To Design a Framework for Test Suite Optimization in Regression Testing

Thesis

SUBMITTED TO

BABASAHEB BHIMRAO AMBEDKAR UNIVERSITY

(A Central University)

Lucknow

FOR THE DEGREE OF

Doctor of Philosophy

In

INFORMATION TECHNOLOGY

By

Shilpi Singh

(Enrolment No. 1377/15)

Under the Supervision of

Dr. Raj Shree

Department of Information Technology

BABASAHEB BHIMRAO AMBEDKAR UNIVERSITY

(A CENTRAL UNIVERSITY)

LUCKNOW-226025, INDIA

2018

Dedicated to my loving daughter, Saanvi

DECLARATION

I, Shilpi Singh, solemnly declare that this thesis of research on **"To Design a Framework for Test Suite Optimization in Regression Testing"** is my original work. The study has been conducted under the guidance of Dr. Raj Shree, at Department of Information Technology, Babasaheb Bhimrao Ambedkar University (A Central University), Lucknow (U.P.), India-226025. It is further declared that to the best of my knowledge and belief it has not been submitted earlier for the award of any degree. The thesis is essentially free from all kinds of plagiarism.

Dated: 16/07/2018

Shilpi Singh

(Shilpi Singh)

Research Scholar
Department of Information Technology
Babasaheb Bhimrao Ambedkar University
(A Central University)
Lucknow, (U.P.), India-226025

CERTIFICATE

This is to certify that the thesis titled **"To Design a Framework for Test Suite Optimization in Regression Testing"** submitted by **Mrs. Shilpi Singh** is an original research work and has not been previously submitted in part or full for the award of any other degree or diploma to this or any other University.

The thesis submitted to Babasaheb Bhimrao Ambedkar University, Lucknow satisfies all the requirements as stipulated in the *Doctor of Philosophy (Ph.D.) regulations-1999 as amended in 2008/2010/2013* and it is fit for submission and evaluation for the award of the degree of Doctor of Philosophy of the University.

Date: 16/07/2018

Supervisor

16/07/18.

Head of the Department

ACKNOWLEDGEMENT

The completion of this thesis would not have been possible without the help and support from many people, to whom I shall be eternally grateful.

First, I would like to thank God for the gift of life and for blessings all through my life that allowed me to get here.

I am deeply indebted to Assistant Professor and my supervisor **Dr. RAJ SHREE**, Department of Information Technology, Babasaheb Bhimrao Ambedkar University (A Central University), Vidya Vihar, Raebareli Road, Lucknow, who, in addition to being the best supervisor one could hope for, made the whole experience so exciting and enjoyable throughout. I am extremely grateful for providing me with the valuable insights that allowed me to consider my work in different contexts. She not only showed belief in me and supported me when I started the long journey, but also provided me with constructive criticism that helped me to refine my work. Her enthusiasm and involvement in my research have been instrumental in helping me stay motivated and excited about my dissertation all along. I will always be obliged to her for her wholehearted support and kindness extended to me during my entire course.

I would like to extend my sincere gratitude to **Prof. (Dr.) R. A. Khan**, Professor and Head, Department of Information Technology, Babasaheb Bhimrao Ambedkar University (A Central University), Vidya Vihar, Raebareli Road, Lucknow, for his invaluable support. I will remain indebted to him for his valuable suggestions during the entire thesis work and providing thoughtful feedback to improve its content. I will always be gratified to him for his unconditional support and kind-heartedness extended to me during my entire course.

I would like to thank to all the **experts** from India and abroad for their valuable observation during review process. I would also like to thank to all the **faculty members** and **office staff** of the Department for their cooperation and continuous support extended during the thesis work.

To my loving daughter, Saanvi, I thank you for your tenderness and for moments of happiness, making my life lighter and more joyous. My words of thanks

cannot compensate your contribution, yet with all humility I thank you for your patience and splendid support.

I especially want to thank my mother **Mrs. Sheela Singh** and my father **Mr. Ranvijay Prasad Singh**, for unconditional love, attention, affection, dedication, understanding, motivation and teachings always given to me. Thank you for all your effort extended to my personal and professional training.

To my beloved husband, **Dr. Manish Kumar**, my companion and friend, who has constantly been by my side on this long journey, stimulating me to continue and never stop believing in myself. Thank you for your dedication, support, trust, and for showing me the meaning of love every day.

To my father-in-law, **Mr. Narendra Singh**, who have been witnesses of my walk and have always cheered me on, I thank you for your affection, and all your support.

My friends **Ms. Kanika Sharma** and **Ms. Tahmish Fatima**, I thank you for always being there when I felt tired and worn out. Without your friendship this would have been a much tougher journey.

My colleagues have been a source of inspiration to me and I would like to convey my sincere thanks to **Mr. Aasim, Mr. Amal**, **Mr. Ajay**, **Mr. Neeraj, Mr. Prabhishek** and **Mr. Tarique**, for their feedback, cooperation and of course friendship and support during the entire Doctorate.

Most of all, I could not even have dreamt of starting all this without the support from my family. I thank my parents, brother and sister and everyone in the family for the love, encouragement and support they have given to me. I thank you all from the core of my heart.

To all, despite not mentioned, who directly or indirectly have contributed to the accomplishment of this work.

Shilpi Singh

SHILPI SINGH

ABSTRACT

Leading-edge technologies demand software to be built on a solid foundation of requirements, development processes, project management methodologies and quality practices. Such software-intensive systems demand high-quality product. However, it is estimated that software developers approximately spend half of the development cost and time on testing to release a quality product. Better software testing practices can cut down the cost incurred to develop quality products and also avert software failures. Regression testing is an important testing procedure used in validating modifications introduced in a system during software maintenance. It is expensive, yet an important process. As the test suite size is very large, system retesting consumes a large amount of time and computing resources. Unfortunately, there may not be sufficient resources to allow for the re-execution of all test cases during regression testing. This leads to focusing on proper test optimization techniques. Test suite reduction and test case prioritization are such techniques that address the problem of optimizing test suites. Test case optimization using reduction and prioritization techniques aim to improve the effectiveness of regression testing. Test suite reduction minimizes the test suite size by discarding duplicate and obsolete test cases according to some selected test criteria. Whereas, prioritization re-orders the test cases so that the most beneficial test cases are executed at the earliest, with higher priority.

In regression testing, software testing and retesting is done frequently. Growing of software in regression testing add a new test case to test new functionality, which is added to the existing software. These evolutions of software, which is denoted as version, create a redundant test case in the test suite. Test case becomes redundant due to more test cases satisfying the same requirement or covering same code. Availability of limited resources and time force to detect those redundant test cases which exercise the same requirement. The process of removing these redundant test cases is called as test suite minimization. The reduced test case which is derived after removal of a redundant test case is called as a representative set.

Test case prioritization techniques organize the test cases in a test suite, allowing for an increase in the effectiveness of testing. A primary performance goal of the system, the fault detection rate, is a measure of how quickly faults are determined during the testing process. An improved rate of fault detection can provide faster and productive feedback about the quality of the software application under test.

The details of the literature study reveal that the most of the reduction and prioritization strategies proposed in the literature are coverage based which do not always give a satisfactory result in terms of fault detection rate. Moreover, few reductions and prioritization strategies consider the similarity-based approach to optimize the test cases. The study also suggests that the use of multiple coverage criteria is very beneficial for the optimization process. This recommends that a combination of more than one testing criteria should be used for determining the optimal representative set, thus promoting the multi-objective heuristics to solve the optimization problem effectively.

In this sense, the main objective in this thesis is to improve the process of test suite optimization by proposing a strategy based on similarity and multi-coverage criteria in the context of code based testing aiming to maximize the fault coverage. In the context of minimization or reduction, the idea is to identify the degree of similarity among the test cases and keep in the suite the most different ones that together can meet a set of test requirements, and at the same time maintaining some redundancy in the reduced suite with the applicability of the multiple criteria. Whereas, similarity-based test case prioritization techniques primarily concentrated on test case diversity that helps to detect more faults. The technique evaluates the similarity degree for each test case pair and accordingly assigns the execution order to the test cases in a test suite.

This thesis presents a new similarity-based greedy approach, which involves the combination of regression testing techniques: minimization, and prioritization both. The main focus is on multiple regression activities with multiple criteria rather than using only a single activity to produce an optimal solution. The clustering method is also incorporated in this work, which could simplify and enhance the minimization and prioritization task. To evaluate the effectiveness of the strategy, we performed an experimental investigation together with an eminent heuristic Harrold Gupta and Soffa (HGS), considering the testing measures of the minimized test suite size and fault

coverage. The results show that similarity-based greedy approach with multiple coverage criteria can be quite effective in terms of fault detection loss of reduced test suite without much affecting the percentage of suite size reduction.

A new similarity-based test suite optimization (SB-TSO) algorithm using multiple coverage criteria is proposed in this thesis. The approach identifies the diverse test cases by comparing the distances between the test case pair with the help of three important coverage criteria i.e. statement, branch, and MC/DC. A cluster of similar test cases is generated using agglomerative clustering method. Within each cluster, optimal test cases are identified and to make the representative test suite more effective the test cases are ranked according to their assigned weight. To assess the effectiveness of the proposed approach, performance is evaluated and compared to the existing approaches. The result shows that the proposed method generates more optimal test cases that satisfy maximum coverage, minimum FDE loss, and overall size is also reduced.

This thesis presents several similarity-based prioritization techniques based on pair-wise selection strategy. Pair-wise comparison of test cases is a fundamental strategy to inspect test cases and choose an association between them in a finite test set. The proposed approach evaluates the similarity degree for each test case pair in three levels and accordingly assigns the execution order to the test cases in a test suite. The results of the experimental study show the improvement in fault detection rate that let developers initiate debugging and amending faults before the release of any software product. The comparative analysis reveals that some of the proposed techniques can attain higher fault detection rate, in terms of APFD, in comparison with the other existing techniques and random ordering.

The proposed approaches have been validated using ten different software programs. The applications were developed in C++ and necessary test cases were designed and executed for getting test related data such as code coverage in terms of statement, branch and, MC/DC. To collect the coverage information of test cases for each selected criterion, the subject program and the source code were instrumented. The proposed approaches also use hand-seeded faults for the subject programs to measure the fault detection rate of the optimized test suite.

The results of the experimental study show that the proposed approaches efficiently optimize the test suite by identifying the optimal subset of test cases from a large pool

of test cases and reordering the test cases to improve the FDE thereby enhancing the quality of software under test.

List of Tables

List of Figures

TABLE OF CONTENTS

Chapter 1 Introduction

1.1 Background

The software is a fundamentally vital and the most significant part of any computer system. A system can't perform without software that controls their functionalities and influences it to do valuable work. A computer without software is pointless, much the same as an auto without somebody to drive it. The software products are widely used in various critical real-time based applications such as aeronautics, astronautics, nuclear technology, navigation and other control operations. To make the software product quality one (robust, reliable, scalable, safe and secure), the organizations around the globe are spending lots of efforts and their resources.

A major problem for any software organization is to develop error-free software product. Therefore, ‘software crisis’ has become a fixture of everybody lives with many well-publicized failures, which has not only become a cause of economic loss but also a loss of life. The Explosion of the Ariane 5 Rocket, The Y2K Problem, Experience of Windows XP, The USA Star-Wars Program etc. is an example of some drastic failures. During the software development process, Software Requirements Specification (SRS) for Software Products has been set to explain the intended use of the product. To fulfill the customer’s requirements is one of the most important tasks for any software development organizations. Software development has been a manual process for a long time. Errors are a part of human’s daily routine. They make errors in their thoughts, in their actions, and that might affects the quality of any software product. For example: misinterpretation in user requirements, violations in rules of the system design, or insensitive mistake by the programmer during coding etc. There are other various sources of program errors are possible during the software development process. Therefore, regardless of putting much care and best effort in the design, development

and coding phase of the software development life cycle, a few errors may stay in the product after its completion.

Because of technological development and competitiveness in business, software continues evolving. One of the real key components for any successful software development is the prompt delivery of the software product to the customer. Developers are aimed to develop quality software with the help of modern technology and powerful tools. But, the occurrence of human errors is unavoidable and this resulted in the disastrous failure of the final software product. Testing the software is the only way to reveal the human errors that determine whether the errors are in human thought, actions and in the software product generated. Research has demonstrated that no less than half of the total software costs are included in the testing activities (4, 7, 13, 41).

Software testing is an activity that aims to evaluate a system attribute or component under specified conditions to observe and ensure that the desired results are produced or not. Also, in light of that, an assessment is to be made for some part of the system or component (IEEE Standard 610.12-1990) [40]. Software testing is the most important and expensive technique used to decide; whether the quality of the software system is acceptable or not; whether the software meets all the customer requirements or not; whether the product meets functional and no-functional objectives or not; and whether the product meets the production standards or not. It is mainly the execution of a test object with the purpose of finding errors by a number of test cases. Testing can show the presence of errors but not their absence [22]. Accordingly, the primary challenge here is to design an error revealing test case that eventually determines the scope and quality of the testing process. Even though achieving zero-defect quality software is the ambition, it is not possible in reality [12].

According to the study of the National Institute of Standards and Technology (NIST), the cost of insufficient infrastructure for software testing is estimated to be $22.2 to $59.5 billion [14]. These studies illustrate the requirement for an optimization approach in the software testing process to reduce the number of resources without compromising the quality. Therefore, the significance of software testing optimization has increased in recent years as clients/customers require the fast development of quality software. Rapid changes in the market with high competition and development of new tools and technologies also require optimization approach that improves the testing efficiency.

1.2 Need for Software Testing

Software testing is defined as “the process of executing the program with the intention of finding errors along with establishing confidence that the given program behaves as per specified requirements and function correctly” [15]. Software testing has two major roles in the software development process. The first role is to reveal the bugs from the SUT (Software under test) and the second is to validate that the SUT performs correctly or not. Tassey [14] suggested that by improving software testing infrastructure, almost one-third of the loss can be positively reduced that are caused by software errors. So, a well-designed testing process improves the rate of successful delivery of quality software product that performs as per the customer’s requirements under all circumstances. The testing process is very expensive and time-consuming task but releasing any software without being tested is undeniably more costly and dangerous. Hence testing is very essential to enhance the quality of any software product by thoroughly testing them at various steps of the development process. It was assessed that product testing and debugging alone devour right around half of software development resources [8].

The main issue of the above discussion is that serious issues can arise after releasing the software product without adequate testing. To find the maximum errors in the initial steps of the development cycle is one of the approaches to control the testing cost. The cost of identifying and removing those errors will be very reasonable as compared to identifying them in the later phases or at the end of the development process. Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. The cost to fix the errors is increasing drastically from the specification phase to testing and then at last to the maintenance phase as displayed in Fig. 1.1. If this cannot be detected even during testing and the customer found it later after release, the cost becomes very high. We are quite unable to predict the cost of failure for a real-time based applications software. The world has seen many catastrophic failures and these failures are quite expensive for software organizations.

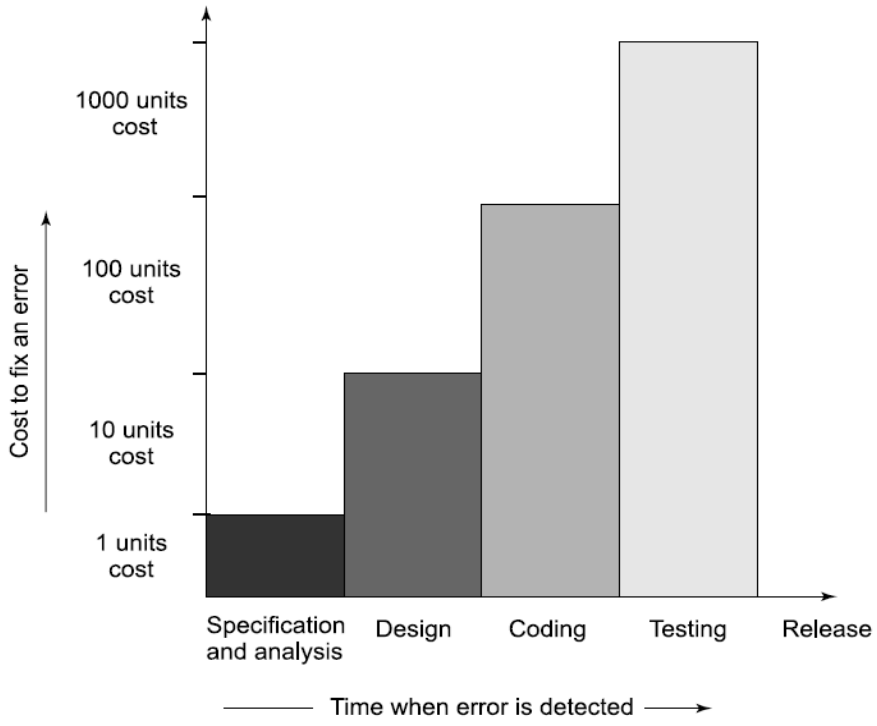


Figure 1.1 Phase wise cost of fixing an error [39]

Software testing provides an objective, independent view of the software to the business to rise and understand the risks of implementing the software. Software testing can be stated as the process of validating and verifying that a computer program/application/product fulfills the given requirements as mentioned by customers and also satisfies the needs of stakeholders or not. Verification and Validation are used interchangeably. The Institute of Electrical and Electronics Engineers (IEEE) has given definitions of both the terms which are generally accepted by the testing community.

Verification: "Whether the products of a given development phase satisfy the conditions imposed at the start of that phase". It is the process of reviewing the documents of the software project like requirement document, design document, source code etc. produced after the accomplishment of each development phase.

Validation: "Whether the system or component satisfies the specified requirements during or at the end of the development process". It is a dynamic testing and needs actual execution of the given program.

Therefore, software testing comprises of both verification and validation.

