t56 is given the highest priority because of its highest fault exposing potential, but in branch based priority list t56 is on 58th position and out of 36 errors 8 are detected by t56. The graphical representation of fault detection of test cases for triangle problem is given below in Figure 5.3

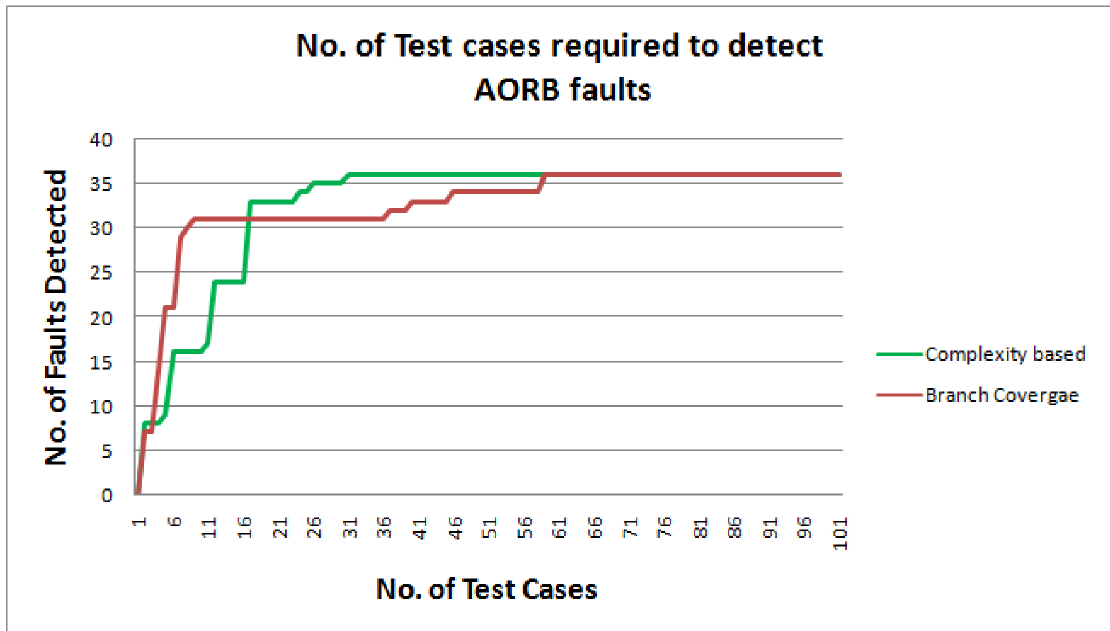

FIGURE 5.3: Graphical representation of fault detection of test cases for Triangle Problem

Following figure 5.4 explains the Graphical representation of APFDs of Subject Programs.

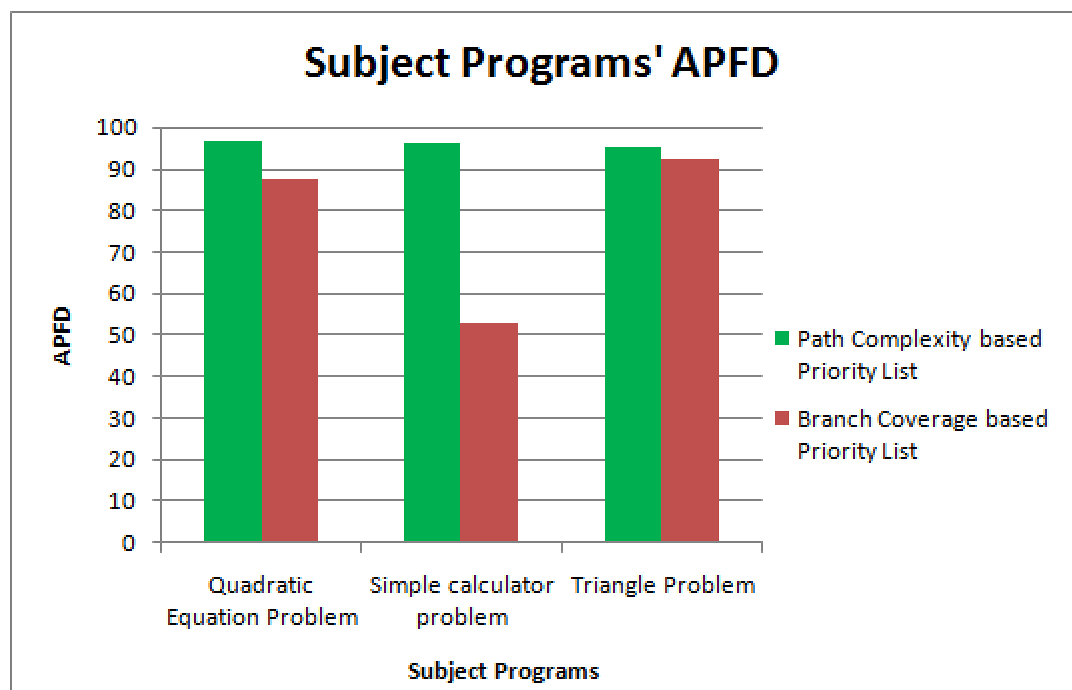

FIGURE 5.4: Graphical Representation of APFD

By comparing the APFD of path complexity based prioritization with branch coverage based prioritization, it can be concluded that it is higher than the latter.

In coverage based prioritization, it is assumed that the coverage will maximize the fault detection rate, but from the above results and comparison, it can be seen that this assumptions does not always hold. The test cases which have higher fault detection potential can be given low priorities or they can be treated as redundant test cases. The test case covering maximum entities, may not be structurally complex. Whereas; a test case covering less entities may be structurally more complex and can cover maximum faults. Since int the path complexity based prioritization, test cases are assigned priorities on the basis of their path complexity, thus it performs better than coverage based prioritization techniques in terms of APFD.

Chapter 6

Conclusion and Future Work

After reviewing literature, it has been concluded that test case prioritization is of considerable importance because it decreases the cost of regression testing. A lot of work has been done in this field and many researchers have contributed their efforts to achieve improved results. To overcome the drawbacks of other regression techniques, test case prioritization is more commonly used.

A literature survey is conducted through which we identify the existing and most commonly used techniques. A large number of white box and black box prioritization techniques have been proposed to find out the solution to the problem of test case prioritization. White box techniques use source code to for test case prioritization and black box techniques are specification based approaches to prioritize test cases. It is also observed that white box prioritization techniques, specifically those proposed by Rothermel et al., [24] including branch, statement and function coverage based, are more commonly used than black box prioritization. Many black box techniques are also very effective and giving improved results. the APFD comparison shows that the difference between white box and black box technique's APFD ranges from 2% to 5% , yet they are not commonly used due to lack of availability of structural information. In general, white box prioritization approaches outperform black box prioritization approaches in 50 to 60% cases. Through the detailed literature survey and experimentation, we are able to answer our research questions described in Chapter 1 as follows:

RQ. 1: What are gaps in existing white box prioritization techniques?