Testing = Verification + Validation

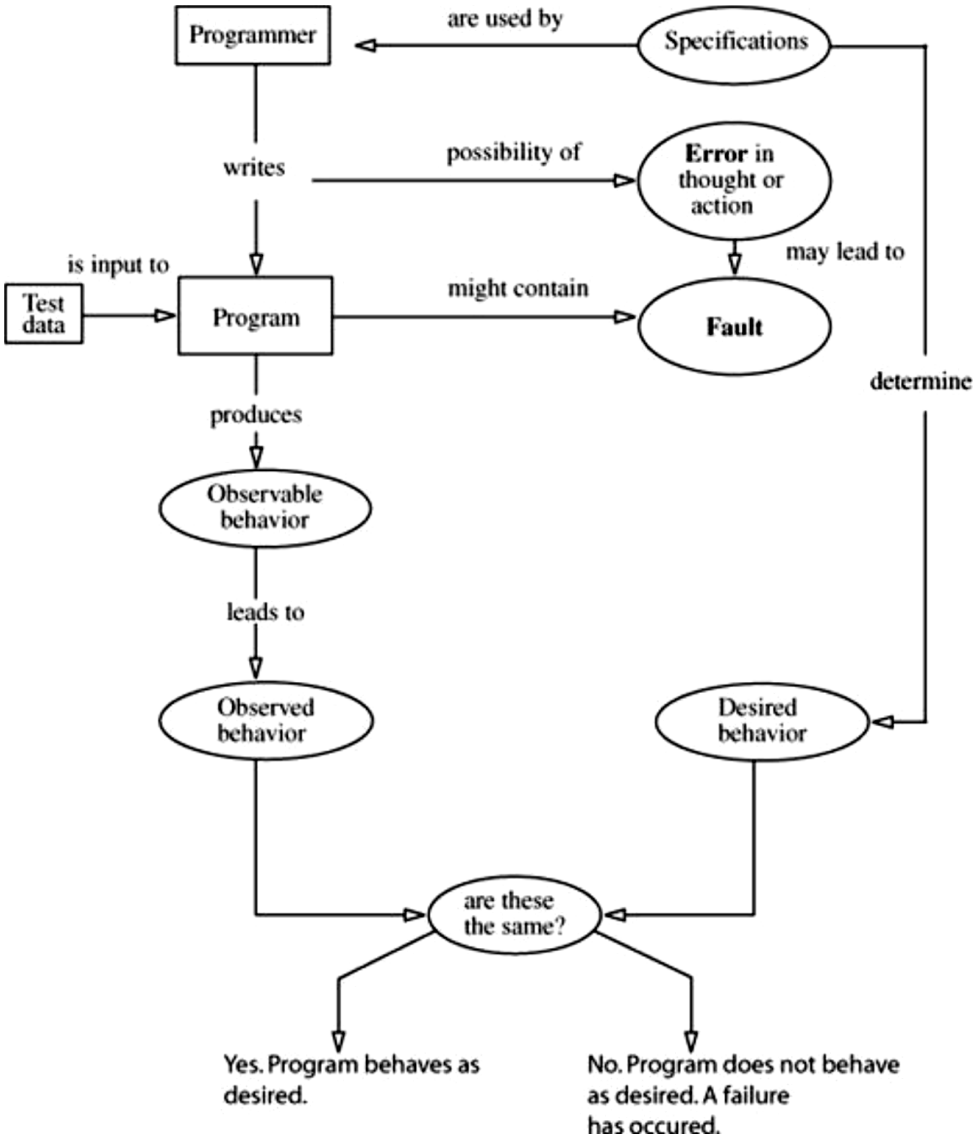


Figure 1.2 Errors, faults, and failures in the process of programming and testing [39]

1.2.1 Fault, Error, Bug and Failure

When a human makes an error during programming/coding, this termed as a 'bug'. Hence, an 'error'/ 'mistake'/ 'defect' occurs in the process of writing a code is called a bug. A 'fault' is the representation/expression of one or more errors. Execution of faulty code leads to 'failure' or to the incorrect state. Depending upon the different combination of inputs, a particular fault may cause different failures. When there is a difference between expected and observed behavior, then this means that failure has occurred. Fig. 1.2 illustrates the meanings of the above-discussed terms and clearly shows the difference in observed and desired behavior.

1.2.2 Test, Test Case and Test Suite

Test and Test Case are may be used interchangeably. A test case comprises of inputs given to the program and expected outputs. The observed output(s) with the expected output(s) are compared to determine that the test case is successful or not. If the output is the same, the test case is successful, otherwise, the failure occurs and it should be entered correctly to find the cause of that failure. A good test case is one that has a high probability of revealing faults in a program. Therefore, testers should first identify the weak points of the program and design the test cases accordingly. The group of test cases or any combination of test cases is known as a test suite.

1.2.3 Testing Methods

For effective testing, test cases are generated by a number of different testing techniques [22]. Test cases are responsible for making software complete by finding maximum errors with different testing conditions. Therefore, the testers do not estimate that the test case and testing techniques should be chosen to enable them to systematically prepare the test conditions [56]. Apart from this, the combination of testing techniques can give improved outcomes than the use of a single testing technique [57]. The testing approaches used in practice can be categorized in two different perspectives:

- Black box testing
- White box testing

Black Box Testing

Black box testing is also known as a functional testing. The tester design the test cases based on the functionality of the software program and the internal structure or knowledge of source code are completely ignored. Normally, during black box testing, the tester interacts with the system's interface by providing valid or invalid inputs and examines the output without knowing it how the input works (Figure 1.3).

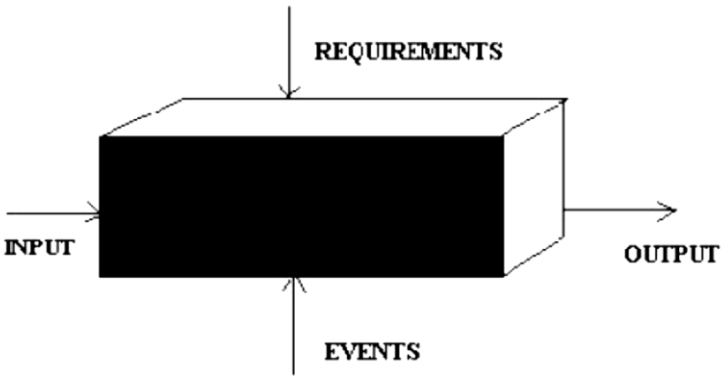


Figure 1.3 Block diagram of black box testing

White Box Testing

White box testing is more technical than black box testing and also known as structural testing. It requires detailed study of internal logic and structure of the source code to generate test cases. In order to perform white box testing, the tester needs to thoroughly examine the source code to understand the internal workings and other implementation details of the code (Figure 1.4). It allows a tester to recognize which section of the source code is performing inappropriate. White box testing is a verification technique. Here source code becomes the base document to inspect the internal structure of the program and it also requires a good knowledge of program structure [12, 39]. Many white box testing techniques are available and some of them are Control-flow testing, data flow testing and mutation testing. Test requirements required for these techniques may include statement coverage, def-use coverage, path coverage etc.

The major structural testing techniques can be broadly classified as:

- Statement testing: It is a test strategy in which each statement of a program is executed at least once
- Branch testing: Tests all branches in the source code at least once
- Path testing: Tests all paths in the source code at least once
- Condition testing: Allows the programmer to determine the path through a program by selectively executing code based on the comparison of values

- Expression testing: Tests an application for different values of a regular expression
- Data flow testing: Focuses on the flow of variables used within a program.

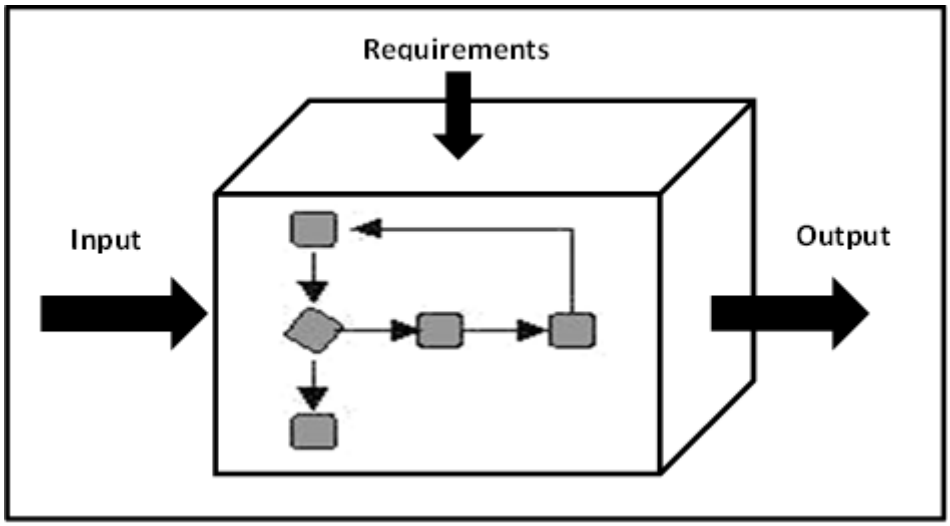


Figure 1.4 Block diagram of white box testing

The main objective of software testing is to find out the failures to detect and improve the defects before deployment. We must define the testing requirements of the program prior to performing testing [9]. Then, according to the requirements we have to design test cases to satisfy them. With the advancement of software, size of test suite grows, which also increases the possibility of the existence of duplicate and obsolete test cases in a test suite. Because the cost of running and maintaining the test suite is high, it is quite beneficial to produce representative subsets of test cases to meet all the test requirements. To evaluate the adequacy of any test suite, coverage criteria can be used. In essence, the coverage criterion defines a set of test requirements that a test suite should be exercised and for each testing requirements, a suitable test case is generated and then evaluated accordingly. As the software system enhances, the efforts and resources required during testing also increase in terms of the number of test cases, time, cost and manpower. It is extremely difficult to test software on all input datasets; hence optimization of the test suite is mandatory that generates a small subset of test cases that have the same fault revealing ability as the original one.

1.3 Regression Testing

Regression testing is to be performed in two cases: (a) addition of new component/module in the existing system or (b) some corrections/modifications in the system, which may affect other features of the existing software system. So, the purpose of regression testing is to re-establish the confidence that the newly introduced features or any function in the updated software program have not badly affected existing features. It ensures that the previous program code still working properly or not after some modifications. Software regression testing processed continuously during the software development and maintenance of evolving software. Maintenance requires some modifications, which leads to growth in software and it results in an increment in test suite size. Over time, some test cases in a constructed test suite may become redundant, because the test cases created specifically for some selected testing criteria may also satisfy other requirements, and a requirement may still satisfy by some of the proper subsets of the test suite. Two test cases are termed as duplicate or redundant if their satisfied testing objectives are same. On the other hand, some of the test cases are termed as essential if their testing objective is unique. So, the prime objective is to remove the duplicate test cases and extract the essential or diverse test cases to generate the optimal test suite. In the maintenance phase, regression testing works as a major component. Retesting of the program is most of the time as tedious as the original test. It means if the modification causes the existing functional part of the program to fail, this error often goes undetected. To solve this problem we can use retest-all approach [37].

Regression test must be conducted once the software modification has been carried out. However, as software grows after required modifications, the test suite also grows accordingly, which means it may be too costly to execute the whole test suite. The testers cannot get enough resources and sufficient time to successfully perform a regression test. This problem forces the researcher to think about the development of those optimization techniques, whose purpose is to reduce cost and effort required for regression testing in so many ways. Figure 1.5 presents a basic flow diagram of regression testing process.

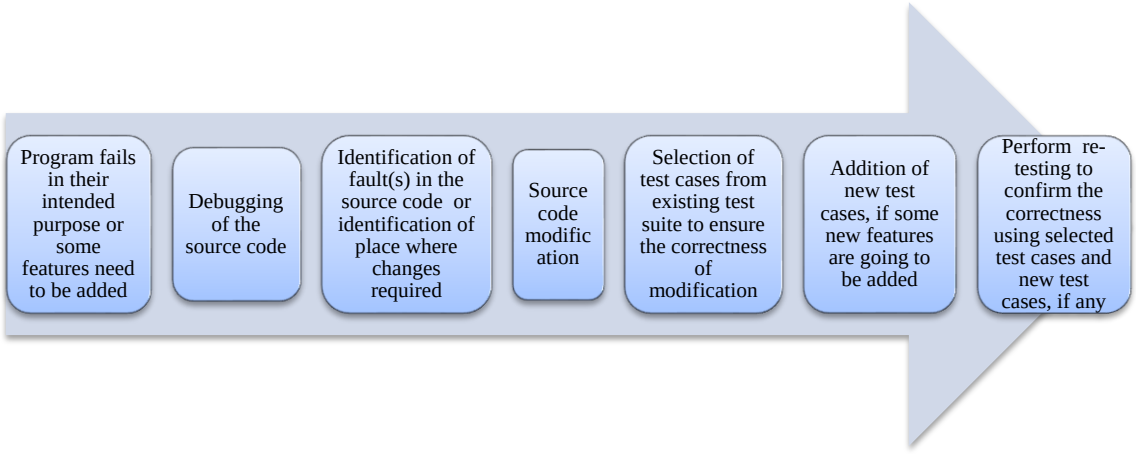


Figure 1.5 Regression Testing Process

Regression testing can be characterized by Progressive Regression testing and Corrective Regression testing [38]. Progressive Regression testing involves changes of requirement specifications because of new enhancement or new data requirements in a system. In contrast, Corrective Regression testing does not involve changes in requirement specifications, but only in some design decisions and actual instructions of the program [37].

1.4 Need for Regression Testing

In addition to the software development and testing, software maintenance is also an important and widely accepted part of SDLC (Software Development Life Cycle) in the domain. The IEEE Standard 1219-1993 [30] has defined software maintenance as the "modification of a software product after delivery to correct faults, to improve the performance or other features, or to adapt the product to a modified environment" [6]. Figure 1.6 shows the maintenance process defined by IEEE standard. Software programs, although initially well-written, are subject to unavoidable changes. The changes due to changed requirements instigate the software programs to be modified accordingly. The improvements related to any new version of the existing software also require modifications in the software programs. Similarly, when new features are added to the existing software, it requires changes in programs as well. Effectiveness and utility of any software project prominently depend on how the changes or modifications

in the software programs are managed. Regression testing can be considered as an important evaluation technique to manage and revalidate the changes in the software programs.

In software regression testing, to maintain the effectiveness of the test suite is also one of the important and challenging tasks that are performed by testers. A test suite is generated to validate any software product and reused to check each successive release of that product.

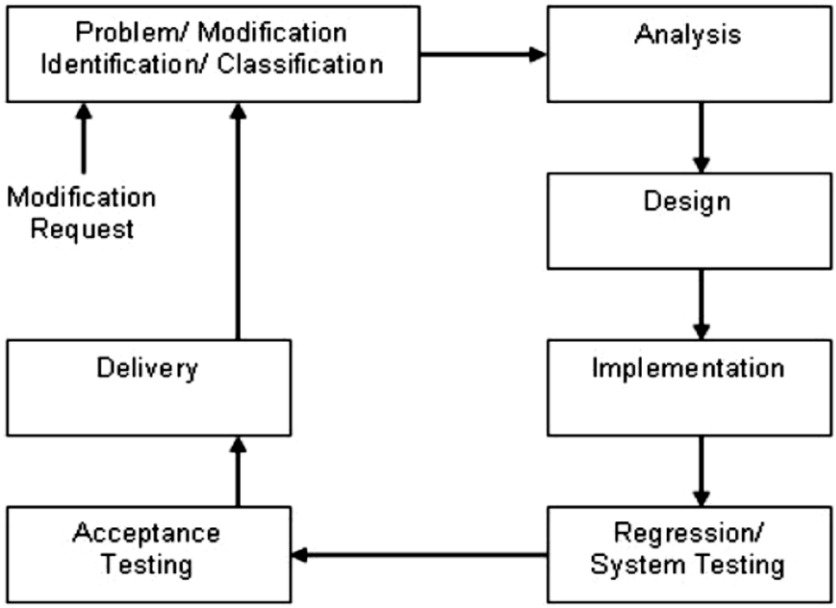


Figure 1.6 IEEE Standard 1219-1998 Software Maintenance Process [40]

In IEEE standard a test case is defined as a set of test inputs, condition, and expected results generated to achieve specific objectives such as to cover certain path or requirement. The output of the executed test illustrates whether it meets the customers and system requirement or not. The collection of test cases is termed as a test suite and its quality depends on mainly two factors i.e. size and fault coverage. To test an updated version of the software, new test cases are added to the existing test suite and that results in suite size increment. The execution of the whole test suite is very time consuming and costly as well. Therefore, different regression testing techniques are proposed to optimize and generate more effective test suite, which is of three categories namely test case selection, test suite minimization and test case prioritization.

Regression testing in software maintenance is an expensive but necessary activity and this almost consumes half the time spent on software maintenance activities. To improve the time and expenses required in software maintenance, there is a need to improve regression testing techniques. Empirical studies have proved that by optimizing the test suite, the performance of the regression test can be substantially improved. Since regression testing is a very expensive task, superfluous execution of duplicate and obsolete test cases will increase the avoidable cost. Therefore the resolution is to select the best test cases and to remove inappropriate, unnecessary ones, which in turn leads to the optimization of test cases [11].

1.5 Test Suite Optimization

Testing is a very expensive process as well as an important task of the SDLC, through which we add some value to the software program. Inadequate testing is one of the major cost factors. Testing efforts, often consume more than half of the overall development resources. Early detection of faults and failure reduces maintenance costs as well as requires fewer corrections [32-33]. According to the IEEE definition [20], a test case is a collection of input data given to the program and expected output results created to evaluate a software function or test requirement. It is hard for a single test case to satisfy the coverage of entirely given test requirements. That is why; a number of test cases are generated and collected in a test suite [35]. Because of the extensive use of testing to measure the quality of the software, one of the challenges faced by organizations is the test suite optimization [36]. Test suite optimization aims at finding the best subset of test cases from the current test suite to do regression testing on updated software programs. Test case selection, minimization, and prioritization are some of the activities that are involved in test case optimization. A brief description of each optimization technique is presented here.

Test cases minimization is a selection of the smallest subset the test cases from a pool of test cases to be audited for a program. Test suite reduction seeks to reduce the number of test cases in a test suite while retaining a high percentage of the original suite's fault detection effectiveness. Test suite minimization techniques seek to reduce the effort required for regression testing by selecting an appropriate subset of the test suite.

Test cases selection also finds minimal cardinality subset of test cases from the pool of test cases. One major difference between test cases minimization and test case selection is that test case selection chooses a temporary subset of test cases, whereas test suite minimization reduces the test suite permanently based on some external criterion such as structural coverage.

Test case prioritization techniques try to find an ordering/ ranking of test cases so that some test case adequacy can be maximized as early as possible.

Minimization and prioritization of test cases are one of the two important solutions to the problem of test case optimization. So far several research works have been carried out to optimize the test cases using these techniques. However, all of them have focused on single criteria to minimize or prioritize the test cases. Conducting a regression test based on single criteria is not more effective. One of the effective methods to conduct regression testing is executing the test suite by covering multiple criteria using a single test case order so that more than one performance goal will be achieved in a single regression test process. The overall aim of the thesis is to optimize test cases for the regression testing by employing multiple factors.

1.6 Motivation

Test suite optimization is more interesting and important due to many reasons. First of all, it reduces the total execution time of a test suite to test any software in a limited time. Secondly, the generation of effective test cases improves the coverage and fault detection ability of test suite during regression testing. During regression testing, it is quite infeasible and even costly to run a large number of test cases to test any new and modified part of the software program. An effective way to optimize the test suite is to discard the duplicate and obsolete test cases and to retain the most essential test cases in a test suite. By distinguishing between redundant and essential test cases and dealing with them reduces the cost, effort and the risk of losing reliability of test suite. However, in the case of minimization, it may be effective to minimize the test suite, but not guaranteed the fault detection effectiveness (FDE) [26, 28]. So, the reduction is done in such a way that there should be no or less fault detection loss. Whereas, regardless of minimizing the test suite size early fault detection is also important while

doing regression testing that is entirely dependent on the execution order of test cases. Early detection of faults, quick delivery of software product etc. is the result of a good order of test cases. Test case prioritization schedules the test cases in such a way that if the test cases executed according to the given order, it maximizes its effectiveness to meet some specified criteria. Here criteria are some goal that is established by tester according to their requirement and expertise. In this manner, early fault detection leads to early debugging process so as to reduce the software maintenance cost.

The minimization and prioritization techniques basically divided into two categories: Coverage-based and Similarity-based. Coverage based techniques have gained wide attention but they do not always give a satisfactory result [27, 29]. In recent years, the similarity-based approach has to gain popularity among researchers. The purpose of similarity-based techniques is to maximize the diversity (i.e., minimize the similarity) of selected test cases. Where, the diversity between a pair of test cases is computed through any of the chosen distance measures. Selection of the distance/dissimilarity measure may directly influence the performance of test suite optimization strategies. The core idea behind this approach is to select the most dissimilar subset of test cases rather than focusing on maximizing code coverage and group them accordingly for further test case minimization or prioritization process. Consequently, this will increase the chance of detecting faults as early as possible if we maximize the diversity of the test cases. The similarity-based approach can be applied to both the minimization and prioritization techniques.

The empirical study reveals that research on similarity-based minimization and prioritization techniques are done using single criteria [30, 31]. Multi-criteria formulation of the test suite optimization problem may produce better results in terms of coverage as well as minimum fault detection loss with early fault detection rate. The observations made out of the literature study, have motivated this research work to propose a new framework for test suite optimization based on similarity approach. This approach results in the reduction of overall costs associated with regression testing.

1.7 Problem statement and objective

As software grows, testers perform regression testing to validate the updated software before their release. The main aim of regression testing is to check new features and make sure that the changes lead to new faults into the existing software or not. For the above purpose, new test cases are also added to the existing test suite to test new features and that results in obvious suite size increment. However, executing all the test cases is infeasible and expensive for testers.

With the aim of optimizing the test suite, many researchers proposed different regression testing techniques; such as test case selection, minimization, and prioritization techniques by using different approaches. Majority of the existing minimization tools and framework consider code coverage information of the software to be tested as a base to determine the minimized test suite. Coverage based test suite reduction techniques have gained wide consideration but they do not always give a satisfactory output. Empirical studies, however, reveals that code coverage may not the strong criteria for test suite effectiveness. To address this problem, a number of techniques and framework have been proposed to make the reduction process more effective which is based on test case classification according to similarity degree measured by a distance function. Diversity and similarity-based test case selection and prioritization are one of the new approaches with the favorable output. In a similarity-based approach, optimization of the test suite is processed on the basis of the calculated similarity degrees between test case pairs. So, the study suggests that use of similarity-based approach could be a better option for effective regression testing.

Instead of using single coverage criteria, use of multiple coverage criteria to calculate the distance between the pair of test cases is quite beneficial to generate optimal test cases. The purpose of coverage criteria is to measure the adequacy of test suites and its quality. Thus, the process of minimizing/prioritizing the test cases with respect to certain coverage criteria preserves the adequacy of the suite with respect to those criteria. But, discarding the test cases that are selected as a duplicate/ unnecessary according to a specific criterion may not be sufficient and may execute unique condition with respect to another criterion. The ignorance of such essential test cases can be the reason of fault detection loss. This recommends that a combination of more than one

testing criteria would be worthwhile for determining the optimal representative set, thus promoting the multi-objective heuristics to solve the optimization problem effectively.

The aim of test suite optimization is to reduce the size of the test suite to be assessed by discarding unessential test cases and improve the effectiveness and quality of test suite. So, test cases minimization and prioritization are one of the approaches that improve the quality of testing by optimizing size and fault detection ability of test suite. It will automatically reduce the cost and efforts requisite for software regression testing. Thus the focus of our work is to develop an optimization approach that can improve the regression testing process by reducing the size of test suite without compromising their fault detection ability and can also improve the APFD (Average Percentage of Fault Detection) value by scheduling the execution order of test cases. Use of similarity-based approach with multiple criteria for minimizing and prioritization of test cases are also another major objective, which ultimately improves the effectiveness of regression testing.

1.8 Contributions of the Research

This dissertation makes the following contributions:

It reviews the relevant literature on regression testing with a greater focus on minimization and prioritization.

It presents a novel similarity-based greedy approach to identify duplicate test cases using distance metric. It also uses a clustering approach for test suite minimization.

It presents a similarity-based test suite optimization framework using multiple coverage criteria to get optimal test cases.

It proposed a pair-wise similarity-based approach for test case prioritization using distance metric.

Each work is empirically or statistically validated on ten sets of data to evaluate the performance of the proposed work which makes the proposed work socially acceptable.

1.9 Organization of Thesis

The thesis is organized as follows.

Chapter 1 provides a detailed introduction to testing and the importance of software testing. The regression testing and their need are described. Some issues related to regression testing and about test suite optimization are also presented herewith. The motivation behind this work and problem statement with objective are described. Major contributions of the thesis are also included.

Chapter 2 provides the basic introduction to regression testing and their techniques. The chapter discusses test suite minimization and test case prioritization. Categories of test cases and different coverage criteria are also briefly explained. A detailed section is provided on the core topic of the thesis i.e. about test case optimization. This chapter also covers the comparison between coverage based and similarity-based approach. Various distance measures, performance measures are also discussed in this chapter.

Chapter 3 provides a review of related research literature on the core topic of the thesis. The chapter presents a survey of studies on test case minimization and test case prioritization. A detailed study of research works on similarity-based minimization and prioritization is provided.

Chapter 4 presents the proposed similarity based greedy approach to get an optimal test suite. In order to evaluate the quality and effectiveness of the proposed approach, the experiment is conducted on a sample of the subject programs. The well-known standard greedy approach i.e. HGS algorithm is implemented; to compare the results of optimized test suites using the proposed similarity-based test suite optimization algorithm with those of minimizing test suites using the HGS algorithm.

Chapter 5 presents the proposed similarity based test suite optimization approach for effective regression testing. The chapter introduces a novel approach that incorporates a similarity-based strategy with multiple coverage criteria to get an optimal result. The approach uses three coverage information i.e. Statement coverage, MC/DC coverage and branch coverage to construct a similarity matrix by calculating the distance between test case pairs. Various statistical studies are carried out to show the acceptability of the framework.

Chapter 6 presents the pair-wise selection strategy to prioritize the test cases in regression testing. A similarity-based approach is used to identify the difference level between a pair of test cases which quantitatively illustrates that how much the test case pair are similar or diverse to each other. The proposed approaches or techniques reorganize the execution order of test cases based on the similarity value of test case pairs computed in three levels. Each level represents the integration of selected coverage criteria. Statistical validation is carried out to make the approach acceptable in the society.

Chapter 7 includes the major conclusions and future scope of work. In this chapter, major research findings and their significance are presented in detail. Future plans for extending the study are discussed.

Chapter 2 Regression Testing

2.1 Background

Software testing is the process of demonstrating that a software program executes its intended function as expected. The primary opportunity for any testing team is to deliver quality software while maintaining the development cost as well. During the testing process, the program is executed on the test cases, and the observed outcome is compared with the expected outcome [4, 16, 17]. The goal of software testing is to run the software program according to the test plan, discover the faults that cause failures, and improve the software quality by removing the discovered faults. It is a very expensive process as well as the important task of the software development lifecycle, through which we add some value to the software program. Adding some value means improving the quality and reliability of the program.

Inadequate testing (not properly examine) is one of the major cost factors. Early detection of faults and failure reduces maintenance costs as well as requires fewer corrections. In software testing, the testing requirements are gathered from Software Requirement and Specifications (SRS). Once a set of requirements is found, a test set is generated to fulfill the requirements manually or automatically [18, 19]. According to the IEEE definition [20], a test case is a collection of input data and expected output data, which are mainly created to evaluate a particular software function by executing the software against these. With the help of a single test case, the tester cannot check the coverage of given requirements. That's why; there is a need for a test set or test suite (collection of test cases) to be generated [21]. A test case in a test suite either said to be redundant or essential. Two test cases are termed as duplicate or redundant if their satisfied testing objectives (testing requirements or criteria) are same. On the other hand, some of the test cases are termed as essential if their testing purpose is unique (not

satisfied by remaining test cases). This chapter discusses the important regression testing techniques that are responsible for test suite optimization. The chapter is organized as follows. The second section presents the basics of regression testing. The third section presents types of regression testing techniques. The most significant topic in this chapter is about test suite minimization and prioritization techniques. The fundamental concept of minimization and prioritization techniques is presented in the fourth and fifth section. The sixth section presents some important distance metrics that are used to calculate the similarity level between test cases in the optimization process. This section also presents some important performance measures to evaluate the effectiveness of the proposed work.

2.2 Regression Testing

Regression testing is defined as “the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously tested code” [1]. Regression testing is performed on modified software to ensure the correct behavior of the software and the absence of any adverse effect of the modifications on the software quality. To understand software regression testing, consider a development cycle of a program as shown in Figure 1.1. The Figure shows a highly simplified develop-test-process lifecycle of a program P, which is referred to as Version 1. While P is in use, there might need to add new features, remove any reported errors, and rewrite some code to improve performance. Such modifications lead to P', referred to as Version 2. This modified version must be tested for any new functionality. However, when making modifications in P, developers might mistakenly add or remove code, which affects the existing and unchanged functionality of P. One performs regression testing to ensure that any malfunction of the existing code could be detected and repaired prior to the release of P' [23].

It should be obvious from the above description that regression testing can be applied in each phase of software development. Regression testing is also needed when a subsystem is modified to generate a new version of an application. When one or more components of an application are modified, the entire application must also be subjected to regression testing.

Version 1	Version 2
1. Develop P	1. Modify P to P'
2. Test P	2. Test P' for new functionality
3. Release P	3. Perform regression testing on P' to ensure that the code carried over from P behaves correctly
	4. Release P'

Figure 2.1 Two phases of product development and maintenance

However, regression testing requires an excessive number of test cases for testing any new or modified functionality of the program.

2.2.1 Categories of test cases

Leung and White [38] give a classification of the test case (see Figure 2.2) and regression testing systematically. Regression testing as:

1. **Progressive regression testing:** Whenever new improvements or new functionalities are incorporated in a software system, the specification will be modified to reflect these additions. It involves major modifications such as: adding and deleting modules. This testing usually performed during adaptive and perfective maintenance and invoked at regular interval. Fewer test cases can be reused here.
2. **Corrective regression testing:** The specification does not change after any modifications or enhancements in the system. It involves some minor modification such as addition and deletion of statements, modifications in design decisions etc. This is usually performed during development and corrective maintenance and invoked at an irregular interval. Many test cases can be reused here.

Leung and White classify test cases into five different groups.

1. **Reusable:** This class of test cases only executes the parts of the software program that remain unchanged between two versions P and P0. However, they

are termed as reusable because they may still be reused for the regression testing of the updated versions of program P0.

2. **Retestable:** This class of test cases executes the parts of program P that have been changed in P0. Thus, retestable test cases must be re-run to test P0.
3. **Obsolete:** This class of test cases no longer proves what they were designed to test due to modifications in the program.

The RTS techniques have accomplished regression testing by identifying the test cases that need not be rerun on a new version of the software [39]. Let us examine the RTS technique using Figure 2.3. Let P denote Version X that has been tested using test set T against specification S. Let P' be generated by modifying P. The behavior of P' must conform to specification S'. Specifications S and S' could be the same and P' is a resultant of removing faults from P.

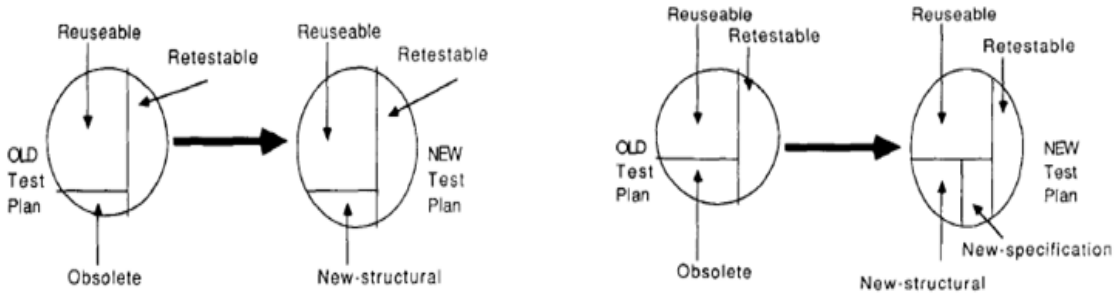


Figure 2.2 Corrective and Progressive Regression Testing

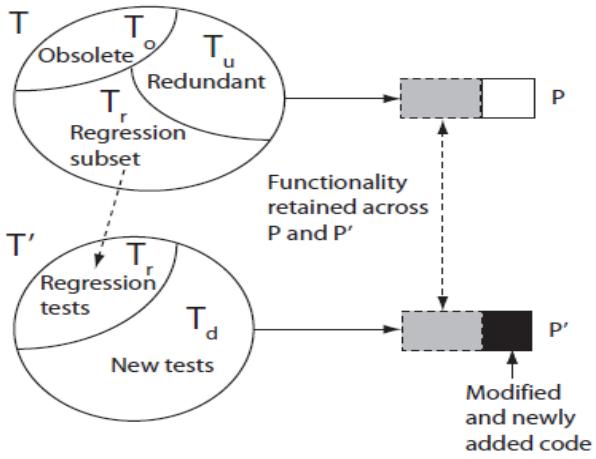


Figure 2.3 The RTS Problem

After the modification, two new groups or classes may be created which contains those test cases that have yet to be generated for the regression testing of P0.

4. **New-structural:** This class of test cases tests the modified or updated program constructs. They are usually created for structural coverage of the modified parts in P0.
5. **New-specification:** This class of test cases tests the new code generated from the modified parts of the requirement specifications of P0.

2.3 Regression Testing Techniques

Regression testing executes existing test cases on the modified program to assure that the changes cannot affect the functionality of the previous program. With the rapid advancement in the software systems, the test suite size usually grows very large [110] and it may contain obsolete and redundant test cases as well. It may be very cost effective to re-run the whole test suite within limited time and resources. It is quite effective to eliminate such unessential test cases from a test suite to reduce the cost and effort required for regression testing. Usually, there is a strict requirement of optimization approach to improving the quality of regression testing.

Many regression testing techniques, such as test case selection, minimization, and prioritization have been proposed to optimize the process of regression testing [37]. Test case selection and minimization makes the regression testing more effective by executing a subset of the existing test suite. Test case prioritization ranked the execution order of test cases so that the test cases with higher priority are executed first. Current state-of-the-art test suite optimization approach can be categorized into four groups: coverage-based, search-based, similarity-based, and Integer Linear Programming (ILP) based [26]. Majority of the existing minimization tools and framework consider code coverage information of the SUT (software under test) as a base to determine the minimized test suite and follow any of these concepts: Greedy [110, 111], GE [112], GRE [113], and HGS [114]. Coverage based minimization techniques have gained wide attention but they do not always give a satisfactory result. Empirical studies, however, reveals that code coverage may not the strong reason for test suite effectiveness [27]. It may be effective to minimize the test suite, but not guaranteed the fault detection

effectiveness (FDE) [72]. To address this problem, several other heuristics have been proposed to make the minimization process more effective which are based on test case classification according to similarity degree measured by a distance function [71]. Diversity and similarity-based test case selection and prioritization are one of these new approaches with favorable output [95]. The core idea behind the diversity-based approach is to select the most dissimilar subset of test cases rather than focusing on maximizing code coverage and group them accordingly for further test case minimization or prioritization process.

2.4 Test Suite Minimization

Test suite minimization aims to retain essential test cases and remove redundant ones from a test suite, generating an optimal test suite that is a minimal subset of the original test suite. Test suite minimization techniques aim to find duplicate test cases with respect to some testing criteria, such as statement, branch coverage, path, MC/DC etc. A test case is either termed as essential or duplicate. Essential test cases are the reverse of duplicate test cases [114]. If any test case satisfies the requirement r_i uniquely, the test case is termed as essean ntial test case. In contrary, if a test case covers equal requirements or only a subset of the test requirements covered by another test case, called as dupla icate test case. Most of testthe suite minimization problems can be stated as a minimal hitting set problem which is known as NP-complete [37].

According to Harrold et al. [41], the test suite minimization problem can be defined as follows:

Given: $\{t_1, t_2, t_3, \dots, t_n\}$ represents a test suite T containing n test cases and $\{r_1, r_2, r, \dots, r_k\}$ represents a set of testing requirements that must be satisfied in order to provide required coverage of the program and each subsets $\{T, T_2, T, \dots, T_n\}$ from T are associated to one of the r_i's such that each test case t_j related to T_i covers r_i.

Problem: Discover minimal test suite T' from T which satisfies all r_i's covered by the original test suite T.

Referring to Figure 2.4 there are four possible similarity scenarios of any test case pair.

1. (T1=T2) – Two test cases are equal, meaning their test case coverage is similar for the same data.
2. (T1<T2) – A first test case T1 is a subset of second test case T2. A set of requirement coverage satisfied by T1 is part of another larger satisfied coverage set of T2.
3. (T1>T2) – Second test case T2 is a subset of first test case T1. A set of requirement coverage satisfied by T2 is part of another larger satisfied coverage set of T1.
4. (T1≠T2) – The test case pair T1-T2 are not similar to each other or not redundant.

In the first, second and third scenarios, there exist test case redundancy. So, in the second and third scenarios, the test case T1 and T2 can be ignored or otherwise not used. The purpose of the test suite reduction is to minimize the number of test cases, but also reduce the fault detection ability of the test suite. Wong et al. [62] conducted a series of minimization experiments and observed that lack of fault detection was not very significant. Though, Elbaum et al. [63] also accompanied minimization experiments but observed that the loss of fault detection capability was very significant.

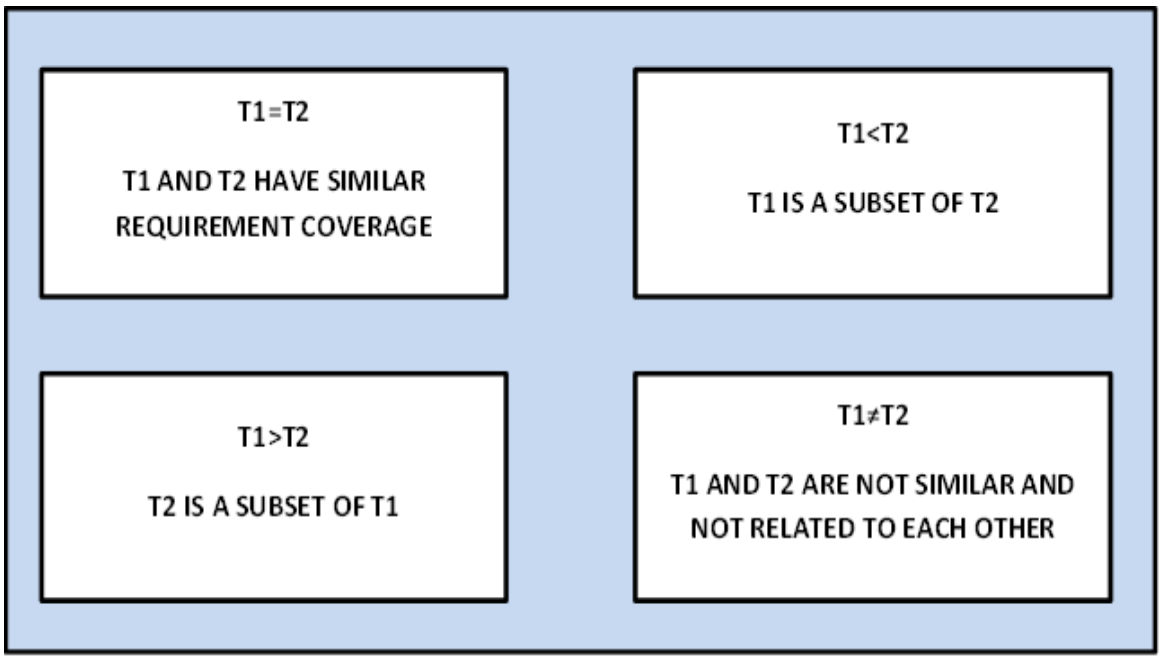


Figure 2.4 Test Case Similarity Scenarios

2.5 Test Case Prioritization

Test case prioritization ranked the test cases within a test suite based on some criteria with the aim of maintaining fault detection effectiveness. The main focus of test case prioritization is to maximize the probability of meeting certain objective (maximum coverage or higher fault detection rate) of the test cases when executed in a particular order [2]. Under limited time and resource constraints, ordering the test cases may increase the testing effectiveness by executing the essential test cases as early as possible. The effectiveness of minimized test suites can be further improved or optimized by ordering the reduced test suite.