Though existing white box prioritization approaches perform well, but most of these approaches are not fault based. They order the test cases on the basis of some coverage criteria with assumption that the test case covering maximum entities can detect maximum faults. moreover, it is assumed that the test case with high coverage is more proned to faults but this assumption not always hold. The test case covering maximum entities may consist of simple I/O statements whereas; the one with less coverage may contain structurally complex statements and high fault proneness. Very few prioritization approaches exist which are fault based or complexity based. Thus, the structural complexity matter a lot in prioritization.

RQ. 2: How well does the proposed prioritization technique compare with the well studied white box prioritization techniques with respect to fault detection rate?

Fault based prioritization techniques normally assume that the fault data is available, very few techniques in literature use some criteria to calculate fault proneness of any module or entity. Moreover, when the software undergoes some modification, test suite changes and thus the fault data may also change. Existing fault data may not be valid for modified software thus this may affect the prioritization process. Fault based approaches still perform better than coverage based prioritization approaches.

The technique used in this research is simply based on path complexity. Here the path means the statements executed by a particular test case. The path complexity has been calculated using Halstead,s complexity metric. The test case having high value of path complexity is assigned higher priority. The proposed approach breaks the tie among test cases with same path complexity by using branch coverage as secondary criterion. Thus, the use of path complexity along with branch coverage has provided better rate of fault detection. Using three different subject programs, we have generated priority lists for both path complexity based and branch coverage. We have concluded that the branch coverage based approach has assigned lower priority to the test cases with higher value of

path complexity which is the major drawback of branch coverage approach. The proposed approach based on path complexity has addressed this drawback and assigned higher priorities to the test cases with higher value of path complexity (i.e. structurally complex test cases). The main objective of the proposed technique was to increase the average percentage of fault detection (APFD) of test suite. After generating the complexity based prioritization list and comparing it with the existing approach, we have observed that the APFD of our proposed technique is more than the strongest coverage based prioritization technique which is branch coverage [25], [38]. The difference between the APFDs of branch coverage and complexity based coverage ranges from 2.5 to 42%.

6.1 Future Work

After successful experiments of the proposed technique, we plan to use a combination of complexity metrics for prioritization of test suites in near future. Use of more than one complexity metric may help in increase rate of fault detection. We also plan to perform experiments with larger case studies.

Bibliography

[1] I. Burnstein, *Practical software testing: a process-oriented approach*, Springer, Ed. Springer Science & Business Media, 2006.

[2] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," in *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, vol. 1. IEEE, 2006, pp. 411–420.

[3] R. Feldt, S. Poulding, D. Clark, and S. Yoo, *Test set diameter: Quantifying the diversity of sets of test cases*, 2016, pp. 223–233.

[4] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[5] F. Spaven, "Cost of software errors: How much could software errors be costing your company?" raygun.com/blog/cost-of-software-errors/, Mar. 2017.

[6] S. G. Singh S and S. S., "Software testing." *International Journal of Advanced Research in Computer Science*, no. 1(3), 2010.

[7] B. L. P. M., "An introduction to software testing," *Electronic Notes in Theoretical Computer Science*, vol. 148, pp. 89–111, 2006.

[8] S. Quadri and S. Farooq, "Software testing: Goals, principles and limitations." *International Journal of Computer Applications,*, pp. 7–10, 2010.

[9] D. Jeffrey and N. Gupta, "Experiments with test case prioritization using relevant slices," *Journal of Systems and Software*, vol. 81, no. 2, pp. 196–221, 2008.

[10] P. S. Hooda A and K. S., "Regression testing: A complete overview." *International Journal of Advanced Research in Computer Science and Software Engineering,*, no. 5(5), 2015.

[11] R. Pradeepa and K. VimalDevi, "Effectiveness of testcase prioritization using apfd metric: Survey," in *IJCA Proceedings on International Conference on Research Trends in Computer Technologies*, 2013, pp. 1–4.

[12] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on software engineering.* ACM, 2002, pp. 119–129.

[13] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[14] H. K. Leung and L. White, "Insights into regression testing (software testing)," in *Software Maintenance, 1989., Proceedings., Conference on.* IEEE, 1989, pp. 60–69.

[15] M. R. Garey, "A guide to the theory of np-completeness," *Computers and intractability*, 1979.

[16] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135–141, 1996.

[17] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 3, pp. 270–285, 1993.

[18] J. R. Horgan and S. London, "A data flow coverage testing tool for c," in *Assessment of Quality Software Development Tools, 1992., Proceedings of the Second Symposium on.* IEEE, 1992, pp. 2–10.

[19] J. Pan and L. T. Center, "Procedures for reducing the size of coverage-based test sets," in *Proceedings of International Conference on Testing Computer Software*, 1995.

[20] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[21] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*. IEEE, 1997, pp. 264–274.

[22] M. J. Harrold, "Testing evolving software1," *Journal of Systems and Software*, vol. 47, no. 2-3, pp. 173–181, 1999.

[23] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 179–188.

[24] Rothermel and C. C. H. M. J. Gregg Untch, Roland H., "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[25] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 523–534.

[26] M. E. Khan, F. Khan *et al.*, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl*, vol. 3, no. 6, 2012.

[27] R. C. Bryce and A. M. Memon, "Test suite prioritization by interaction coverage," in *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 2007, pp. 1–7.

[28] M. J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 312–321.

[29] B. Qu, C. Nie, B. Xu, and X. Zhang, "Test case prioritization for black box testing," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 1. Ieee, 2007, pp. 465–474.

[30] H. Kumar, V. Pal, and N. Chauhan, "A hierarchical system test case prioritization technique based on requirements," in *13th Annual International Software Testing Conference*, 2013, pp. 4–5.

[31] A. Kaur and S. Goyal, "A genetic algorithm for regression test case prioritization using code coverage," *International journal on computer science and engineering*, vol. 3, no. 5, pp. 1839–1847, 2011.

[32] R. U. Maheswari and D. J. Mala, "Combined genetic and simulated annealing approach for test case prioritization," *Indian Journal of Science and Technology*, vol. 8, no. 35, pp. 1–5, 2015.