Test case prioritization for early fault detection has many main advantages.

- First, if a bug is found early in the regression testing process, debugging can start earlier and hence bugs could be fixed faster.
- Minimization of testing costs.
- Testers may wish to increase their confidence in the reliability of the system under test at a faster rate.
- Identification of high-risk defects earlier.
- Increasing the reliability of the software.

Test case prioritization problem can be defined as follows [2]:

Given: TS represents a test suite, PT represents a permutation of TS, and a function $f: PT \rightarrow R(real\ number)$

Problem: to find $T' \in PT$ such that: : $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$

Test case prioritization aims to order the execution of test cases in the regression test suite, such that the benefit to the tester is maximized. In other words, the test suite is ordered based on some criterion that is expected to lead to the early detection of faults. This prioritization criterion can be based on coverage, requirements, models or the history of modifications to the system. More details about these and other prioritization techniques can be found in Yoo and Harman's survey of regression testing [37].

Prioritization based on coverage is based on the hypothesis that early maximization of coverage could lead to early detection of faults. Several coverage-based prioritization techniques have been proposed and evaluated in the literature [37].

2.6 Coverage versus Similarity-based Test Suite Optimization

The main aim of any optimization approach is to remove the redundant test cases and generate an optimal test suite that contains the most diverse or essential test cases. Where, test suite quality is measured by their coverage and fault detection capability. So, while applying any test suite reduction technique, the trade-off between the test case execution time required and their fault detection effectiveness should be considered. The optimization techniques can be classified into two main classes: coverage-based and similarity-based techniques.

2.6.1 Coverage-based techniques

Coverage-based optimization techniques, where we estimate that "in cases where there is more coverage (such as code coverage) more likely to detect defects" [24].These techniques intended to identify and remove duplicate test cases though increasing the coverage of a program to get a confidence that all independent paths are executed, and faults are uncovered as well. However, previous studies reveal that the test cases which are identified as redundant by the primary coverage criterion may be identified as essential test cases by the other secondary coverage criteria. These results propose that while optimizing test suite, taking multiple coverage criteria is more beneficial rather than using single coverage criterion. In some of the cases, code coverage might be insufficient to achieve a higher rate of fault detection.

2.6.2 Similarity-based techniques

Similarity-based techniques estimate that "more diversity in test cases is more capable of manifesting the faults" [25]. These techniques intended to select test cases according to the calculated diversity of test case pair that represents the degree of similarity of each test case pair. Different similarity measures are discussed in the further section

which helps to calculate the difference level of each test case pair. Based on their similarity degree, a cluster of similar test cases can be generated. Similarity measures have been widely used in numerous areas, such as information retrieval, document clustering etc. In recent times, similarity measures are also used in the field of software testing. Ledru et al. [94] applied string distances for prioritizing test cases. For model-based testing Hemmati et al.[28] proposed a novel test case selection technique based on a similarity-based approach. Yoo et al. [92] proposed a collective approach of clustering with expert knowledge to attain scalable prioritization. Fang et al. [95] proposed different similarity-based test case prioritization techniques by means of edit distances of ordered sequences. Singh et al. [74] proposed different test case comparison metrics to measure the diversity between a pair of test cases to reduce the test suite size my removing most similar test cases. Figure 2.5 shows the block diagram for test suite optimization using distance or similarity function.

2.7 Test Coverage Criteria

Code coverage analysis is the structural testing technique and their purpose is to measure the adequacy of test suites and its quality. The term ‘coverage’ describes as a ‘percentage of source code which has been tested with respect to the available total source code for testing’. Given certain coverage criteria, C that is satisfied by one of the test cases t residing in the test suite TS, another test case $t'$of that suite is redundant with respect to C if the criteria C is also satisfied by that test case [83]. One of the aspects of coverage analysis is to identify redundant test cases that are unable to increase coverage.

Thus, the process of removing test cases from a test suite that are redundant with respect to certain coverage criteria preserves the adequacy of the suite with respect to those criteria. But, redundant test cases that are selected to be removed from the test suite according to a particular criterion may not be redundant and may execute unique condition with respect to another criterion. The removal of such redundant test cases can be the reason of fault detection loss. This recommends that a combination of more than one testing criteria should be used for determining the optimal representative set, thus promoting the multi-objective heuristics to solve the minimization problem

effectively. A large variety of coverage criteria exist for coverage analysis. We may like to obtain a reasonable level of coverage using any of the structural testing techniques such as control flow, data flow testing etc. The test coverage information of a test suite can be represented using coverage matrix [2]. This matrix represents the relationship between requirements and test cases. In the coverage matrix, each row represents a coverage requirement, which may be a statement, branch, path etc. and each column represents a test case t_j (Table 2.1).

Table 2.1 Requirements coverage matrix

Test Cases / Requirements	T_1	T_2	. . .	T_n
r_1	1	0	. . .	1
r_2	0	1	. . .	1
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
r_n	1	0	. . .	0

The matrix $A = [a_{ij}]_{m \times n}$ is used to describe the satisfaction relation between requirements and test cases, Equation (2.1).

$$a_{ij} = \begin{cases} 1, & r_i \; is \; covered \; by \; t_j \\ 0, & Otherwise \end{cases} \tag{2.1}$$

Where, i = 1, 2, 3,…, m and j = 1, 2, 3,…, n.

Table 2.1 is an example that shows the coverage information of test cases in a test suite T. The value '1' indicates coverage requirement by a test case and '0' represents that the requirement is not satisfied by the test case.

The coverage of requirements is a fundamental issue to be addressed throughout the software life cycle. Requirement coverage involves mapping test cases with requirements to check whether all the requirements have been covered by the given test cases (or) not. The empirical study reveals that structural testing based on either control flow or data flow coverage criteria can show significantly improve fault detection when compared to random testing [66].

2.8 Distance Measures

In this section, we present the six distance functions applied in this work to calculate the similarity degree between pairs of test cases. These functions are good candidates to identify the similarity between test case pair. While other works have already applied these functions to similarity-based selection strategies [31, 72, 95] in our work, we apply these functions in the context of test suite reduction and test case prioritization for code-based testing. Moreover, despite the fact that there are many other distance functions presented in the literature, we describe a small set with the ones that are included in other studies in the general area of selection, minimization, and prioritization of test cases.

2.8.1 Similarity Function

Cartaxo et al. (2011) define a redundancy measure that calculates the similarity degree between two test cases defined as paths. The degree is measured as the number of identical transitions divided by the average of paths length.

$$Similarity\ function[i,j] = \frac{nit}{Avg(|i|,|j|)} \tag{2.2}$$

Where,

- nit is the number of identical transition pairs between the two test cases;
- $Avg(|i|,|j|)$ is the average between paths length.

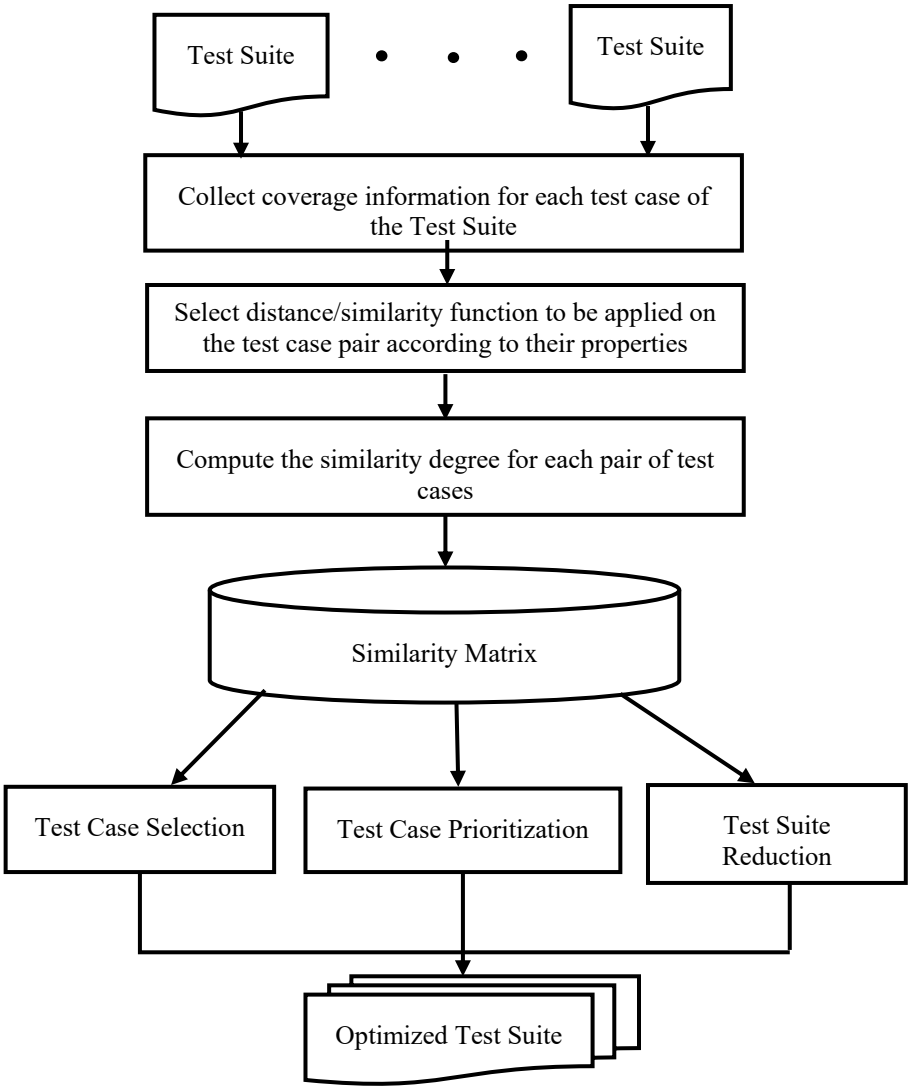


Figure 2.5 Block diagram for similarity-based test suite optimization

2.8.2 Cosine Similarity

By executing the test cases t1 and t2, the binary coverage vectors covered by them are Y: $(y_1, y_2, \dots, y_n)$ and Z: $(z_1, z_2, \dots, z_n)$ respectively. The similarity value between t1 and t2 can be computed as:

$$S(t1, t2) = \frac{Y^t \cdot Z}{\|Y\|\|Z\|'} \tag{2.3}$$

Where,

- Y^t→Transposition of vector Y,
- $\|Y\|$→Euclidean norm of vector Y,
- $\|Z\|$ → Euclidean norm of vector Z,
- S → Cosine of the angle between Y and Z.
- And, the corresponding dissimilarity is represented as:

$$Diss(t1, t2) = 1 - S(t1, t2) \tag{2.4}$$

2.8.3 Euclidean Distance

Let, V1: $(v_{11}, v_{12}, \dots, v_{1n})$ and $V2$: $(v_{21}, v_{22}, \dots, v_{2n})$ represent two coverage vectors (Binary) by performing test cases t1 and t2 on a given program. Where, coverage may represent statement, branch, the path etc. The Euclidean distance between pair of test casesthe t1 and t2 can be evaluated as:

$$d(t1, t2) = \sqrt{\sum_{i=1}^{n} (v_{1i} - v_{2i})^2} \tag{2.5}$$

2.8.4 Hamming Distance

Hamming Distance is a very popular distance measure function in the literature. It is based on edit distance method and only valid for two strings of identical length. Between two test cases, hamming distance is evaluated as the ratio of the total number of dissimilarities by the total number of positions.

To apply Hamming Distance in test suite optimization, each test case to be encoded and denoted as a binary vector of length n. Where n signifies a total number of all selected criterions used for encoding (e.g., n might be a statement, branch, function etc.). In a binary vector, if the test case covers the selected coverage criteria, their corresponding bit is set to be true (1), otherwise false (0).

2.8.5 Jaccard Index

Based on the Jaccard Index and its variants, the similarity degree between two test cases t1 and t2 are calculated as follows:

$$s(t1, t2) = \frac{Y \cdot Z}{Y \cdot Z + a(\|Y\|^2 + \|Z\|^2 - 2(Y \cdot Z))'} \quad (2.6)$$

Where,

- Y·Z→ inner product of Y and Z,
- If we set a = 1, the above formula is known as a jaccard index,
- If we set a=2, this formula is known as Soka-Sneath distance measure,
- If we set a = 1/2, this formula is known as Gower- Legendre distance measure, respectively.

For Jaccard Index and its variations, the corresponding distance is calculated as:

$$\mathrm{d}\ (\mathrm{t1};\ \mathrm{t2}) \ = \ 1 - \mathrm{s}\ (\mathrm{t1};\ \mathrm{t2}) \quad (2.7)$$

2.9 Summary

This chapter presented an overview of the regression testing and their techniques. The chapter also described the core concept of the thesis i.e. similarity based test suite optimization in a detail. A section on basic concepts in test suite minimization and prioritization was also included in this chapter. The fundamental concepts about coverage criteria were also discussed. Finally, some of the important distance measures used in the similarity based optimization have been identified.

Chapter 3 Review of Literature

Test case selection, minimization, and prioritization are three important techniques for effective regression testing. Selection and minimization try to obtain an optimal subset of the existing test suite. Whereas, test case prioritization order the test cases according to some specified objective. The test suite optimization techniques can be classified into two main classes: coverage-based and similarity-based techniques.

3.1 Test suite minimization

Minimizing the size of the test suite is a major task of any test suite minimization techniques. First of all, it detects the duplicate and obsolete test cases and then removes them from the existing suite. The minimization problem is considered to be the same as the minimal hitting set problem.

3.1.1 Coverage-based test suite minimization

Harrold et al [41] proposed an approach named HGS algorithm for test suite minimization, which attempts to minimize a test suite in terms of a given set of testing requirements. This algorithm starts by calculating the cardinality of all T_is. The test set having single cardinality is first included in the representative set. The algorithm progresses with the next higher cardinality and the test cases with maximum requirement coverage are added to the set. If there is a tie, then Th_is is checked with cardinality (n+1) and the test case randomly chosen in the representative set, otherwise, the test case having maximum requirement coverage is selected in the representative test set.

Chen & Lau [42] proposed a technique using divide and- conquer approach to obtain near-optimal solutions. It comprises of decomposing the original test suite into subsets.

Initially, a minimized test suite is calculated separately for each subset and then these subsets are joined to obtain the final minimized test suite. The researchers suggested two different techniques for distributing a test suite into subsets and illustrate how the final optimal test suite could be generated. Experimental analysis was conducted to measure the performance of the developed technique and it observes that the resulting optimal test suites are quite effective for the purpose of regression testing.

Jones and Harold [83] offered two new techniques for test suite minimization, which are specially designed to be used with the consideration of quite complex Modified Condition / Decision Coverage (MC / DC) criteria. Every condition in the criterion takes a decision by executing it independently which affects the outcome of the decision. To show this, the criterion requires that every condition is covered by a particular MC/DC pair which is a pair of truth vectors (truth values for each condition in a decision). By deciding each condition independently in the criterion affects the outcome of the decision. And for this, the criterion requires that each condition is covered by a special MC / DC pair, which is a pair of true vectors. An empirical investigation concluded that both proposed minimization techniques could achieve significant improvement in suite size reduction. Though, the fault detection effectiveness of these techniques was still relatively impulsive.

Marre´ and Bertolino [44] designed a test suite reduction technique that uses the graph to identify the representative test suite. A decision to decision graph (dd graph) is generated in this graph based approach using the software which is more optimized and compact than the control flow graph (CFG). The testing requirement like def-use is properly mapped into dd graph that makes the test suite reduction problem reduces to minimal spanning tree problem.

Tallam & Gupta [45] proposed a heuristic named as a Delayed-Greedy strategy. In this approach, test cases can be seen as objects and requirements as attributes. This technique classified the cluster items and their respective attributes. Their experimental study reveals that the delayed-greedy approach has achieved consistent or better reduction in test suits as compared to HGS or in classical greedy approach.

Jeffrey and Gupta [46, 47] used the concept of selective redundancy for minimizing test suite. They improved the HGS algorithm for selectively retaining some test cases for

better coverage. First of all, they identify the duplicate test cases according to primary criteria and mark them. Then, these test cases are checked for redundancy in terms of secondary criteria. Second, if it is not duplicate then it is also chosen. The proposed approach is experimentally evaluated with HSG strategy using branch coverage and def-use coverage. The experimental observation reveals that the proposed approach is more capable of detecting faults as compared to HSG with the single criterion. But, this strategy not produces a minimal test suite than the single criterion one.

Black et al. [48] also considered Bi-criteria approach to solving the minimization problem effectively. In this approach, the previous fault history is collected with test coverage criteria. Here the ILP method is used to find the optimal test set. ILP method is also used by Hsu and Orso [49]. To improve the proposed work of Black et al., they introduce a multi-criterion ILP approach by combining the weighted sum approach and priority optimization.

Heimdahl and George [50] followed the idea of using a simple greedy approach for test suite minimization using model-based tests. They generated similar output as similar to the work by Rothermel et al [51]. They also discussed that optimal test suite could be achieved with an undesirable fault detection loss. The empirical studies were conducted based on specified criteria by optimizing the test suites. However, they reported less average fault detection loss than those stated in Rothermel et al [51]. They also claimed that the fault detection loss is unacceptable in the critical and real-time based systems which must be improved.

Rui Zhang et al [52] improved the Quine-McCluskey method to generate the minimized test suite. Intuition was drawn from the fact that the reduction in boolean function and test suits is equivalent to resolving the test suite optimization problem.

Parsa et al [53] introduced a novel algorithm for generating a minimized test suite. The proposed method considers the minimization of test suite as an optimization problem with two main objectives: (a) the ability to detect the faults should be maximized and (b) the existing test suite should be reduced. This algorithm tries to select a test case which meets the maximum number of test requirements upon the minimum overlap in coverage of the requirements with other test cases. The proposed algorithm is actually

very effective in minimizing test suite with an appropriate reduction in suit size and minimum fault detection loss.

Selvakumar et al [54] presented a heuristic based on Changed Condition/Coverage Criteria (CC/CC) for test suite reduction. The proposed approach concentrated on satisfying maximum user requirements with less time, cost and effort required for testing thereby increase the computational efficiency.

Shengwei et al [55] proposed a new approach, name as a modified greedy algorithm for resolving the test suite reduction problem. The approach uses Weighted Set Covering (WSC) techniques to generate a more optimal test suite.

3.1.2 Similarity-based test suite minimization

da Silva Simao et al. [67] proposed a new technique to generate optimal test cases, reducing the regression time spent on evaluating a new version of software while maintaining the fault detection ability. The technique uses an ART-2A self-organizing neural network architecture to classify the test cases within a test suite. The test cases are concise in a feature vector, which includes all significant information about the software activities. The feature vectors are classified into different clusters with the help of a neural network, which are labeled according to software activities. The feature vectors obtained from all-uses code coverage information to a random selection approach are compared during the experimental evaluation and it observes that the performance was improved by applying the new technique.

Kovcs et al. [68] proposed an optimal test selection method that generates efficient test sets for systems based on SDL specifications. They used string edit distance based coverage metrics for selection. They evaluated the proposed method by implementing and incorporating the proposed algorithm into SDL-based test selection framework by conducting experiments. The observed results specify that the string edit distance based method produces quite similar output in terms of suite size reduction and coverage, whereas complexity is less as compared to other heuristics.

Chen et al. [69] presented a theoretical analysis to demonstrate that ART (Adaptive Random Testing) is an effective alternative to random testing. Important intuition to develop ART was the concept of "even spreading" across the entire input domain. Here

"even spreading" can be described as variety. The achievement of ART proves the effectiveness of the failure-based testing approach. It also shows the impact and importance of similarity or diversity on the test suite effectiveness.

Cartaxo et al.[31] proposed the similarity-based selection strategy in the context of MBT. The main focus of the proposed strategy is on LTSs. The main objective is to apply a distance function to identify the most different or similar test cases among the generated target number of test cases. For evaluation purpose, two types of coverage criteria were considered to equate it with a random strategy that is mostly used in previous works. The results represent that the similarity strategy is very effective in terms of discarding duplicate or obsolete test cases according to the specified criteria. The goal of the similarity based approach is not to change the user's experience when choosing high-quality test cases. In contrast, the approach can reduce the test suite so that, if necessary, options become possible; it means, the approach can select many test cases, which can be dealt with increasing diversity within a test suite.

Coutinho et al. [71] proposed a new similarity-based approach for test suite reduction in the context of Model-based Testing (MBT) which aims to identify the similar test cases based on the calculated similarity degree between a pair of test cases. The main idea is to classify the test suite between the most similar and the most diverse test case according to specific criteria. So that the optimized test suites have the less similar test cases, aiming to have maximum requirement and fault coverage. The experimental results illustrate that the minimized test suite obtained by using the proposed approach is on average greater than the test suite obtained by using other heuristics.

Hemmati et al. 2013 [72] introduced a different test case selection techniques in the context of MBT, called similarity-based test case selections (STCS). The technique minimizes the similarity and increases the diversity between test cases to decrease the fault detection loss. They empirically evaluated 320 different similarity-based test selection techniques and comparatively analyzed the similarity-based test selection technique with other existing selection techniques. The results after experimental evaluation show significant improvement in performance when using a similarity-based approach.

Sharma et al. [73] introduced a set of test case comparison metrics algorithms which minimizes the test suite size by calculating the diversity between test case pair of an existing test suite. With the help of these algorithms, testers can easily capture the diversity level for each specified criteria in terms of signature values between two test cases. The approach mainly concentrates on branch coverage, control flow coverage, def/use and data flow coverage. By using this coverage information signature values is computed to know how much the test cases are diverse or similar to each other. Accordingly, a number of clusters are generated, that can effectively test under limited time.

Coutinho et al. [91] examined the effectiveness of distance functions and presents the results of empirical studies, aims to compare distance functions while reducing test suite based on the similarity strategy in the context of MBT. The primary objective is to provide confirmation that the choice of any distance function can directly affect the performance of the approach in terms of suite size reduction and fault coverage. And the observation reveals that the selection of distance function has little impact on suite size reduction, but it can greatly affect the fault coverage ability.

3.2 Test case prioritization

The main motive behind the prioritization is to generate the ordered test cases for getting maximum profit in terms of APFD. Preferably, test cases should be executed in a sequence which maximizes early fault detection ability. However, the information related to the ability to reveal the fault is unknown until the test is over.

3.2.1 Coverage-based test case prioritization

The coverage-based techniques are methods to prioritize test cases based on coverage criteria, such as statement coverage, total requirement coverage, additional requirement coverage, fault coverage etc. The following paragraphs present existing coverage-based prioritization techniques that have been proposed.

Wong et al [75] offered a coverage-based prioritization technique and specified cost per additional coverage the primary function of prioritization. The coverage information

collected previously by executing the test cases, this proposed technique provides the rank for test cases in terms of the achieved coverage according to the specified criteria.

Gregg Rothermel et al [76] presented several prioritization techniques for ordering the test cases with the aim of improving the fault detection rate. The proposed method enlightens about the random and optimal prioritization for effective ordering of test cases in a test suite to improve the fault detection rate. Researchers considered nine approaches to prioritize a set of test cases and also report the outcomes of measuring the effectiveness of those approaches to improve the fault detection ability. Overall they proposed the following techniques: (a) random approaches (b) optimal prioritization (c) total branch coverage prioritization (d) additional branch coverage prioritization (e) total statement coverage prioritization (f) additional statement coverage prioritization (g) total fault-exposing-potential prioritization and (h) additional fault-exposing-potential prioritization.

Gregg Rothermel et al [77] describe test execution information to order the test cases for effective regression testing like prioritizing the test cases according to their total coverage of code components prioritize the test cases based on their assessed fault revealing ability. In [78-82] different techniques for prioritization have been presented and discussed according to fault detection rate or code coverage capabilities and also evaluated by different empirical studies.

Jones and Harrold [83] proposed new algorithms for both test suite reduction and prioritization. The proposed algorithm designed very effectively to associate with Modified Coverage (MC) and Decision Coverage (DC). Most existing techniques consider a set of coverage criteria such as, statements, decisions, def-use etc. to investigate the test suite reduction and prioritization techniques. During the empirical study, they concentrated on MC and DC coverage criteria for test case reduction and prioritization. To weight and schedule the test cases, the proposed approach uses total requirement coverage and the additional requirement coverage accordingly.

Leon and Podgurski [84] presented the comparative analysis of four different prioritization techniques to get the optimal test suite: test suite minimization, prioritization by additional coverage, cluster filtering with one-per-cluster sampling and failure pursuit sampling. The first two techniques are based on selecting subsets, which

maximize code coverage as soon as possible, while the latter two are based on the analysis of the delivery of the test execution profiles. Their results show that compared to coverage-based techniques their techniques can be as efficient or more efficient in exposing the defects. Consequently, some possible combinations of these techniques were assessed to prioritize the test cases in a test suite. The observed output suggested that using this combination of techniques can produce a more effective result as compared to prioritize the test cases by additional coverage only.

Aggrawal et al [85] presented a prioritization technique that achieves improved code coverage at the fastest rate. The proposed technique has the benefit over the general test case prioritization which can use information about the code changed to give preference to test cases.

Bryce and Colbourn [86] proposed a new prioritization technique for interaction testing. This is the mechanism for testing software system installed on a variety of hardware and software configurations. When the ordered set of test cases is completed, they optimize a greedy method to generate a set of tests to identify interactions according to all pairs; other important test cases are run on completion without completing the test.

Srikanth et al [87] proposed techniques that are based on some estimated factors. Most importantly, the techniques imitate the software project practices of any industry more closely. The proposed technique mainly focuses on the priority of requirements, perceived code complexity, and customer satisfaction. Customer satisfaction is one of the most influential components in their technique.

Jeffrey and Gupta [88] not only proposed a new priority approach to the order of test cases based on total statement coverage, but it also considered the number of statements executed in that case or the output produced by the test case has the ability to influence. The proposed approach is based on the observation that, if the revision in the program is to affect the output of a test case in the test suite, then it must affect some calculation in the relevant piece of the output of that test case. Therefore, to prioritize test cases, their heuristics specifies high weight for a test case with a large number of statements in relevant pieces of output. Following factors were used in the proposed approach to prioritize test cases: (a) the number of statements in the relevant piece of output for the

test case, and (b) the number of statements that are executed by the test case but are not in the relevant piece of the output.

Sampath et al [89] presented the prioritization approach for test cases of web applications. Session-based test cases are considered ideal for testing web applications because they reflect the actual usage patterns of real users, making them realistic test cases. Empirical evaluation has shown that ordered test suits perform better than randomly ordered test suits, and there is not any single ordering criterion that is always best as well.

Srivastava [108] proposed cost-effective test case prioritization. In the proposed technique test cases are ordered by applying priority based scheme. Priority is assigned to each test case based on high code coverage, number of faults or on fault detection rate. The average number of faults detected per minute is computed by dividing the number of faults discovered by each test case by the time taken to discover faults. The calculated value is used to identify the priority order of test cases in a test suite.

Lilly Ramesh et al [109] discussed the behavior of test cases, redundancy among test cases and knowledge mining of the test suite. They also discuss the three main data mining techniques i.e. are classification, association rules, and clustering. This helps adequately to identify patterns in the test suite, which is evidently extracted through many means.

Ryan Carlson et al [90] discussed the prioritization of test cases using one of the data mining technique i.e. clustering approach in regression testing. The clustering method helps to simplify the prioritization processes by dividing test cases into groups of common properties.

Badhera et al. [58] presented the prioritization technique to execute the updated lines of code with a minimal number of test cases. The prioritization technique schedule the test case in a test suite in an order such that less number of code needs to be executed again, thus more rapidly code coverage is obtained which facilitates early identification of the defects.

B. Jiang et al. [59] presented an ART-based test case prioritization technique which takes the test suite as an input and gives the output in a prioritized order. The basic idea

behind this is by creating a set of candidates for test cases, which in turn selects a test case from the candidate's set until all the test cases are selected. There are two functions used in the proposed algorithm to calculate the distance to choose a pair of test cases and a test case from the candidate's set. Calculation of distance is primarily based on code coverage data. We then find that the candidate test case is connected to the distance with the test cases already selected.

Kaur et al. [60] proposed a new genetic algorithm to prioritize the test cases within a time-constrained environment based on the total fault coverage. The proposed algorithm is empirically evaluated with the help of Average Percentage of Faults Detected (APFD).

Hong Mei et al. [61] proposed a new prioritization approach for ordering test cases in the lack of coverage information under the JUnit framework. A proposed approach named as JUPTA(JUnit test case Prioritization Techniques operating in the Absence of coverage information). The approach uses the static call graphs of JUnit test cases and evaluates the test case ability to achieve coverage and rank the test cases according to those estimates.

Daniel et al. [64] present a case study on a real-world application with real faults to evaluate the performance of the coverage based regression testing techniques. The study assesses four techniques which are: test selection technique, test suite minimization technique and a hybrid technique comprises of selection and minimization both. The work also examines the impact of using different coverage criteria on the effectiveness of the studied optimization approaches. The observation reveals that prioritization techniques based on additional coverage with finer grained coverage criteria achieve better fault detection rates and it also reveals that the use of information about modifications in prioritization techniques does not significantly improve the fault detection rates.

3.2.2 Similarity-based test case prioritization

Yoo et al. [92] proposed human interactive test case prioritization technique using the clustering method. Human interaction is obtained using AHP, which is a widely used decision-making tool that was earlier adopted by the requirements engineering

community. The clustering method is used by the proposed technique to reduce the number of pair-wise comparisons requisite by AHP, makes scalable to regression testing problems. A hybrid interlaced cluster priority technology was proposed to combine these two technologies. To experience, empirical studies were conducted that the hybrid ICP algorithm could perform better than traditional coverage-based priorities for some programs, even if the human input is incorrect.

Jiang et al. [93] presented the first family of adaptive random test case prioritization techniques and organizes an experiment to evaluate its performance. It investigates ART techniques with the different test set distance definitions at various code coverage levels rather than spreading test cases equally and quickly on the input domain. Empirical results show that our techniques are quite effective as compared to a random order. Apart from this, one of the ART prioritization techniques is quite competitive with some of the best coverage-based techniques and yet includes very little time cost.

Ledru et al. [94] proposed a new prioritization technique based on string distances between test cases in a test suite. String distance is tested for comparison of test cases and broadens the prioritization algorithm. This type of prioritization does not require the availability of any implementation or a specificity of the program to give priority to the test suite: it depends only on the information contained in the test suite. The empirical study approved by a statistical analysis indicates that the prioritization based on string distance is more effective in identifying faults by the random sequence of suits: the ordered test suite using string distance is more effective in detecting the strongest mutants. And, on an average, the proposed technique got better APFD than the randomly ordered test suite.

Fang et al. [95] used the ordered sequences of program objects to improve the effectiveness of test case prioritization process. The execution frequency details of test cases are collected and converted into the ordered sequence. Based on the distance of the ordered sequences, several novel similarity-based test case prioritization techniques are proposed. Two algorithms i.e. FOS and GOS are proposed to get ordered test cases. Moreover, an empirical study is conducted and the output revealed that the proposed techniques can improve the fault detection rate more significantly than the existing techniques.

Wang et al. [96] proposed an algorithm based on the global similarity-based approach by comparing the similarity between test cases in a given test suite. The approach re-determines the execution order of test cases based on the distance between paired test cases. In addition, they compared the effects of six similar measures on the global similarity based technique. Experimental results show that using the Euclidean distance, the proposed approach is most effective.

Wang et al. [97] evaluated and statistically analyzed to recommend the best combination of similarity-based prioritization algorithms and distance measures. Experimental results confirmed by a statistical analysis, indicate that Euclidean distance is more efficient in finding faults than other similarity measures. Combination of global similarity based prioritization algorithms and Euclidean distance may be a better option. It generates not only higher FDE but also a lower standard deviation.

3.3 Summary

The summary of the literature survey is as follows:

- The software test optimization problem is much more difficult to solve and cause discontinuities in the objective function. And it is known to be Non-Polynomial (NP) hard.
- Many algorithms have been proposed for specific types of problems, but still, more common problems require a more effective solution.
- Research on software regression testing focuses mostly on code coverage based test suite optimization. However, code coverage is not sufficient to guarantee a high fault detection rate in some cases. Whereas similarity based test suite optimization is one of the promising approaches for effective regression testing.
- Instead of using single coverage criteria, use of multiple coverage criteria for minimization and prioritization is quite beneficial to generate optimal test cases.
- Clustering approaches have been extensively applied to make the optimization process more effective.
- The survey on other similarity-based techniques is an eye-opener to apply them for test suite optimization problem which is also NP-hard.

Chapter 4 A New Similarity-based Greedy Approach for Generating Effective Test Suite

4.1 Introduction

Software testing is the most commonly used but expensive method for developing quality software by validating the software program [17]. The goal of software testing is to execute the software system, identify the faults that cause failures, and improve the software quality by removing the identified faults. It is a very expensive process as well as the important task of the software development life cycle (SDLC), through which we add some value to the software program [32, 74].

According to the IEEE definition [20], a test case is a collection of input data given to the program and expected output results created to evaluate a software function or test requirement. It is hard for a single test case to satisfy the coverage of entirely given test requirement. Because of the extensive use of testing to measure the quality of the software, one of the challenges faced by organizations is the test suite optimization [36]. Software regression testing processed continuously during the software development and maintenance of evolving software. Maintenance requires some modifications, which leads to growth in software and it results in an increment in test suite size. Over time, some test cases in a constructed test suite may become redundant, because the test cases created specifically for some selected testing criteria may also satisfy other requirements, and a requirement may still satisfy by some of the proper subsets of the test suite. Two test cases are termed as duplicate or redundant if their satisfied testing objectives are same. On the other hand, some of the test cases are termed as essential if their testing objective is unique. So, the prime objective is to remove the duplicate test cases and extract the essential or diverse test cases to generate the optimal test suite.

With the aim of optimizing the test suite, many researchers proposed different regression testing techniques; such as test case selection, minimization, and prioritization techniques by using different approaches. Test suite minimization is a process that tries to identify similarity and diversity between test cases and then accordingly remove the obsolete or duplicate test cases from the test suite. Test case selection mainly concerned with the problem of choosing a subset of test cases that will be used to verify the modified parts of the software. At last, test case prioritization concerns the finding of the `ideal' ranking of test cases that improves desirable coverage properties, such as early fault detection. Majority of the existing minimization tools and framework consider code coverage information of the software to be tested as a base to determine the minimized test suite. Coverage based test suite reduction techniques have gained wide consideration but they do not always give a satisfactory output. Empirical studies, however, reveals that code coverage may not the strong criteria for test suite effectiveness [27]. To address this problem, a number of techniques and framework have been proposed to make the reduction process more effective which is based on test case classification according to similarity degree measured by a distance function [73]. Diversity and similarity-based test case selection and prioritization are one of the new approaches with favorable output [91]. So, the study suggests that using the greedy technique with similarity-based approach could be a better option for effective regression testing.

In this chapter, a similarity-based greedy approach is proposed to overcome the limitations of existing greedy techniques for test suite reduction. The main idea of the proposed approach is to analyze the similarity degree among test case pairs and systematically remove them by applying an enhanced greedy algorithm while maintaining test requirements coverage. The clustering approach is also considered in this work, which could simplify the further optimization process. We divided the proposed approach into different phases, and each phase helps to get an optimal representative test set. However, rather than using single coverage criterion, here we used multiple criteria i.e. block, control flow, def-use, and data flow to compute the similarity (distance) values for each test case pair. With the help of distance values, cluster of test cases are generated accordingly. And, on each cluster combination of regression testing techniques i.e. test case minimization with prioritization is applied (testers will decide the order of execution). In order to evaluate the quality and

effectiveness of the proposed approach, we performed experiments on a sample subject programs. We also implemented the well-known standard greedy approach i.e. HGS algorithm [41]; to compare the results of optimized test suites using our similarity-based test suite optimization algorithm with those of minimizing test suites using the HGS algorithm.

4.2 Background

Different approaches including test case selection, minimization, and prioritization aimed at finding the optimal representative test set that reduces the cost and effort required for regression testing. A complete survey of these approaches and critical investigation on different TSR (Test Suite Reduction) framework and tools are discussed in detail by Yoo and Harman [37] and Khan et al. [26].

The proposed approach involves the combination of different regression testing techniques such as test case selection, minimization, and prioritization and employs agglomerative hierarchical clustering approach as well as to produce an optimal solution. The main aim is to find an optimized representative test set by concentrating on multiple regression testing techniques rather than using the single technique. Khan et al. combines the reduction and prioritization to get minimal and ordered test suite. However, other previous work did not consider any potential combination of regression testing techniques.

Despite the above-discussed techniques researchers are also working on a similarity-based approach. The main aim of using similarity-based approach is to increase the diversity of test suite for regression testing. Different distance measures were used for this purpose. With the help of any distance measure, we can evaluate how much the two test cases are different or similar to each other. All the existing optimization techniques have their common objective i.e. to optimize the test suite. But, they have used only single coverage criterion to compute the similarity degree between test case pairs. And the removal of such test cases by their similarity degree based on certain single coverage criterion may suffer from the quality of the test suite generated and overall fault detection ability may also reduce. The proposed approach presented a systematic

way to get the desired output by considering different coverage criteria with a similarity-based approach to minimize and prioritize the test cases.

4.3 Proposed approach

The proposed approach helps the testing team to determine an optimal and ordered test cases. By using different coverage criterion, the comparison metrics are calculated first, and the similarity matrix is generated accordingly. With the help of the calculated signature values, cluster of test cases is generated. On each cluster, an optimization technique is executed. According to the tester requirement, the optimization technique could be selected; it means either the minimization is executed before prioritization or vice versa. The proposed approach consisting of three phases: (1) Test case analysis (2) Clustering and (3) Optimization. Where Figure 4.1 shows the flow diagram of the proposed similarity based greedy approach. In the following, we briefly discuss all the three phases.

a) Test Case Analysis

The required inputs for this phase are - Program source code, Test suite, and Test case coverage metrics. The test case coverage metrics are used to compare any pair of test cases in a quantifiable manner. Four metrics used in this work are Block coverage equivalence, Control-flow divergence, def-use equivalence and data flow divergence. A first metric measures the block testing overlap between two test cases of a test suite. Second metric control-flow divergence measures the similarity of two test cases that test the same blocks that have conditional path within them. Third metric DU equivalence measures def-use path testing overlaps between two test cases in a test suite. And, the last metric Data divergence measures the similarity or diversity of test cases by data values used by test cases for code variables. In this phase, all the four metrics values are calculated. After getting these values, similarity and diversity values of test case pairs are calculated. In this phase, test cases are analyzed and selected based on their similarity and diversity values. In selecting test cases the rigid, sensitive and relaxed divergence threshold values are (0), (0.02) and (0.05) respectively. Similarly, rigid, sensitive and relaxed equivalence threshold values are one (1), (0.95) and (0.9). Here,

the test cases which satisfy the chosen (according to user needs) divergence and equivalence threshold values are coming under most similar test cases otherwise termed as diverse test cases. With the help of calculated similarity values, a clustering approach will be applied to categorize the test cases accordingly.

b) Clustering

For clustering, we applied an agglomerative hierarchical clustering approach which is based on the distance between test case pairs [92]. Test cases are grouped into a cluster with the help of their similarity value as measured in the analysis phase.