[33] A. Ansari, A. Khan, A. Khan, and K. Mukadam, "Optimized regression test using test case prioritization," *Procedia Computer Science*, vol. 79, pp. 152–160, 2016.

[34] M. Mann, "Generating and prioritizing optimal paths using ant colony optimization," *Computational Ecology and Software*, vol. 5, no. 1, pp. 1–15, 2015.

[35] H. Hemmati, Z. Fang, and M. V. Mantyla, "Prioritizing manual test cases in traditional and rapid release environments," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–10.

[36] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445.

[37] Z. Ma and J. Zhao, "Test case prioritization based on analysis of program structure," in *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*. IEEE, 2008, pp. 471–478.

[38] A. M. Sinaga, "Branch coverage based test case prioritization," *ARPN Journal of Engineering and Applied Sciences*, vol. 10, no. 3, pp. 1131–1137, 2015.

[39] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.

[40] T. Honglei, S. Wei, and Z. Yanan, "The research on software metrics and software complexity metrics," in *Computer Science-Technology and Applications, 2009. IFCSTA'09. International Forum on*, vol. 1. IEEE, 2009, pp. 131–136.

[41] T. M. Khoshgoftaar and J. C. Munson, "Predicting software development errors using software complexity metrics," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 2, pp. 253–261, 1990.

[42] K. Magel, R. M. Kluczny, W. A. Harrison, and A. R. Dekock, "Applying software complexity metrics to program maintenance," *IEEE*, pp. 65–79, 1982.

[43] N. Prakash and T. Rangaswamy, "Modular based multiple test case prioritization," in *Computational Intelligence & Computing Research (ICCIC), 2012 IEEE International Conference on*. IEEE, 2012, pp. 1–7.

[44] A. A. Ahmed, M. Shaheen, and E. Kosba, "Software testing suite prioritization using multi-criteria fitness function," in *Computer Theory and Applications (ICCTA), 2012 22nd International Conference on*. IEEE, 2012, pp. 160–166.

[45] M. Tyagi and S. Malhotra, "An approach for test case prioritization based on three factors," *International Journal of Information Technology and Computer Science (IJITCS)*, vol. 7, no. 4, p. 79, 2015.

[46] R. Kavitha and N. Sureshkumar, "Test case prioritization for regression testing based on severity of fault," *International Journal on Computer Science and Engineering*, vol. 2, no. 5, pp. 1462–1466, 2010.

[47] Y. Wang, X. Zhao, and X. Ding, "An effective test case prioritization method based on fault severity," in *Software Engineering and Service Science (ICSESS), 2015 6th IEEE International Conference on.* IEEE, 2015, pp. 737–741.

[48] E. L. Alves, P. D. Machado, T. Massoni, and M. Kim, "Prioritizing test cases for early detection of refactoring faults," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 402–426, 2016.

[49] S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, D. Sundmark, and S. Larsson, "Towards earlier fault detection by value-driven prioritization of test cases using fuzzy topsis," in *13th International Conference on Information Technology: New Generations ITNG 2016*, 2016.

[50] K. H. S. Hla, Y. Choi, and J. S. Park, "Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting," in *Computer and Information Technology Workshops, 2008. CIT Workshops 2008. IEEE 8th International Conference on.* IEEE, 2008, pp. 527–532.

[51] M. M. Navleen Kaur, "Regression testing with multiple criteria based test case prioritization." *ijraset*, 2017.

[52] P. Kaur, P. Bansal, and R. Sibal, "Prioritization of test scenarios derived from uml activity diagram using path complexity," in *Proceedings of the CUBE International Information Technology Conference.* ACM, 2012, pp. 355–359.

[53] D. Kafura and G. R. Reddy, "The use of software complexity metrics in software maintenance," *IEEE Transactions on Software Engineering*, no. 3, pp. 335–343, 1987.

[54] H. Zhang, X. Zhang, and M. Gu, "Predicting defective software components from code complexity measures," in *Dependable Computing, 2007. PRDC*

2007. 13th Pacific Rim International Symposium on. IEEE, 2007, pp. 93–96.

[55] T. M. Khoshgoftaar and J. C. Munson, "The lines of code metric as a predictor of program faults: A critical analysis," in *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International.* IEEE, 1990, pp. 408–413.

[56] A. J. Albrecht, "Measuring application development productivity," in *Proc. of the Joint SHARE/GUIDE/IBM Applicaiton Development Symposium*, 1979, pp. 83–92.

[57] C. R. Symons, "Function point analysis: difficulties and improvements," *IEEE transactions on software engineering*, vol. 14, no. 1, pp. 2–11, 1988.

[58] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: a software science validation," *IEEE transactions on software engineering*, no. 6, pp. 639–648, 1983.

[59] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[60] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, pp. 30–36, 1988.

[61] M. H. Halstead, *Elements of software science.* Elsevier New York, 1977, vol. 7.

[62] H. R. Bhatti, "Automatic measurement of source code complexity," 2011.

[63] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[64] H. Do and G. Rothermel, "A controlled experiment assessing test case prioritization techniques via mutation faults," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on.* IEEE, 2005, pp. 411–420.

Appendix A

Source code of Quadratic Equation problem:

```
import java.util.Scanner;
public class calculateQuadraticEqua{
public static void main(String[] args) {
double discr, root1, root2;
System.out.println(”Applying the quadratic formula”);
Scanner input = new Scanner(System.in);
double a = input.nextDouble();
double b = input.nextDouble();
double c = input.nextDouble();
// Solve the discriminant (SQRT (b^2 - 4ac)
discr = Math.sqrt((b * b) - (4 * a * c));
System.out.println(”Discriminant = ” + discr);
// Determine number of roots
// if discr> 0 equation has 2 real roots
// if discr == 0 equation has a repeated real root
// if discr< 0 equation has imaginary roots
// if discr is NaN equation has no roots
if(Double.isNaN(discr))
System.out.println(”Equation has no roots”);
if(discr> 0)
{
System.out.println(”Equation has 2 roots”);
root1 = (-b + discr)/2 * a;
```

```
root2 = (-b - discr)/2 * a;
System.out.println("First root = " + root1);
System.out.println("Second root = " + root2);
}
if(discr == 0)
{
System.out.println("Equation has 1 root");
root1 = (-b + discr)/2 * a;
System.out.println("Root = " + root1);
}
if(discr< 0)
System.out.println("Equation has imaginary roots");
}
}
```