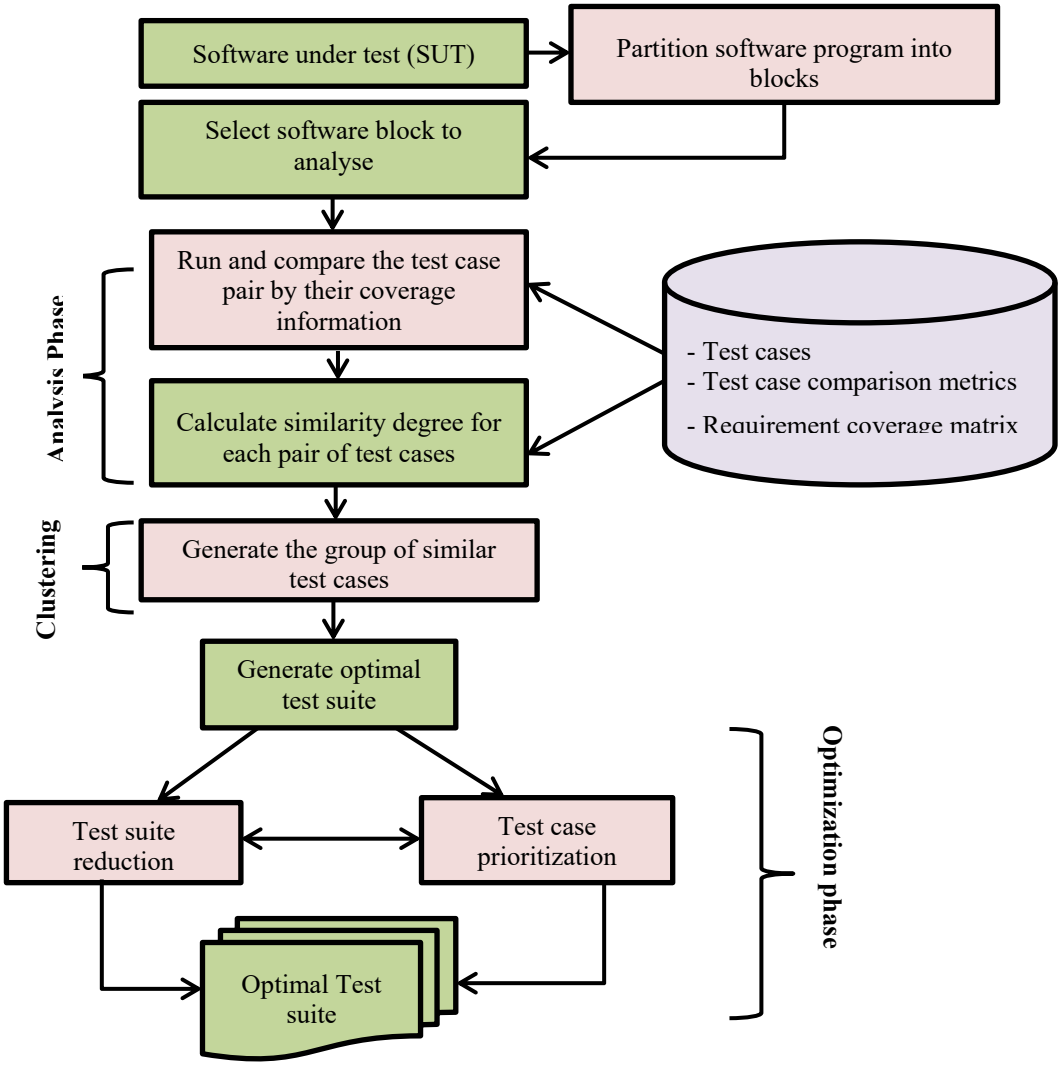


Figure 4.1 Block diagram for effective test suite generation

For making a group of similar and diverse test cases, threshold values of divergence and equivalence as 0.05 and 0.90 are assumed. Only those test cases having divergence value less than (0.05) and equivalence value greater than (0.90) comes under the similar or duplicate group and the remaining test cases are selected for a diverse group.

c) Optimization

On each cluster, a combination of optimization technique (Reduction and Prioritization) is applied to get the refined result. As discussed earlier, users are free to select the execution order of combination. If they use the combination of both the techniques, it helps them to determine the execution order of test cases as well as they get improved fault detection rate. Note that we have employed Enhanced Greedy Algorithm (CMIMX technique) [39] for test suite reduction followed by prioritizing the test cases by their total number of the unique path coverage.

These are the following steps to prioritize the test cases:

- Calculate the number of unique paths covered by test cases as compared to other test cases: unqcov (t).
- Greater value of $|unqcov(t)|$ will be assigned highest priority. Arrange all the test cases accordingly. To break the tie the test case having highest $|unqcov(t)|$ is selected.

After applying the optimization technique, the testing team will get the minimized as well as an ordered test suite, which is termed as an optimized test suite.

4.3.1 Algorithm of SBGA

The proposed SBGA algorithm comprises of the following steps which are illustrated below:

Step 1: Generate test cases for the subject program under consideration.

Step 2: Obtain coverage requirement according to the selected test coverage criteria. Here the selected coverage criteria are Block, Control flow, def/use, and data flow.

Step 3: For each selected criteria, generate the coverage matrix accordingly.

Step 4: With the help of criteria coverage matrix calculate the commonality and diversity values for each test case pair which determine that how much the test case pair are common and diverse to each other.

Step 5: Generate a Test Case Distance matrix to compare the pair of test cases.

Step 6: Apply hierarchical agglomerative clustering to generate a group of similar test cases.

Step 7: On each cluster apply the optimization process that comprises of minimization and prioritization approach. Apply CMIMX procedure for test suite minimization. Prioritize the test cases on the basis of a number of unique path coverage of test cases.

Step8: Construct the optimal representative test suite by merging the optimal cluster of test cases.

Algorithm 1: Similarity-Based Greedy Algorithm (SBGA)

Input: N: Total number of test cases

$R_{criteria}$: Requirement coverage information in terms of the selected criteria i.e. blocks(R_{Br}), Control-Flow(R_{CF}), Def-Use(R_{DU}), Data Flow (R_{DF}) coverage, and Path (R_P) coverage.

Output: Similarity Matrix

Begin

/* Calculate block coverage commonality */

1. X ← Number of common blocks between each pair of test cases;
2. Y ← Number of unique blocks tested by both test case pair;
3. **for** each test case $(T_i \leq N)$ **do**
4. **for** each test case $(T_{i+1} \leq N)$ **do**
5. $Block\ Coverage\ Metrics\ M_B\ (T_i\ , T_{i+1}) = X/Y$;
6. **end for**
7. **end for**

/* Control Flow (CF) is calculated for each test case that executes block B with a conditional branch. */

8. $n(T)$ ← Number of times true branch is executed

9. $n(F) \leftarrow$ Number of times false branch is executed

/* Calculate the total number of times each branch is executed */

10. **for** each test case $(T_i \leq N)$ **do**

11. $Sum(T,F) \leftarrow n(T) + n(F)$;

/* Calculate the average number of times each branch is executed */

12. $Avg(T,F) \leftarrow Sum(T,F)/Total(Br)$;

13. $CF_B(T_i) \leftarrow ((n(T) - Avg(T,F)) + (Avg(T,F) - n(F)))/Sum(T,F)$;

14. **end for**

/* Variance of CF values is calculated for each test case pair that executes common blocks with conditional statements */

15. **for** each test case $(T_i \leq N)$ **do**

16. **for** each test case $(T_{i+1} \leq N)$ **do**

17. $n\ (common(T_i, T_{i+1}))$ = Number of common shared block for each test case pair;

18. $M = Mean(CF_B(T_i), CF_B(T_{i+1})$;

/* Calculate the variance of CF (Control Flow) value for a common block B for each test case pair */

19. $\Delta B(T_i, T_{i+1}) \leftarrow (CF_B(T_i) - M)^2 + (CF_B(T_{i+1}) - M)^2$;

20. $Control\ Flow\ Coverage\ Metrics\ M_{CF}\ (T_i\ , T_{i+1}) = \sum \Delta B(T_i, T_{i+1})/n\ (common(T_i, T_{i+1})$

21. **end for**

22. **end for**

/* Calculate def-use coverage equivalence */

23. **for** each test case $(T_i \leq N)$ **do**

24. **for** each test case $(T_{i+1} \leq N)$ **do**

25. W= No. of common def-use chain tested by test case pairs

26. Z = No. of unique def-use chain tested by both test case pairs

27. $def - use\ Coverage\ Metrics\ M_{DU}\ (T_i\ , T_{i+1}) = W/Z$;

28. **end for**

29. **end for**

/* For each test case that executes a block B, a data flow (DF) diversity value is calculated */

30. **for** each test case ($T_i \leq N$) **do**

31. $DF(T_i, B) \leftarrow (n(T) - n(F))/2$;

32. **end for**

/* Calculate the variance of DF value for a common block B with loop statements */

33. **for** each test case ($T_i \leq N$) **do**

34. **for** each test case ($T_{i+1} \leq N$) **do**

35. $n\ (common(T_i, T_{i+1}))$ = Number of common shared block for each test case pair;

36. $M = Mean(DF_B(T_i), DF_B(T_{i+1})$;

37. $\Delta B(T_i, T_{i+1}) \leftarrow (DF_B(T_i) - M)^2 + (DF_B(T_{i+1}) - M)^2$;

38. $Data\ Flow\ Coverage\ Metrics\ M_{DF}\ (T_i\ , T_{i+1}) \leftarrow \sum \Delta B(T_i, T_{i+1})/n\ (common(T_i, T_{i+1})$;

39. **end for**

40. **end for**

/* Generate Test Case Distance matrix to compare the pair of test cases */

41. **for** each test case ($T_i \leq N$) **do**

42. **for** each test case ($T_{i+1} \leq N$) **do**

43. $M_{Dist}(C\ V[T_i\ , T_{i+1}]) \leftarrow (M_B + M_{DU})/2$; /* Commonality Values

44. $M_{Dist}(D\ V[T_i\ , T_{i+1}]) \leftarrow (M_{CF} + M_{DF})/2$; /* Diversity Values

45. **end for**

46. **end for**

47. **select_optimal (**$M_{Dist}(C\ V[T_i\ , T_{i+1}])$, $M_{Dist}(D\ V[T_i, T_{i+1}])$**)**

48. **end SBGA**

function **select_optimal (**$M_{Dist}(C\ V[T_i\ , T_{i+1}]), M_{Dist}(D\ V[T_i, T_{i+1}])$**)**

Begin

1. Apply agglomerative clustering;

2. **if** ($selection == 1$) **then**

3. On each cluster, apply enhanced greedy algorithm (CMIMX);

4. **for** each test cases T_i **do**

5. Calculate number of unique coverage for each test cases: $unqcov_p(T_i)$;

6. Ranked the test cases based on highest $unqcov_p(T_i)$ to lowest;
7. **end for**
8. **else if** $(selection == 2)$ then
9. Apply the step 4, step 5, step 6 and then step 3 respectively;
10. **else** apply step 3 onwards; **endif**
11. **endif**
12. end **select_optimal**

4.4 Experimental Results and Discussion

The performance evaluation of the proposed algorithms and state-of-the-art algorithms has been performed on different subject programs with different coverage criteria (see Table 4.10). The subject programs considered for this work have been well structured and without any compilation errors. The proposed algorithm and state-of-the-art algorithms are independent of the programming language. The program description is shown in Table 4.1.

The four coverage criteria i.e. Block, Control flow, def-use and data flow are used to assess the level of similarity between the test case pair. Alternatively, path coverage is used for optimizing a test suite. Hand seeded faults were used for the subject program. The representative optimized test set obtained for the subject program is analyzed for test suite size and fault detection effectiveness.

We also implemented the state-of-the-art algorithm i.e. HGS and conducted similar experiments to compare the results of the proposed algorithm and the conventional algorithm. For the HGS algorithm, we have taken each coverage criteria independently for measuring their adequacy. Moreover, for the proposed algorithm we have taken multi-coverage criteria. To show the importance of combining prioritization with minimization, we have compared the proposed prioritization approach with random prioritization.

4.4.1 Performance Measures

Software test metrics are the most important in evaluating the test cases and their quality of the testing activity and it plays an important role in assessing attributes that are very important to the success of any software. One of the main components of experiments to be considered is the variables. They categorize the elements into dependent and independent variables. The variables used in the presented work are defined as follows:

Independent variables: subject program, coverage criteria, seeded faults, requirement coverage matrix and similarity function.

Dependent variables: suite size reduction (SSR), fault detection loss (FDL) percentage and average percentage of faults detected (APFD) Percentage.

This section describes key measurements that are widely used to compare different test suite minimization techniques. The primary motivation for measuring these quantities is to control them to avoid conflating effects.

a) Suite Size Reduction (SSR) Measurement

Suite size reduction (SSR) percentage implies the number of test cases removed from the original suite.

$$SSR = \frac{|TS| - |MS|}{|TS|} \times 100\ \% \tag{4.1}$$

Where, $|TS|$ represent the number of test cases in the original test suite and $|MS|$ is the number of test cases in the minimized test suite.

b) Fault Detection Loss (FDL) Measurement

Fault detection loss (FDL) percentage signifies the total number of faults revealed by the minimized test suite.

$$FDL = \frac{|F| - |F_m|}{|F|} \times 100\ \% \tag{4.2}$$

Where, $|F|$ represents the total number of distinct faults revealed by the original test suite and $|F_m|$ is the number of distinct faults exposed by the minimized test suite.

Table 4.1 Subject program description

Program No.	Subject Programs	Program Description
Pr. 1	Prime number	To determine whether a number is prime or not
Pr. 2	Pushdown	It pushes the element down through its descendants by a sequence of swaps to its proper position
Pr. 3	Triangle	Return the type of a triangle by three integers
Pr. 4	Leap Year	To determine whether the year is a leap year or not
Pr. 5	Greatest Number	To find the largest number among the three numbers
Pr. 6	Number of digits	To calculate the number of the digits of a given number
Pr. 7	Odd/even	To determine whether a number is odd/even
Pr. 8	Quadratic equation	Find all roots of a Quadratic equation
Pr. 9	Calc	A simple calculator to add, subtract, multiply and divide
Pr. 10	Bubble sort	Sort by repeatedly stepping through the list

4.4.2 Case Study

In order to demonstrate the effectiveness of the proposed approach, a sample case study has been used. It is a C++ program that accepts any number from the user and determines whether the given number is a prime number or not. The case study is discussed in detail. The program (see Table 4.1) is taken as input for the proposed approach. The source code is transformed into the control-flow graph. The flow graph of the given program is shown in Figure 4.2. Figure 4.2 (a) illustrates the blocks presented in the program, which helps in identifying the block coverage information for each test case by executing them. In Figure 4.2 (b), the numbers such as 1, 2, 3, and so

on represent nodes and $P_1, P_2, P_3, \ldots, P_n$ represents the path number respectively. Similarly to gather other coverage information i.e. Def-Use, Control-Flow and Data flow, the different control flow graphs are generated indicating blocks, def-use etc. Based on the obtained coverage information, the test case coverage matrix is constructed to map the test cases with the coverage requirement. The path covered by the test cases are shown in Table 4.4 (5 *13 input matrix). One (1) and zero (0) represents paths covered and not covered respectively. The test cases are generated randomly for execution the of all selected coverage requirement. In order to collect all the selected coverage information for each test case, the program was hand instrumented. Table 4.4 represents the path coverage matrix for the sample test cases. We have randomly extracted some of the test cases for the given program to validate the proposed work (see Table 4.3). The proposed approach also uses hand seeded faults [65] for the subject program that are represented as $F_1, F_2, F_3, \ldots, F_n$ in Table 4.2, where 'n' signifies the number of injected faults.

Table 4.2 Sample program and injected faults

S. No.	Original Program	Fault No.	Injected Faults
1	#include <iostream.h>		
2	int main()		
3	int num i;		
4	cout << "enter a number \n";		
5	cin >> num;		
6	if(num%2==0)	F1	if(num%2>=0)
7	cout<< "even\n"; else		
8	cout<< "odd\n";		
9	if(num == 1)	F2	if(num<1)
10	cout<< "prime\n"; else		
11	{for (i=2; i ≤ num/2; ++i)	F3	{for (i=2; i ≤ num/2; --i)
12	if(num%i==0)	F4	if(num%i==1)
13	{cout<< "not prime\n";		
14	goto lb;}		

15	cout<< “prime\n”;}		
16	lb;		
17	return 0; }		

Table 4.3 Sample test cases

Test Cases	Input	Expected Output
T1	7	Odd, Prime
T2	2	Even, Prime
T3	6	Even, Not Prime
T4	15	Odd, Not Prime
T5	1	Odd, Prime
T6	10	Even, Not Prime

The generation of an effective test suite is carried out in three essential phases as discussed above. For each test case pair, the coverage metrics are calculated using the SBGA algorithm, where the values are shown in Table 4.5. Here, M_B , M_{CF}, M_{DU} and M_{DF} represents coverage metric value for block, control-flow, def/use and data flow coverage. After applying TCCA algorithm, the Commonality (Comm.) and Divergence (Div.) signature values are calculated for each pair of test cases (see Table 4.5). For example, test case pair T1-T2 have the divergence signature value and equivalence signature value is (0.69) and (0.83) respectively. With the help of these values, cluster of similar test cases are generated in subsequent phases. The next step is optimization of the test suite that further comprises of minimization and prioritization techniques. We apply Similarity Based Greedy Algorithm (SBGA) on collected coverage metrics for each pair of test cases. According to the agglomerative clustering approach, we have created four groups i.e. diverse, relaxed, sensitive, and duplicate (see Table 4.7). To remove some conflicts, we merge them with each other and creates only two groups i.e. diverse and similar group. Here T1, T2, and T5 grouped in one cluster C1 and the remaining test cases T3, T4 and T6 arranged in another cluster C2. During the optimization process, the selected combination of optimization technique applied on the generated clusters.