Source code of Simple Calculator Program:

```
/****************************************************
*    MYCPLUS Sample Code - http://www.mycplus.com     *
*                                                     *
*     This code is made available as a service to our *
*        visitors and is provided strictly for the    *
*               purpose of illustration.              *
*                                                     *
*    Please direct all inquiries to saqib at mycplus.com *
****************************************************/
package calculator;
/**
* Title: Calculator
* Description: Calculator
* Copyright: Copyright (c) 2003
* Company: Nagina Computers
* @author Muhamnmad Saqib
* @version 1.0
*/
```

```
import java.io.*;
import java.math.*;
public class Calculator {
static double numAdd1=0,numAdd2=0;
static double numSub1=0,numSub2=0,numMul1=0;
static double numMul2=0,numDiv1=0,numDiv2=0;
static double numSqr1=0,numCube1=0,numPow1=0;
static double numPow2=0,numSqrt1=0;
static int choice;
static String myString;
//makes the full user interface at start up
public static int UI()throws Exception {
BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
System.out.println(”\ n\ n****************************\ nWel Come to Java Language”);
System.out.println(” CALCULATOR \ n******************************”);
System.out.println(”0:\ tEXIT()”);
System.out.println(”1:\ tAdd two Numbers”);
System.out.println(”2:\ tSubtract two Numbers”);
System.out.println(”3:\ tMultiply two Numbers”);
System.out.println(”4:\ tDivide two Numbers”);
System.out.println(”5:\ tSquare of a number”);
System.out.println(”6:\ tCube of a number”);
System.out.println(”7:\ tFind the SQUARE-ROOT of a Number”);
System.out.println(”8:\ tFind the X power Y”);
choice = Integer.parseInt(input.readLine());
return choice; }
//Calculate the Addition of two numbers
public static double add(double numAdd1, double numAdd2){
return numAdd1 + numAdd2;
}
//Calculate the subtraction of two numbers
public static double sub(double numSub1, double numSub2){
return numSub1 - numSub2;
```

```
}
//Calculate the multiplication of two numbers
public static double multiply(double numMul1, double numMul2){
return numMul1 + numMul2;
}
//Calculate the Division of two numbers
public static double divide(double numDiv1, double numDiv2){
return numDiv1 / numDiv2;
}
//Calculate the Square of a numbers
public static double square(double numSqr1){
return numSqr1*numSqr1;
}
//Calculate the Cube of a numbers
public static double cube(double numCube1){
return numCube1 * numCube1 * numCube1;
}
//Calculate the SQUARE-ROOT of a numbers
public static double squareRoot(double numSqrt1){
return Math.sqrt(numSqrt1);
}
//Calculate the power of numbers
public static double power(double numpow1, double numPow2){
return Math.pow(numPow1,numPow2);
}
//press any key to Goto Main Menu
public static void mainMenu(){
System.out.print("Press Enter key.....");
try {
System.in.read();
}
catch(IOException e){
```

```
return;
}
}
//Function to check the input validation
public static boolean checkInput(String str){
int stringLength = str.length();
if (stringLength
=300){ return false;}                              <
for (int i=0;i¡stringLength-1;i++)
if (str.charAt(i) ¡=0 —— str.charAt(i)
=9)                                                <
return false;
return true;
}
//main function
public static void main(String[] args)throws Exception {
boolean isValidInput;
BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
choice= UI();
while (choice!=0){
switch(choice){
case 0:
return;
case 1:
//Add two numbers code Code
System.out.println("\ nEnter First Number");
String str = input.readLine();
isValidInput = checkInput(str);
if(isValidInput==true){
numAdd1= Double.parseDouble(str);
}
else {System.out.println("\ n****************\ nInput ERROR\ n************** ");}
System.out.println("\ nEnter second Number");
```

```
str = input.readLine();
isValidInput = checkInput(str);
if(isValidInput==true){
numAdd2 = Double.parseDouble(str);
double numAddSum= add(numAdd1,numAdd2);
System.out.println("\ n**********\ nThe Sum is= " + numAddSum + "\ n********** ");
}
else{System.out.println("\ n***************\ nInput ERROR\ n************** ");}
mainMenu();
UI();
break;
case 2:
//Code
//subtract two numbers code Code
System.out.println("\ nEnter First Number");
numSub1= Double.parseDouble(input.readLine());
System.out.println("\ nEnter second Number");
numSub2 = Double.parseDouble(input.readLine());
double numSub = sub(numSub1,numSub2);
System.out.println("\ n**********\ nThe Difference is= " + numSub + "\ n********** ");
mainMenu();
UI();
break;
case 3:
//Code
//subtract two numbers code Code
System.out.println("\ nEnter First Number");
numMul1= Double.parseDouble(input.readLine());
System.out.println("\ nEnter second Number");
numMul2 = Double.parseDouble(input.readLine());
double numMul = multiply(numMul1,numMul2);
System.out.println("\ n**********\ nThe Multiplication is= " + numMul + "\ n**********
");
```

```
mainMenu();
UI();
break;
case 4:
//Code
//Divide two numbers code Code
System.out.println("\ nEnter First Number");
numDiv1= Double.parseDouble(input.readLine());
System.out.println("\ nEnter second Number");
numDiv2 = Double.parseDouble(input.readLine());
double numDiv = divide(numDiv1,numDiv2);
System.out.println("\ n**********\ nThe Division is= " + numDiv + "\ n********** ");
mainMenu();
UI();
break;
case 5:
//Code
//square of a number code
System.out.println("\ nEnter a Number");
numSqr1= Double.parseDouble(input.readLine());
double numSqr = square(numSqr1);
System.out.println("\ n**********\ nThe SQUARE is= " + numSqr + "\ n********** ");
mainMenu();
UI();
break;
case 6:
//Code
//cube of a number code
System.out.println("\ nEnter a Number");
numCube1= Double.parseDouble(input.readLine());
double numCube = cube(numCube1);
System.out.println("\ n**********\ nThe CUBE is= " + numCube + "\ n********** ");
mainMenu();
```

```
UI();
break;
case 7:
//Code
//square-root of a numbver
System.out.println("\ nEnter a Number");
numSqrt1= Double.parseDouble(input.readLine());
double numSqrt = squareRoot(numSqrt1);
System.out.println("\ n**********\ nThe SQUARE-ROOT is= " + numSqrt + "\ n**********
");
mainMenu();
UI();
break;
case 8:
//Code
//Divide two numbers code Code
System.out.println("\ nEnter a Number");
numPow1= Double.parseDouble(input.readLine());
System.out.println("\ nEnter a 2nd Number");
numPow2= Double.parseDouble(input.readLine());
double numPow = power(numPow1,numPow2);
System.out.println("\ n**********\ nThe " + numPow1 + " power " + numPow2 + " is= " +
numPow + "\ n********** ");
mainMenu();
UI();
break;
default:
UI();
break;
}
}
}
}
```

Source code of Triangle Program:

```
public class TriangleProbelm {
String newline = System.getProperty(”line.separator”);
public void triangle(inta,intb,int c){
int match=0,d,e;
if(a==b)
match = match - match + 1;
if(a==c)
match= match - match + 2;
if(b==c)
match= match - match + 3;
d=a+b;
e=b+c;
if(match==0){
if(d<=c)
System.out.print(”NOT A TRIANGLE” +newline);
else if(e<=a)
System.out.print(”NOT A TRIANGLE” +newline);
else if(a+c<=b)
System.out.print(”NOT A TRIANGLE” +newline);
else
System.out.println(”Scalane” +newline);
}
else if(match==1){
if(a+c<=b)
System.out.println(”NOT A TRIANGLE” +newline);}
else
System.out.println(”Isoscles” +newline);
}
else if (match==2){
if(a+c<=b)
System.out.print(”NOT A TRIANGLE” +newline);
else
System.out.print(”Isoscles” +newline);
```

```
}
else if(match==3){
if(b+c<=a)
System.out.print(”NOT A TRIANGLE” +newline);
else
System.out.print(”Isoscles” +newline);
}
else
System.out.println(”Equilaterial” +newline);
}
}
```

Appendix B

Test data for Quadratic Equation Problem

Test case	Test Case Value	Path Complexity	Branches covered	Cov(t) for branches	Faults detected (Mutants Killed)
t1	(0,0,0)	8	3	1	3,4,7,8,15,16,27,28
t2	(0,0,1)	8	3	1	3,4,7,8,14,27,28
t3	(0,0,4)	8	3	1	3,4,7,8,14,27,28
t4	(0,0,7)	8	4	1	3,4,7,8,14,27,28
t5	(0,0,8)	8	4	1	3,4,7,8,14,27,28
t6	(0,1,0)	8	4	1	2,4,5,6,7,8,15,16, 23,24,29,30,31,32
t7	(0,1,1)	8	4	1	2,4,5,6,7,8,13,23, 24,29,30,31,32
t8	(0,1,4)	8	4	1	2,4,5,6,7,8,13,14,23 ,24,29,30,31,32
t9	(0,1,7)	8	4	1	2,4,5,6,7,8,13,14,23, 24,29,30,31,32
t10	(0,1,8)	8	4	1	2,4,5,6,7,8,13,14,23, 24,29,30,31,32
t11	(0,2,0)	8	4	1	2,3,4,5,6,7,8,15,16,23, 24,29,30,31,32
t12	(0,2,1)	8	4	1	2,3,4,5,6,7,8,13,23, 24,29,30,31,32
t13	(0,2,4)	8	4	1	2,3,4,5,6,7,8,13,23, 24,29,30,31,32

t14	(0,2,7)	8	4	1	2,3,4,5,6,7,8,13,14,23, 24,29,30,31,32
t15	(0,2,8)	8	4	1	2,3,4,5,6,7,8,13,14,23, 24,29,30,31,32
t16	(0,3,0)	8	4	1	1,2,3,4,5,6,7,8,15,16,23, 24,29,30,31,32
t17	(0,3,1)	8	4	1	1,2,3,4,5,6,7,8,13,23, 24,29,30,31,32
t18	(0,3,4)	8	4	1	1,2,3,4,5,6,7,8,13,23, 24,29,30,31,32
t19	(0,3,7)	8	4	1	1,2,3,4,5,6,7,8,13,14,23, 24,29,30,31,32
t20	(0,3,8)	8	4	1	1,2,3,4,5,6,7,8,13,14,23, 24,29,30,31,32
t21	(0,4,0)	8	4	1	1,2,3,4,5,6,7,8,15,16,23, 24,29,30,31,32
t22	(0,4,1)	8	4	1	1,2,3,4,5,6,7,8,13,23,24, 29,30,31,32
t23t	(0,4,4)	8	4	1	1,2,3,4,5,6,7,8,13,23,24, 29,30,31,32
t24	(0,4,7)	8	4	1	1,2,3,4,5,6,7,8,13,23,24, 29,30,31,32
t25	(0,4,8)	8	3	1	1,2,3,4,5,6,7,8,13,23,24, 29,30,31,32
t26	(1,0,0)	6	1	1	3,4,15,16,27,28
t27	(1,0,1)	6	1	1	3,4,27,28
t28	(1,0,4)	6	1	1	3,4,27,28
t29	(1,0,7)	6	1	1	3,4,6,14,27,28
t30	(1,0,8)	14	2	1	3,4,6,14,27,28
t31	(1,1,0)	6	1	1	2,4,5,6,9,13,14,15,16,17,18,19,20, 21,22,29,30,32
t32	(1,1,1)	6	1	1	8,16,27,28
t33	(1,1,4)	6	1	1	6,8,14,16,27,28
t34	(1,1,7)	6	1	1	6,8,14,16,27,28
t35	(1,1,8)	14	2	1	6,8,14,16,27,28

t36	(1,2,0)	8	3	1	2,3,4,5,6,9,10,13,14,15,16,17,19,20, 21,22,29,30,32
t37	(1,2,1)	6	1	1	6,8,14,16,26,27,28
t38	(1,2,4)	6	1	1	6,8,14,15,16,26,27,28
t39	(1,2,7)	6	1	1	6,8,14,15,26,27,28
t40	(1,2,8)	14	2	1	6,8,14,15,26,27,28
t41	(1,3,0)	14	2	1	1,2,3,4,5,6,9,10,13,14,15,16, 17,18,19,20,21,22,29,30,32
t42	(1,3,1)	6	1	1	1,2,3,4,8,9,10,16,17,18,19, 20,21,22,29,30,32
t43	(1,3,4)	6	1	1	5,6,8,13,14,15,16,26,27,28
t44	(1,3,7)	6	1	1	6,8,14,15,16,26,27,28
t45	(1,3,8)	14	2	1	6,8,14,15,16,26,27,28
t46	(1,4,0)	14	2	1	1,2,3,4,5,6,9,10,13,14,15, 16,17,18,19,20,21,22,29,30,32
t47	(1,4,1)	8	2	1	1,2,3,4,8,9,10,16,17,18, 19,20,21,22,29,30,32
t48	(1,4,4)	6	1	1	5,6,8,13,14,15,16,26,27,28
t49	(1,4,7)	6	1	1	5,6,8,13,14,15,16,26,27,28
t50	(1,4,8)	8	2	1	5,6,8,13,14,15,16,26,27,28
t51	(3,0,0)	6	1	1	3,4,15,16,27,28
t52	(3,0,1)	6	1	1	3,4,27,28
t53	(3,0,4)	6	1	1	3,4,6,27,28
t54	(3,0,7)	6	1	1	3,4,6,27,28
t55	(3,0,8)	14	2	1	3,4,6,27,28
t56	(3,1,0)	6	1	1	2,4,5,6,9,13,14,15,16,17,18, 19,20,21,22,29,30,31,32
t57	(3,1,1)	6	1	1	16,27,28
t58	(3,1,4)	6	1	1	6,16,27,28
t59	(3,1,7)	6	1	1	6,27,28
t60	(3,1,8)	14	2	1	6,27,28
t61	(3,2,0)	6	1	1	2,3,4,5,6,9,10,13,14,15,16, 17,19,20,21,22,29,30,31,32
t62	(3,2,1)	6	1	1	6,7,8,16,26,27,28
t63	(3,2,4)	6	1	1	6,15,16,26,27,28