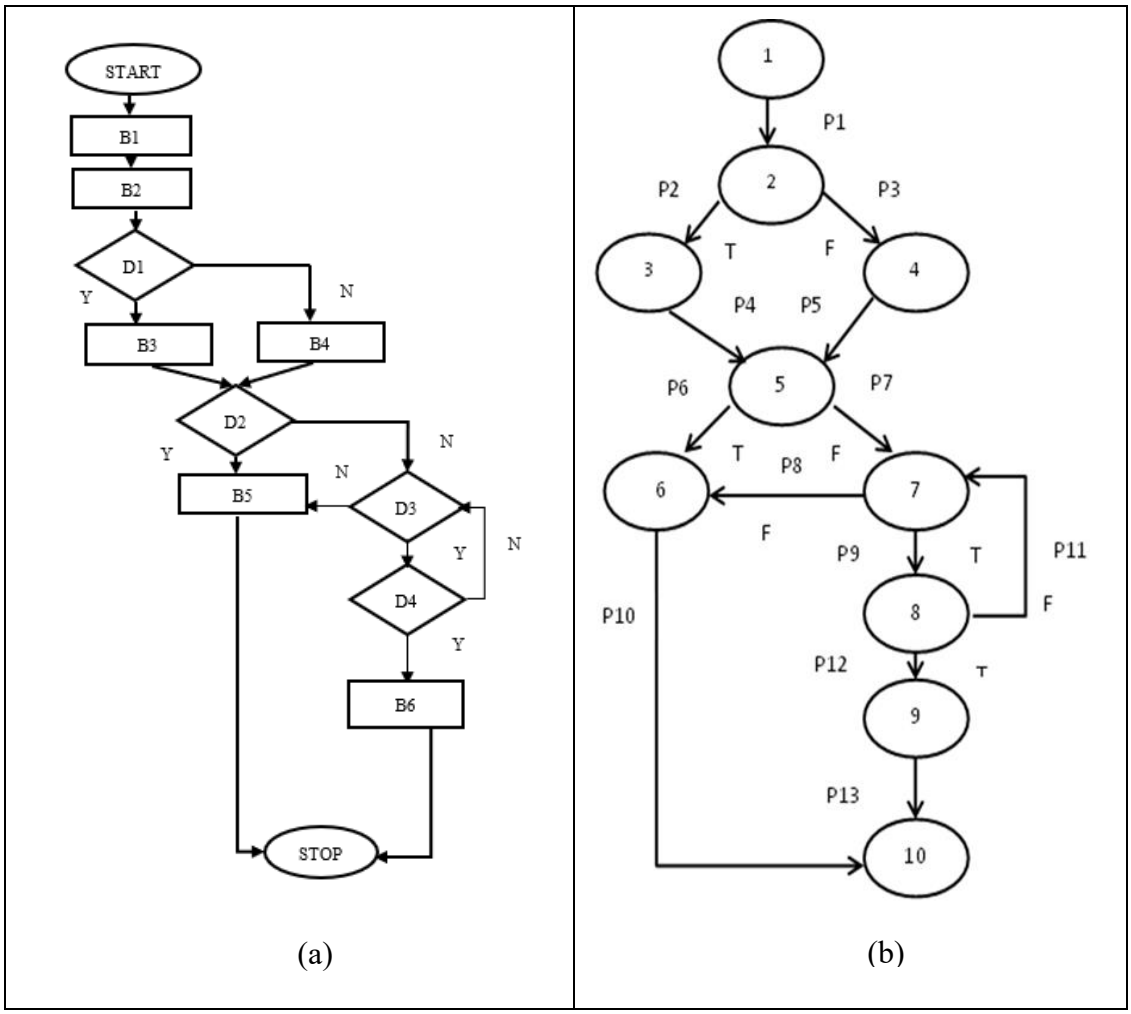


Figure 4.2 Control flow graph of the source program indicating (a) Block coverage and (b) Path coverage

Table 4.4 Path coverage matrix

Test Cases	Path Coverage												
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**P13**
T1	1	0	1	0	1	0	1	1	1	1	1	0	0
T2	1	1	0	1	0	0	1	1	0	0	0	0	0
T3	1	1	0	1	0	0	1	0	1	0	0	1	1
T4	1	0	1	0	1	0	1	0	1	0	1	1	1
T5	1	0	1	0	1	1	0	0	0	1	0	0	0
T6	1	0	1	0	1	0	1	1	1	1	1	0	0

Table 4.5 Coverage metrics value for test case pairs

Test cases		Coverage metrics values					
		T1	T2	T3	T4	T5	T6
T1	M_B	0.00	0.66	0.42	0.66	0.50	0.42
	M_{CF}	0.00	0.88	0.22	0.22	–	0.22
	M_{DU}	0.00	1.00	0.75	0.75	0.66	0.75
	M_{DF}	0.00	0.50	0.50	0.12	–	0.50
T2	M_B	0.66	0.00	0.66	0.42	0.28	0.66
	M_{CF}	0.88	0.00	2.00	2.00	–	2.00
	M_{DU}	1.00	0.00	0.75	0.75	0.66	0.75
	M_{DF}	0.50	0.00	0.00	0.12	–	0.00
T3	M_B	0.42	0.66	0.00	0.66	0.28	1.00
	M_{CF}	0.22	2.00	0.00	0	–	0
	M_{DU}	0.75	0.75	0.00	1.00	0.50	1.00
	M_{DF}	0.50	0.00	0.00	0.12	–	0.00
T4	M_B	0.66	0.42	0.66	0.00	0.50	0.42
	M_{CF}	0.22	2.00	0.00	0.00	–	0.00
	M_{DU}	0.75	0.75	1.00	0.00	0.50	1.00
	M_{DF}	0.12	0.12	0.12	0.00	–	0.12
T5	M_B	0.50	0.28	0.28	0.50	0.00	0.28
	M_{CF}	–	–	–	–	0.00	–
	M_{DU}	0.66	0.66	0.50	0.50	0.00	0.50
	M_{DF}	–	–	–	–	0.00	–
T6	M_B	0.42	0.66	1.00	0.42	0.28	0.00
	M_{CF}	0.22	2.00	0	0.00	–	0.00
	M_{DU}	0.75	0.75	1.00	1.00	0.50	0.00
	M_{DF}	0.50	0.00	0.00	0.12	–	0.00

Table 4.6 Overall distance matrix for test case pairs

Test Cases	Distances	Signature Values					
		T1	T2	T3	T4	T5	T6
T1	Div.		0.69	0.36	0.17	0.00	0.36
	Comm.		0.83	0.58	0.70	0.58	0.58
T2	Div.			1.00	1.06	0.00	1.00
	Comm.			0.70	0.58	0.47	0.70
T3	Div.				0.06	0.00	0.00
	Comm.				0.83	0.39	1.00
T4	Div.					0.00	0.06
	Comm.					0.50	0.71
T5	Div.						0.00
	Comm.						0.39

Table 4.7 Test Case Pair Group

Test Cases	Test Case Pair Group Status					
	T1	T2	T3	T4	T5	T6
T1		Relaxed	Diverse	Diverse	Relaxed	Diverse
T2			Diverse	Diverse	Relaxed	Diverse
T3				Sensitive	Relaxed	Duplicate
T4					Relaxed	Sensitive
T5						Relaxed

1) **Reduction and then Prioritization:** We have a cluster of diverse and similar test cases, where, on each cluster, we apply CMIMIX procedure to minimize the cluster size by removing obsolete or redundant test cases. According to the CMIMX procedure, the following steps are illustrated below:

For cluster C1:

Step 1: minCov = ∅, YetToCov = 11.

Step 2: All three tests T1, T2, T5 and eleven entities are unmarked.

Step 3: As YetToCov > 0, execute the loop body.

Loop 1 (Step 3.1): P6, P9, and P11 entity contain a single one (1); hence qualify as the highest priority entity. LC = {6, 9, 11}.

Step 3.2: Among the unmarked tests, T1 and T5 cover entities in LC. Of these tests, T1 has the maximum coverage of LC entities i.e. 9 and 11. So, S = T1.

Step 3.3: minCov = {T1}, test T1 is marked and entities covered by T1 are also marked i.e. 1, 3, 5, 7, 8, 9, 10 and 11. So, YetToCov = 11-8 = 3.

Loop 2 (Step 3.1): Continues with the second iteration of the loop as YetToCov > 0. Among the remaining unmarked entities 2, 4, and 6 have identical costs. So, LC = {2, 4, 6}.

Step 3.2: Entities in LC are covered by unmarked tests T2 and T5. Between these tests, T2 covers 2 and 4, and T5 covers 6.

Among these test cases, T2 has the maximum benefit of two (2). So, S = T2.

Step 3.3: minCov = {T1, T2}. Test T2 and entities covered by the test are marked i.e.2 and 4. YetToCov = 3-2 = 1.

Loop 4 (Step 3.1): Continues with the second iteration of the loop as YetToCov > 0. Among the remaining unmarked entities, only P6 is left which is covered by only one test case i.e. T5. So, LC = {6} and S = T5.

Step 3.3: minCov = {T1, T2, T5}. Test T5 and entities covered by the test are marked. YetToCov = 1-1 = 0.

In cluster C1, all the three test cases are important as compared to each other so, there is no removal of any test cases are possible.

After applying minimization technique, we prioritize the test cases according to the proposed method of unique coverage.

Input: T (T1, T2, T5), entitiyCov = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}.

Calculate cov (t) and unqcov(t) for each test cases of cluster C1 (see Table 4.8). Hence, the output is PRTest = {T2, T1, T5}.

For cluster C2:

As we can clearly see in Table 4.6, the signature values of test case pair T3-T6 are 0.00 and 1.00 respectively. These values illustrate that the pair is redundant. So, we can remove any one of the test cases from cluster C2. We arbitrarily select T6 to remove from that cluster. After removal of duplicate test cases, we have only T3 and T4 for further optimization. Afterward, we apply CMIMX procedure as we used for cluster one. Finally, we get the optimized (minimized and ordered) clusters C1 (T2, T1, T5) and C2 (T4, T3). In turn, the overall optimized test suite contains T2, T1, T5, T4, and T3 respectively.

2) **Prioritization and then Reduction:** In this optimization approach, first of all, we prioritize the test cases in the cluster, and then we apply the minimization technique. After prioritization, in cluster one the value of PRTest becomes {T2, T1, T5}.

Table 4.8 Unique Paths Covered By Test Cases of C1

Test Cases	Path Covered (cov(t))	$\|cov(t)\|$	$\|unqcov(t)\|$
T1	1, 3, 5, 7, 8, 9, 10, 11	8	2 {9, 11}
T2	1, 2, 4, 7, 8, 12, 13	7	4 {2, 4, 12, 13}
T5	1, 3, 5, 6, 10	5	1 {6}

Table 4.9 Unique Paths Covered By Test Cases of C2

Test Cases	Path Covered (cov(t))	$\|cov(t)\|$	$\|unqcov(t)\|$
T3	1, 2, 4, 7, 9, 12, 13	7	2 {2, 4}
T4	1, 3, 5, 7, 9, 11, 12, 13	8	3 {3, 5, 11}

Table 4.10 Coverage criteria

Coverage Criteria	Description
Block Coverage	Measures the sequence of consecutive groups of statements
Control Flow Coverage	Measures the same block that has a conditional path between them
def-use Coverage	Measures a logic execution sequence in a block that defines and uses a variable
Data Flow Coverage	Measures the coverage with respect to data values used for code variables
Path Coverage	Measure the coverage of each possible routes in each function

Subsequently, the minimization technique (CMIMX) is processed to get the representative test suite in cluster one. Keep in mind that, always prefer the test case have the highest priority while applying minimization here. While considering cluster two (Table 4.9), after prioritization we get the PRTest as {T4, T3, T6}, and because the test case pair T3 and T6 are duplicate the minimized test suite or cluster becomes {T4, T3}. Then, we have combined the results of generated clusters and get the optimized test suite i.e. T2, T1, T5, T4, and T3 respectively. The result of this technique is similar to the previous one. But, if we take a larger set of test cases, results may defer accordingly.

4.4.3 Result and Discussion

As can be seen in Figure 4.3, for each coverage criteria the percentage of size reduction and fault detection loss are different for HGS algorithm. The main drawback of this algorithm is that they achieve high test suite size reduction with the sacrifice of significant loss in fault detection ability. However, the prime objective of any testing scheme is to reveal maximum faults. We observe from Figure 4.3 that suite size reduction with multiple coverage criteria by the proposed algorithm is lower than the HGS algorithm as expected. However, the proposed algorithm achieves zero fault detection loss since the reduced test suite retains all coverage (multi-coverage). Where the loss is very less than the corresponding HGS fault detection loss values. In the proposed approach we have mainly concentrated on fault detection ability of the reduced test suite.

In this work, the test cases are checked for the entire selected coverage criterion for redundancy rather than considering only a single criterion as HGS algorithm. Because while removing any duplicate test cases from the test suite, there might be a possibility that the removed test cases may not become duplicate with respect to another criterion. The user can also use a different set of coverage criteria to evaluate the similarity or difference between test case pair depending on their requirement. Figure 4.4 and Figure 4.5 illustrate the fault coverage percentage of ordered test cases based on random prioritization and proposed prioritization approach. And result reveals that our proposed approach is quite effective in terms of early fault detection as compared to random based approach. As we can clearly see in Figure 4.4 that 100% fault coverage is obtained at last while execution of test case T5. Despite that in Figure 4.5, 100% coverage is obtained at an early stage while executing the test cases on a priority basis. By comparing the proposed approach with the conventional approach (HGS), our approach identifies maximum faults as early as possible and show good improvement in fault detection loss (see Figure 4.3).

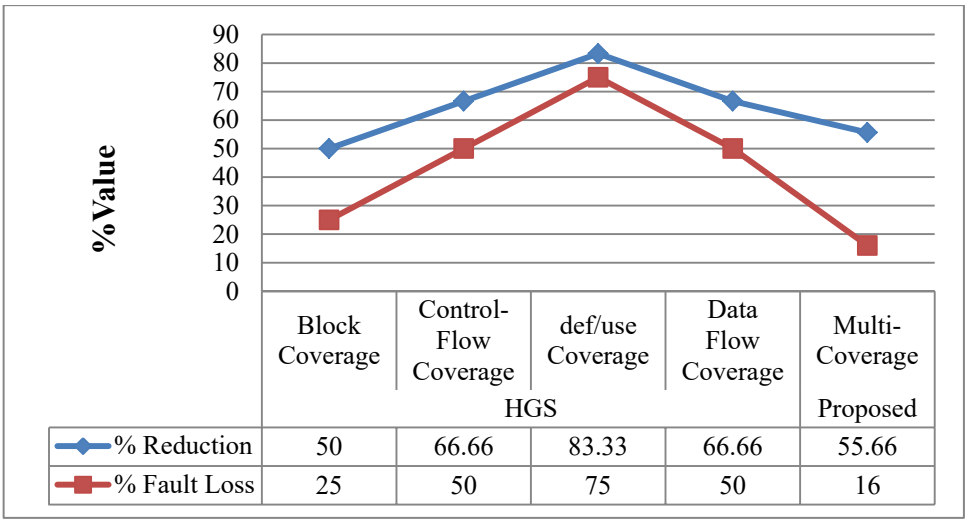


Figure 4.3 Analysis of SSR and FDL percentage for Prime Number (Pr. 1) program

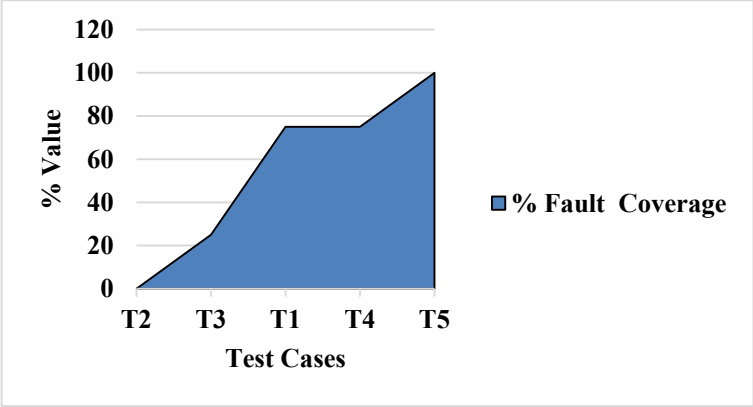


Figure 4.4 Fault coverage graph for random based prioritization of test cases

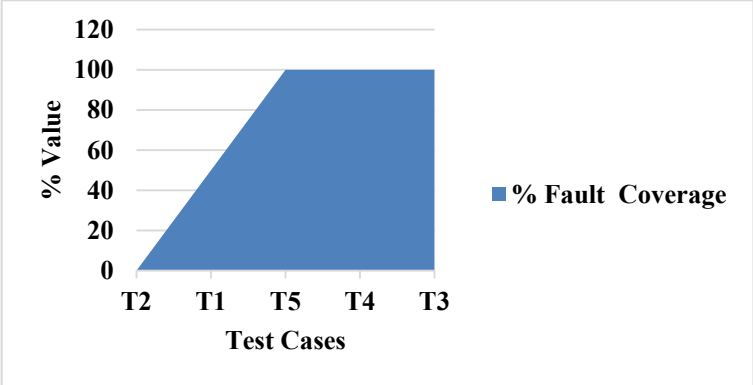


Figure 4.5 Fault coverage graph for the proposed prioritization approach

Therefore analysis reveals that identification of duplicate test cases based on any one single criteria and throwing them away is not a smart approach. Our experimental results clearly reveal that use of more than one criterion improves the quality of reduced and prioritized test suite in terms of coverage and fault detection ability. Our approach also provides the combinatorial approach for regression testing techniques i.e. the possible combination of minimization and prioritization. Overall the proposed similarity based greedy algorithm performs better than the existing coverage based technique HGS. If we have taken larger data set for validation of the proposed work, results would be quite different in terms of suite size reduction and fault detection loss.

The similarity-based greedy algorithm is carried out systematically using the procedure described in this chapter for other subject programs. The suite size reduction (SSR) and fault detection loss (FDL) for each program are illustrated in Figure 4.6 (a) and 4.6 (b). The y-axis in the graph represents the SSR and FDL against the ten subject programs considered in the x-axis. The average, maximum and minimum test suite size reduction and fault detection loss attained for the proposed SBGA algorithm with respect to the HGS algorithms with different criteria labeled as HGS_{BC} (block coverage minimization), HGS_{CF} (control-flow minimization), HGS_{du} (def/use minimization) and HGS_{DF} (data flow minimization) are presented in Table 4.11.

From Figure 4.6 it can be observed that test suite size reduction using SBGA is almost competitive with the state-of-the-art HGS algorithm. Table 4.11 states that when SBGA algorithm is used, the obtained average and maximum test suite reduction percentage is 67.6% and 75.5% respectively. These values are marginally less than HGS_{du} and HGS_{DF} algorithms but higher than HGS_{BC} and HGS_{CF}. So, overall the proposed approach is quite effective and competitive against HGS algorithm in suite size reduction. Figure 4.6 (a) and (b) also reveals that when the SBGA algorithm is used the fault detection loss for the subject programs ranges from 0.00% to 34%. The analysis suggests that the optimal test suite generated using SBGA algorithm detects number of faults than the state-of-the-art algorithm with minimum loss. From these figures, it has been concluded that the fault detection loss of the test cases is less in SBGA due to using multiple coverage criteria for determining similar test cases.