t64	(3,2,7)	6	1	1	6,15,26,27,28
t65	(3,2,8)	14	2	1	6,15,26,27,28
t66	(3,3,0)	6	1	1	1,2,3,4,5,6,9,10,13,14,15,16, 17,18,19,20,21,22,29,30,31,32
t67	(3,3,1)	6	1	1	5,6,7,8,16,26,27,28
t68	(3,3,4)	6	1	1	6,7,8,14,15,16,26,27,28
t69	(3,3,7)	6	1	1	6,7,8,14,15,16,26,27,28
t70	(3,3,8)	14	2	1	6,7,8,14,15,16,26,27,28
t71	(3,4,0)	14	1	1	1,2,3,4,5,6,9,10,13,14,15, 16,17,18,19,20,21,22,29,30,31,32
t72	(3,4,1)	6	1	1	1,2,3,4,5,6,7,8,9,10,12,13,16, 17,18,19,20,21,22,23,24,29,30,31,32
t73	(3,4,4)	6	1	1	6,7,8,14,15,16,26,27,28
t74	(3,4,7)	6	1	1	6,7,8,14,15,16,26,27,28
t75	(3,4,8)	8	3	1	6,7,8,14,15,16,26,27,28
t76	(5,0,0)	6	1	1	3,4,15,16,27,28
t77	(5,0,1)	6	1	1	3,4,6,27,28
t78	(5,0,4)	6	1	1	3,4,6,27,28
t79	(5,0,7)	6	1	1	3,4,6,27,28
t80	(5,0,8)	14	2	1	3,4,6,27,28
t81	(5,1,0)	6	1	1	2,4,5,6,9,13,14,15,16,17,18, 19,20,21,22,29,30,31,32
t82	(5,1,1)	6	1	1	6,7,16,27,28
t83	(5,1,4)	6	1	1	6,7,16,27,28
t84	(5,1,7)	6	1	1	6,7,27,28
t85	(5,1,8)	14	2	1	6,7,27,28
t86	(5,2,0)	6	1	1	2,3,4,5,6,9,10,13,14,15,16,17,19, 20,21,22,29,30,31,32
t87	(5,2,1)	6	1	1	6,7,16,26,27,28
t88	(5,2,4)	6	1	1	6,7,16,26,27,28
t89	(5,2,7)	6	1	1	6,7,16,26,27,28
t90	(5,2,8)	14	2	1	6,7,16,26,27,28
t91	(5,3,0)	6	1	1	1,2,3,4,5,6,9,10,13,14,15, 16,17,18,19,20,21,22,29,30,31,32
t92	(5,3,1)	6	1	1	6,7,8,16,26,27,28

t93	(5,3,4)	6	1	1	6,7,15,16,26,27,28
t94	(5,3,7)	6	1	1	6,7,15,16,26,27,28
t95	(5,3,8)	14	2	1	6,7,15,16,26,27,28
t96	(5,4,0)	6	1	1	1,2,3,4,5,6,9,10,13,14,15, 16,17,18,19,20,21,22,29,30,31,32
t97	(5,4,1)	6	1	1	5,6,7,8,16,26,27,28
t98	(5,4,4)	6	1	1	6,7,15,16,26,27,28
t99	(5,4,7)	6	1	1	6,7,14,15,16,26,27,28
t100	(5,4,8)	8	3	1	6,7,14,15,16,26,27,28
t101	(6,0,0)	6	1	1	3,4,15,16,27,28
t102	(6,0,1)	6	1	1	3,4,6,27,28
t103	(6,0,4)	6	1	1	3,4,6,27,28
t104	(6,0,7)	6	1	1	3,4,6,27,28
t105	(6,0,8)	14	2	1	3,4,6,27,28
t106	(6,1,0)	6	1	1	2,4,5,6,9,13,14,15,16,17, 18,19,20,21,22,29,30,31,32
t107	(6,1,1)	6	1	1	6,7,16,27,28
t108	(6,1,4)	6	1	1	6,7,16,27,28
t109	(6,1,7)	6	1	1	6,7,27,28
t110	(6,1,8)	14	2	1	6,7,16,27,28
t111	(6,2,0)	6	1	1	2,3,4,5,6,9,10,13,14,15, 16,17,19,20,21,22,29,30,31,32
t112	(6,2,1)	6	1	1	6,7,16,26,27,28
t113	(6,2,4)	6	1	1	6,7,16,26,27,28
t114	(6,2,7)	6	1	1	6,7,15,16,26,27,28
t115	(6,2,8)	14	2	1	6,7,15,16,26,27,28
t116	(6,3,0)	6	1	1	1,2,3,4,5,6,9,10,13,14,15, 16,17,18,19,20,21,22,29,30,31,32
t117	(6,3,1)	6	1	1	6,7,8,16,26,27,28
t118	(6,3,4)	6	1	1	6,7,15,16,26,27,28
t119	(6,3,7)	6	1	1	6,7,15,16,26,27,28
t120	(6,3,8)	14	2	1	6,7,15,16,26,27,28
t121	(6,4,0)	6	1	1	1,2,3,4,5,6,9,10,13,14,15,16, 17,18,19,20,21,22,29,30,31,32
t122	(6,4,1)	6	1	1	5,6,7,8,16,26,27,28

t123	(6,4,4)	6	1	1	6,7,15,16,26,27,28
t124	(6,4,7)	6	1	1	6,7,15,16,26,27,28
t125	(6,4,8)	8	1	1	6,7,15,16,26,27,28

Test data for Simple Calculator Program

Test case	Test Case Value	Path Complexity	Branches covered	Cov(t) for branches	Faults detected (Mutants Killed)
t1	(-2 + -1)	3	1,3,5,6, 8	4	
t2	(-1 + -1)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t3	(7 + -1)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t4	(11 + -1)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t5	(12 + -1)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t6	(-2 + 0)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t7	(-1 + 0)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t8	(7 + 0)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t9	(11 + 0)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t10	(12 + 0)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t11	(-2 + 6)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t12	(-1 + 6)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t13	(7 + 6)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t14	(11 + 6)	3	1,3,5,6, 8	4	1,2,4,5,6,7
t15	(12 + 6)	3	1,3,5,6, 8	4	
t16	(-2 + 9)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t17	(-1 + 9)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t18	(7 + 9)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t19	(11 + 9)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t20	(12 + 9)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t21	(-2 + 10)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t22	(-1 + 10)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t23	(7 + 10)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t24	(11 + 10)	3	1,3,5,6, 8	4	1,2,3,4,5,6,7,8,9,10,11,12
t25	(12 + 10)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t26	(-2 - -1)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12

t27	(-1 - -1)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t28	(7 - -1)	4	1,3,10	3	1,2,4,5,6,7
t29	(11 - -1)	4	1,3,10	3	
t30	(12 - -1)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t31	(-2 - 0)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t32	(-1 - 0)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t33	(7 - 0)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t34	(11 - 0)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t35	(12 - 0)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t36	(-2 - 6)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t37	(-1 - 6)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t38	(7 - 6)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t39	(11 - 6)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t40	(12 - 6)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t41	(-2 - 9)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t42	(-1 - 9)	4	1,3,10	3	1,2,4,5,6,7
t43	(7 - 9)	4	1,3,10	3	
t44	(11 - 9)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t45	(12 - 9)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t46	(-2 - 10)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t47	(-1 - 10)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t48	(7 - 10)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t49	(11 - 10)	4	1,3,10	3	1,2,3,4,5,6,7,8,9,10,11,12
t50	(12 - 10)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t51	(-2 * -1)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t52	(-1 * -1)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t53	(7 * -1)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t54	(11 * -1)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t55	(12 * -1)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t56	(-2 * 0)	2	1,3, 11	3	1,2,4,5,6,7
t57	(-1 * 0)	2	1,3, 11	3	
t58	(7 * 0)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t59	(11 * 0)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t60	(12 * 0)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12

t61	(-2 * 6)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t62	(-1 * 6)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t63	(7 * 6)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t64	(11 * 6)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t65	(12 * 6)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t66	(-2 * 9)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t67	(-1 * 9)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t68	(7 * 9)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t69	(11 * 9)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t70	(12 * 9)	2	1,3, 11	3	1,2,4,5,6,7
t71	(-2 * 10)	2	1,3, 11	3	
t72	(-1 * 10)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t73	(7 * 10)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t74	(11 * 10)	2	1,3, 11	3	1,2,3,4,5,6,7,8,9,10,11,12
t75	(12 * 10)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t76	(-2 / -1)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t77	(-1 / -1)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t78	(7 / -1)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t79	(11 / -1)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t80	(12 / -1)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t81	(-2 / 0)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t82	(-1 / 0)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t83	(7 / 0)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t84	(11 / 0)	4	1,3, 12	3	1,2,4,5,6,7
t85	(12 / 0)	4	1,3, 12	3	
t86	(-2 / 6)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t87	(-1 / 6)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t88	(7 / 6)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t89	(11 / 6)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t90	(12 / 6)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t91	(-2 / 9)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t92	(-1 / 9)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t93	(7 / 9)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t94	(11 / 9)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12

t95	(12 / 9)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t96	(-2 / 10)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t97	(-1 / 10)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t98	(7 / 10)	4	1,3, 12	3	1,2,4,5,6,7
t99	(11 / 10)	4	1,3, 12	3	
t100	(12 / 10)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t101	(-2 ^ -1)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t102	(-1 ^ -1)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t103	(7 ^ -1)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t104	(11 ^ -1)	4	1,3, 12	3	1,2,3,4,5,6,7,8,9,10,11,12
t105	(12 ^ -1)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t106	(-2 ^ 0)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t107	(-1 ^ 0)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t108	(7 ^ 0)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t109	(11 ^ 0)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t110	(12 ^ 0)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t111	(-2 ^ 6)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t112	(-1 ^ 6)	3	1,3, 16	3	1,2,4,5,6,7
t113	(7 ^ 6)	3	1,3, 16	3	
t114	(11 ^ 6)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t115	(12 ^ 6)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t116	(-2 ^ 9)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t117	(-1 ^ 9)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t118	(7 ^ 9)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t119	(11 ^ 9)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t120	(12 ^ 9)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t121	(-2 ^ 10)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t112	(-1 ^ 10)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t123	(7 ^ 10)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t124	(11 ^ 10)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t125	(12 ^ 10)	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12
t126	$(-2)^2$	3	1,3, 16	3	1,2,4,5,6,7
t127	$(-1)^2$	3	1,3, 16	3	
t128	$(7)^2$	3	1,3, 16	3	1,2,3,4,5,6,7,8,9,10,11,12

t129	$(11)^2$	3	1, 3, 14	3	1,2,3,4,5,6,7,8,9,10,11,12
t130	$(12)^2$	3	1, 3, 14	3	1,2,3,4,5,6,7,8,9,10,11,12
t131	$(-2)^3$	3	1, 3, 14	3	13,14,15,16,17,18,19,20,21,22, 23,24
t132	$(-1)^3$	3	1, 3, 14	3	13,14,15,16,17,18,19,20,21,22, 23,24
t133	$(7)^3$	3	1, 3, 14	3	13,14,15,16,17,18,19,20,21,22, 23,24
t134	$(11)^3$	3	1, 3, 14	3	13,14,15,16,17,18,19,20,21,22, 23,24
t135	$(12)^3$	3	1,3,15	3	13,14,15,16,17,18,19,20,21,22, 23,24
t136	$(-2)^{1/2}$	3	1,3,15	3	13,14,15,16,17,18,19,20,21,22, 23,24
t137	$(-1)^{1/2}$	3	1,3,15	3	13,14,15,16,17,18,19,20,21,22, 23,24
t138	$(7)^{1/2}$	3	1,3,15	3	13,14,15,16,17,18,19,20,21,22, 23,24
t139	$(11)^{1/2}$	3	1,3,15	3	13,14,15,16,17,18,19,20,21,22, 23,24
t140	$(12)^{1/2}$	3	1,3,15	3	13,14,15,16,17,18,19,20,21,22, 23,24

Test data for Triangle Problem

Test case	Test Case Value	Statements Covered	Branches covered	Cov(t) for branches	Faults detected (Mutants Killed)
t1	(1,2,3)	4	2,4,6,7,9	5	
t2	1,2,4)	4	2,4,6,7,9	5	
t3	(1,2,5)	4	2,4,6,7,9	5	
t4	(1,2,7)	4	2,4,6,7,9	5	
t5	(1,2,8)	4	2,4,6,7,9	5	
t6	(1,3,3)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t7	1,3,4)	4	2,4,6,7,9	5	
t8	(1,3,5)	4	2,4,6,7,9	5	

t9	(1,3,7)	4	2,4,6,7,9	5	
t10	(1,3,8)	4	2,4,6,7,9	5	
t11	(1,4,3)	4	2,4,6,7,10,12,13	7	
t12	1,4,4)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t13	(1,4,5)	4	2,4,6,7,9	5	
t14	(1,4,7)	4	2,4,6,7,9	5	
t15	(1,4,8)	4	2,4,6,7,9	5	
t16	(1,6,3)	4	2,4,6,7,10,12,13	7	
t17	1,6,4)	4	2,4,6,7,10,12,13	7	
t18	(1,6,5)	4	2,4,6,7,10,12,13	7	
t19	(1,6,7)	4	2,4,6,7,9	5	
t20	(1,6,8)	4	2,4,6,7,9	5	
t21	(1,7,3)	4	2,4,6,7,10,12,13	7	
t22	1,7,4)	4	2,4,6,7,10,12,13	7	
t23	(1,7,5)	4	2,4,6,7,10,12,13	7	
t24	(1,7,7)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t25	(1,7,8)	4	2,4,6,7,9	5	
t26	(2,2,3)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t27	(2,2,4)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t28	(2,2,5)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t29	(2,2,7)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t30	(2,2,8)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t31	(2,3,3)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t32	(2,3,4)	4	2,4,6,7,10,12,14	7	26,27,28,30,32,34,35,36
t33	(2,3,5)	4	2,4,6,7,9	5	25
t34	(2,3,7)	44	2,4,6,7,9	5	
t35	(2,3,8)	4	2,4,6,7,9	5	
t36	(2,4,3)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t37	(2,4,4)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t38	(2,4,5)	4	2,4,6,7,10,12,14	7	26,27,28,30,32, 34,35,36
t39	(2,4,7)	4	2,4,6,7,9	5	25
t40	(2,4,8)	4	2,4,6,7,9	5	
t41	(2,6,3)	4	2,4,6,7,10,12,13	7	
t42	(2,6,4)	4	2,4,6,7,10,12,13	7	33

t43	(2,6,5)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t44	(2,6,7)	4	2,4,6,7,10,12,14	7	26,27,28,30,32, 34,35,36
t45	(2,6,8)	4	2,4,6,7,9	5	25
t46	(2,7,3)	4	2,4,6,7,10,12,13	7	
t47	(2,7,4)	4	2,4,6,7,10,12,13	7	33
t48	(2,7,5)	4	2,4,6,7,10,12,13	7	33
t49	(2,7,7)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47, 48
t50	(2,7,8)	4	2,4,6,7,10,12,14	7	26,27,28,30,32, 34,35,36
t51	(3,2,3)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43, 44
t52	(3,2,4)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32, 34, 36
t53	(3,2,5)	4	2,4,6,7,9	5	25
t54	(3,2,7)	4	2,4,6,7,9	5	
t55	(3,2,8)	4	2,4,6,7,9	5	
t56	(3,3,3)	9	1,3,5,8,16,20,23,26	8	17,20,21,22,23,24, 46,47,48
t57	(3,3,4)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t58	(3,3,5)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t59	(3,3,7)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t60	(3,3,8)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t61	(3,4,3)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t62	(3,4,4)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t63	(3,4,5)	4	2,4,6,7,10,12,14	7	26,27,28,30,32,34,35,36
t64	(3,4,7)	4	2,4,6,7,9	5	25
t65	(3,4,8)	4	2,4,6,7,9	5	25
t66	(3,6,3)	6	2,3,6,8,16,19,22	7	16,41
t67	(3,6,4)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t68	(3,6,5)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t69	(3,6,7)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t70	(3,6,8)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t71	(3,7,3)	6	2,3,6,8,16,19,21	7	16,41
t72	(3,7,4)	4	2,4,6,7,10,12,13	7	
t73	(3,7,5)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t74	(3,7,7)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t75	(3,7,8)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t76	(4,2,3)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36

t77	(4,2,4)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t78	(4,2,5)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t79	(4,2,7)	4	2,4,6,7,9	5	25
t80	(4,2,8)	4	2,4,6,7,9	5	
t81	(4,3,3)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t82	(4,3,4)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t83	(4,3,5)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t84	(4,3,7)	4	2,4,6,7,9	5	25
t85	(4,3,8)	4	2,4,6,7,9	5	25
t86	(4,4,3)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t87	(4,4,4)	9	1,3,5,8,16,20,23,26	8	17,20,21,22,23,24,46,48,49
t88	(4,4,5)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t89	(4,4,7)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t90	(4,4,8)	6	1,4,6,8,15,18	6	5,6,7,8,38,39,40
t91	(4,6,3)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t92	(4,6,4)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t93	(4,6,5)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t94	(4,6,7)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t95	(4,6,8)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t96	(4,7,3)	4	2,4,6,7,10,12,13	7	33
t97	(4,7,4)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t98	(4,7,5)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t99	(4,7,7)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t100	(4,7,8)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t101	(5,2,3)	4	2,4,6,7,10,11	6	29,30
t102	(5,2,4)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t103	(5,2,5)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t104	(5,2,7)	4	2,4,6,7,9	5	25
t105	(5,2,8)	4	2,4,6,7,9	5	25
t106	(5,3,3)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t107	(5,3,4)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t108	(5,3,5)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t109	(5,3,7)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t110	(5,3,8)	4	2,4,6,7,9	5	25

t111	(5,4,3)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t112	(5,4,4)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t113	(5,4,5)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t114	(5,4,7)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t115	(5,4,8)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t116	(5,6,3)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t117	(5,6,4)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t118	(5,6,5)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t119	(5,6,7)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t120	(5,6,8)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t121	(5,7,3)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t122	(5,7,4)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36
t123	(5,7,5)	6	2,3,6,8,16,19,22	7	13,14,15,16,42,43,44
t124	(5,7,7)	9	2,4,5,8,16,20,23,26	8	21,22,23,24,46,47,48
t125	(5,7,8)	4	2,4,6,7,10,12,14	7	26,27,28,30,31,32,34,35,36