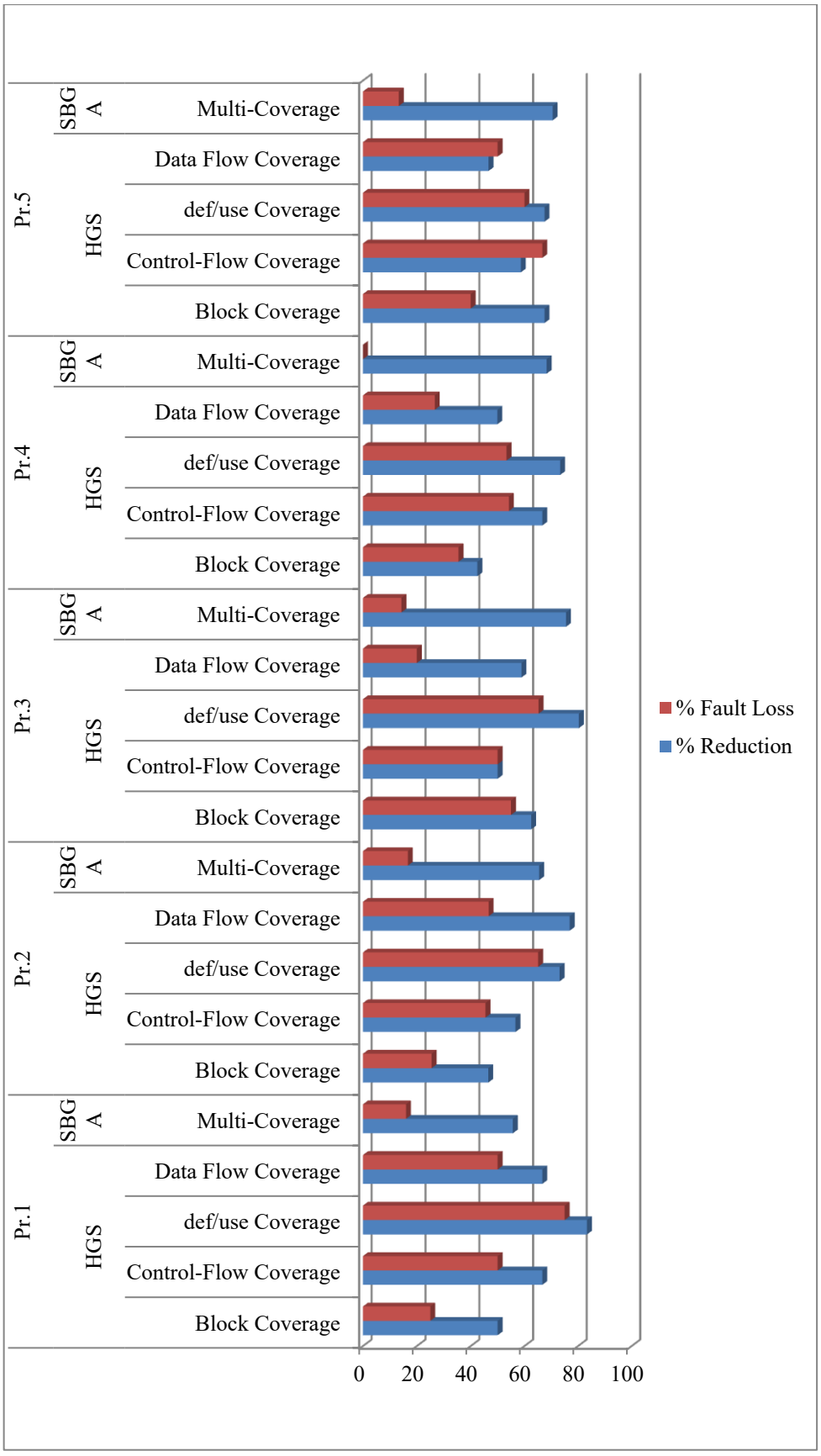


Figure 4.6 (a) Overall analysis of SSR (%) and FDL (%) for programs Pr.1 to Pr.5

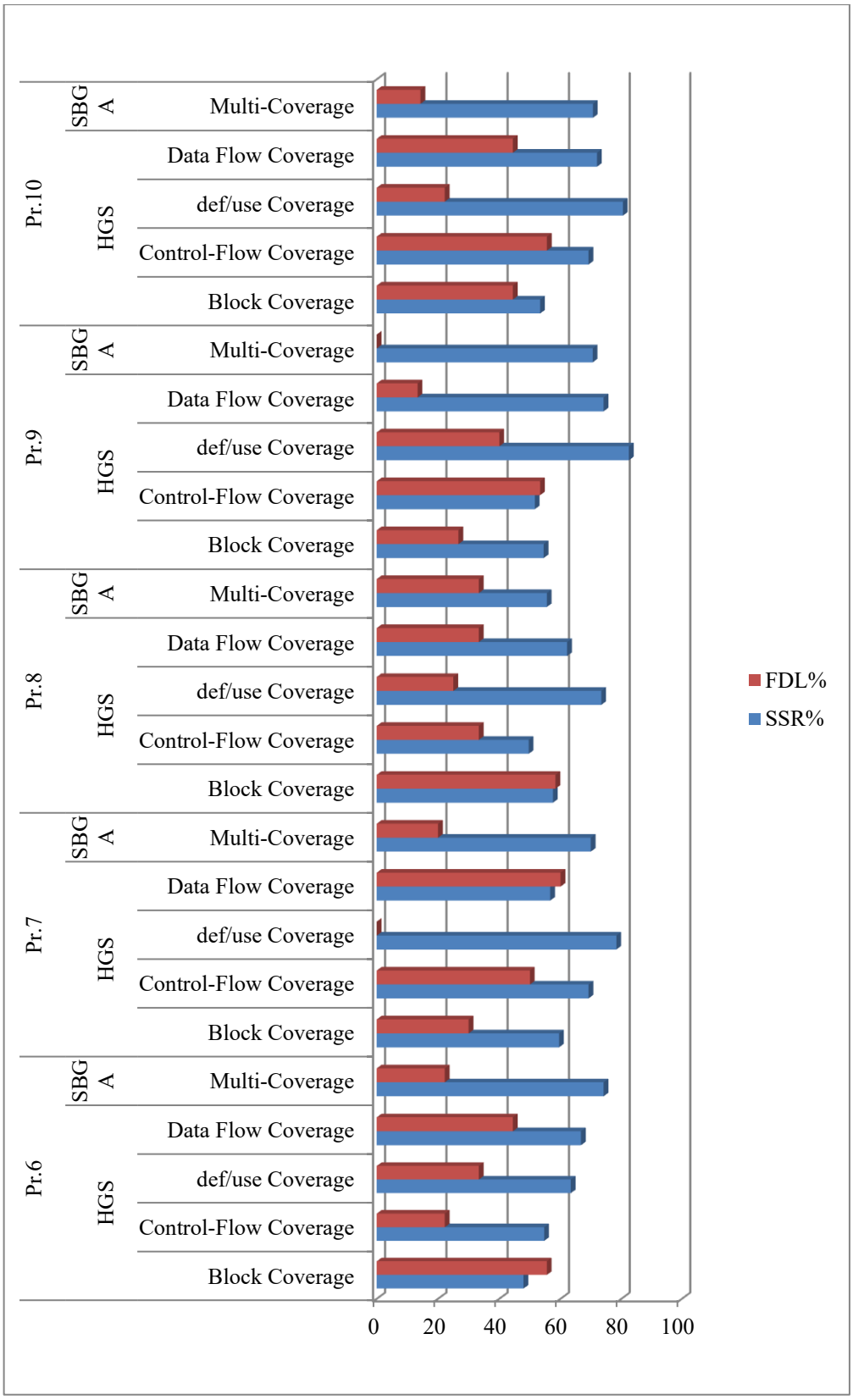


Figure 4.6 (b) Overall analysis of SSR (%) and FDL (%) for programs Pr.6 to Pr.10

Table 4.11 also summarizes the fault detection loss by the representative test set after experimentation with the proposed algorithm and state-of-the-art algorithm HGS. The observations made after experimentation show that SBGO algorithm provided an average of 15.01% fault detection loss against HGS_{BC} (39.59%), HGS_{CF} (48.07%), HGS_{du} (43.92%) and HGS_{DF} (38.89%) algorithms. Further, the obtained results recommend that the proposed SBGA algorithm consistently outperformed the HGS algorithm in fault detection loss.

Table 4.11 SSR and FDL values

	HGS								**SBGA (Proposed)**	
	HGS_{BC}		HGS_{CF}		HGS_{du}		HGS_{DF}		**Multi-Coverage**	
	SSR%	FDL%	SSR%	FDL%	SSR%	FDL%	SSR%	FDL%	SSR%	FDL%
Average (%)	54.17	39.59	59.27	48.07	75.55	43.92	63.03	38.89	67.60	15.01
Maximum (%)	67.5	58.33	69.12	66.66	83.33	75.00	76.9	60	75.5	33.33
Minimum (%)	42.5	25	49.6	22.22	63.33	0.00	46.6	13.33	55.50	0.00

There is a need for validation of any research work to provide certain assurance for the stability, accuracy, and robustness of the proposed system. For the purpose of showing statistical significance or validation of the proposed optimization approach, statistical analysis is carried out. The next section statistically evaluates and tests the trustworthiness of the proposed SBGA algorithms.

4.5 Statistical Validation

Statistical analysis is done to study the effectiveness of the proposed optimization approach. A hypothesis about the result of the proposed method or measurement is a wise assessment. Statistical Hypothesis testing is conducted to assess whether or not the SSR and FDL are improved in the optimal test set as compared to the existing techniques generated a test suite. Since the sample size is small, the student T-test is applied to find out the level of significance and reject the null hypothesis [115]. Meanwhile, the rejection or acceptance of a null hypothesis is based on either (0.05) alpha ($\propto$) or (0.01) alpha ($\propto$) level of significance for one tailed or two tailed test, (0.05) alpha ($\propto$) level of significance for a one-tailed test is taken for rejection of the null hypothesis. An important point in implementing the right statistical hypothesis test is to consider their notion [115]. The statistical analysis process comprises of some sequential steps: the first step begins with the creation of the null hypothesis and the alternative hypothesis. Here the comparison has been made using SSR and FDL percentage against the existing HGS algorithm and the proposed SBGA algorithm. And, by obtained results, it has been concluded that whether there is a significant difference between them or not. The obtained t value will determine whether to reject the null hypothesis and accept the alternative hypothesis.

A null hypothesis shows that there is no significant relationship between two or more parameters, while the alternative hypothesis proves relationships. The rejection of a null hypothesis provides a strong basis for accepting the relationship or accepting the alternative hypothesis [115]. This study relates improvement of suite size reduction and fault detection loss by using the similarity based approach with multiple criteria.

The formulated hypothesis is mentioned below:

Null Hypothesis **(H_{01}):** Percentage of Suite Size Reduction (SSR %) cannot be improved using the proposed approach as compared to HGS_{BC}.

Alternative Hypothesis **(H_{11}):** Percentage of Suite Size Reduction (SSR %) can be improved using the proposed approach as compared to HGS_{BC}.

Null Hypothesis **(H_{02}):** Percentage of Suite Size Reduction (SSR %) cannot be improved using the proposed approach as compared to HGS_{CF}.

Alternative Hypothesis **(H_{12}):** Percentage of Suite Size Reduction (SSR %) can be improved using the proposed approach as compared to HGS_{CF}.

Null Hypothesis **(H_{03}):** Percentage of Suite Size Reduction (SSR %) cannot be improved using the proposed approach as compared to HGS_{du}.

Alternative Hypothesis **(H_{13}):** Percentage of Suite Size Reduction (SSR %) can be improved using the proposed approach as compared to HGS_{du}.

Null Hypothesis **(H_{04}):** Percentage of Suite Size Reduction (SSR %) cannot be improved using the proposed approach as compared to HGS_{DF}.

Alternative Hypothesis **(H_{14}):** Percentage of Suite Size Reduction (SSR %) can be improved using the proposed approach as compared to HGS_{DF}.

Null Hypothesis **(H_{05}):** Percentage of Fault Detection Loss (FDL %) cannot be decreased using the proposed approach as compared to HGS_{BC}.

Alternative Hypothesis **(H_{15}):** Percentage of Fault Detection Loss (FDL %) can be decreased using the proposed approach as compared to HGS_{BC}.

Null Hypothesis **(H_{06}):** Percentage of Fault Detection Loss (FDL %) cannot be decreased using the proposed approach as compared to HGS_{CF}.

Alternative Hypothesis **(H_{16}):** Percentage of Fault Detection Loss (FDL %) can be decreased using the proposed approach as compared to HGS_{CF}.

Null Hypothesis **(H_{07}):** Percentage of Fault Detection Loss (FDL %) cannot be decreased using the proposed approach as compared to HGS_{du}.

Alternative Hypothesis **(H_{17}):** Percentage of Fault Detection Loss (FDL %) can be decreased using the proposed approach as compared to HGS_{du}.

Null Hypothesis **(H_{08}):** Percentage of Fault Detection Loss (FDL %) cannot be decreased using the proposed approach as compared to HGS_{DF}.

Alternative Hypothesis **(H_{18}):** Percentage of Fault Detection Loss (FDL %) can be decreased using the proposed approach as compared to HGS_{DF}.

By observing the average, maximum and minimum values of SSR (%) and FDL (%) in Table 4.11, it can be concluded very easily that the proposed SBGA algorithm is quite competitive against HGS algorithm in suite size reduction (SSR) and comparatively improved in fault detection loss (FDL). Experimental results represented through Figure 4.6 (a) and (b) show that the proposed approach followed for minimizing test suite size with maintaining their fault detection loss is very appropriate. But it is not sufficient and will not be able to prove alone the acceptance of the proposed approach. In general, the level of significance of proposed approach must be calculated to make it acceptable and for this purpose, t-test is found appropriate.

Level of Significance of SSR

To find out the significance of the difference between the HGS algorithm with different criteria labeled as HGS_{BC} , HGS_{CF} , HGS_{du} and HGS_{DF} against the proposed SBGA approach in terms of SSR, the means of both old and new SSR are calculated as shown in Table 4.12 (a) to 4.12 (d). Pearson coefficient of correlation shows that the old SSR values before treatment and new values of SSR after treatment are highly correlated. The degree of freedom for both SSR values is 9.

The t value comes out to be 4.546, 2.414, 2.389 and 1.111 respectively. As the initial three values exceed the t critical value for a two-tailed test at the 0.05 level for 9 degrees of freedom, the null hypothesis H_{01} , H_{02} and H_{03} are strongly rejected and the alternate hypothesis H_{11} , H_{12} and H_{13} are accepted. But in the case of last t value i.e. 1.111 for HGS_{DF} is less than the t critical value, the null hypothesis H_{04} is not strongly rejected and the alternate hypothesis H_{14} is rejected.

Hence it is validated that SSR value is improved by applying similarity based greedy approach with multiple criteria as compared to HGS_{BC} , HGS_{CF} and HGS_{du}. And as

compared to HGS_{DF} the SSR is marginally less improved by the proposed approach. Overall our approach is quite competitive to the state-of-the art algorithm in suite size reduction.

Table 4.12 (a) T-Test: Paired Two Sample for Means in terms of SSR (HGS_{BC} vs. SBGA)

Statistical Observation	HGS_{BC}	SBGA
Mean	54.17	67.605
Variance	59.89898	47.75185
Observations	10	10
Pearson Correlation	0.190029	
Hypothesized Mean Difference	0	
df	9	
t Stat	-4.54642	
P(T<=t) one-tail	0.000697	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.001393	
t Critical two-tail	2.262157	

Table 4.12 (b) T-Test: Paired Two Sample for Means in terms of SSR (HGS_{CF} vs. SBGA)

Statistical Observation	HGS_{CF}	SBGA
Mean	59.275	67.605
Variance	63.32932	47.75185
Observations	10	10

Pearson Correlation	-0.07262	
Hypothesized Mean Difference	0	
df	9	
t Stat	-2.41406	
P(T<=t) one-tail	0.019494	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.038989	
t Critical two-tail	2.262157	

Table 4.12 (c) T-Test: Paired Two Sample for Means in terms of SSR (HGS_{du} vs. SBGA)

Statistical Observation	HGS_{du}	SBGA
Mean	75.554	67.605
Variance	43.67785	47.75185
Observations	10	10
Pearson Correlation	-0.21105	
Hypothesized Mean Difference	0	
df	9	
t Stat	2.389048	
P(T<=t) one-tail	0.02031	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.040621	
t Critical two-tail	2.262157	

Table 4.12 (d) T-Test: Paired Two Sample for Means in terms of SSR (HGS_{DF} vs. SBGA)

Statistical Observation	HGS_{DF}	SBGA
Mean	63.038	67.605
Variance	102.4802	47.75185
Observations	10	10
Pearson Correlation	-0.13187	
Hypothesized Mean Difference	0	
df	9	
t Stat	-1.11198	
P(T<=t) one-tail	0.147483	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.294966	
t Critical two-tail	2.262157	

Level of Significance of FDL

To find out the significance of the difference between the HGS algorithm with different criteria labeled as HGS_{BC} , HGS_{CF} , HGS_{du} and HGS_{DF} against the proposed SBGA approach in terms of FDL, the means of both old and new FDL are calculated as shown in Table 4.13 (a) to 4.13 (d). Pearson coefficient of correlation shows that the old FDL values before treatment and new values of FDL after treatment are highly correlated. The degree of freedom for both FDL values is 9.

The t value comes out to be 6.736, 5.260, 3.200 and 5.619 respectively. As the values exceed the t critical value for a two-tailed test at the 0.05 level for 9 degrees of freedom, the null hypothesis H_{05}, H_{06}, H_{07} and H_{08} are strongly rejected and the alternate hypothesis H_{15} , H_{16} , H_{17} and H_{18} are accepted. Hence it is validated that the

performance of the test suite optimization in terms of fault detection loss (FDL) can be improved using similarity-based greedy approach.

The acceptance of any new approach by society or industry depends on the validation of that approach. This is the belief that proves the usefulness of the approach in society or industry. For testing the usefulness of the proposed approach for optimizing the test suite for effective regression testing, a systematic validation is carried out.

Table 4.13 (a) T-Test: Paired Two Sample for Means in terms of FDL (HGS_{BC} vs. SBGA)

Statistical Observation	HGS_{BC}	SBGA
Mean	39.593	15.01
Variance	172.528	96.74418
Observations	10	10
Pearson Correlation	0.526747	
Hypothesized Mean Difference	0	
df	9	
t Stat	6.736533	
P(T<=t) one-tail	4.25E-05	
t Critical one-tail	1.833113	
P(T<=t) two-tail	8.49E-05	
t Critical two-tail	2.262157	

Table 4.13 (b) T-Test: Paired Two Sample for Means in terms of FDL (HGS_{CF} vs. SBGA)

Statistical Observation	HGS_{CF}	SBGA
Mean	48.077	15.01
Variance	152.2008	96.74418

Observations	10	10
Pearson Correlation	-0.6024	
Hypothesized Mean Difference	0	
df	9	
t Stat	5.260405	
P(T<=t) one-tail	0.00026	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.00052	
t Critical two-tail	2.262157	

Table 4.13 (c) T-Test: Paired Two Sample for Means in terms of FDL (HGS_{du} vs. SBGA)

Statistical Observation	HGS_{du}	SBGA
Mean	43.926	15.01
Variance	567.0289	96.74418
Observations	10	10
Pearson Correlation	-0.32524	
Hypothesized Mean Difference	0	
df	9	
t Stat	3.20081	
P(T<=t) one-tail	0.005409	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.010817	
t Critical two-tail	2.262157	

Table 4.13 (d) T-Test: Paired Two Sample for Means in terms of FDL (HGS_{DF} vs. SBGA)

Statistical Observation	HGS_{DF}	SBGA
Mean	38.887	15.01
Variance	222.8069	96.74418
Observations	10	10
Pearson Correlation	0.473478	
Hypothesized Mean Difference	0	
df	9	
t Stat	5.619726	
P(T<=t) one-tail	0.000163	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.000326	
t Critical two-tail	2.262157	

4.6 Summary

In this chapter similarity-based greedy approach has been proposed for test suite optimization. The optimized test suite is used by testers for effective regression testing. The goal is to apply a similarity-based strategy with multiple criteria to identify the difference between a pair of test cases and compare them to similarity level. This work concentrates on the combination of regression testing techniques i.e. minimization and prioritization with different coverage criterion to optimize the test suite size. The main idea is to analyze the test cases first to know the difference and similarity value of the test case pairs and further apply the greedy and clustering approach to optimize the test cases accordingly. We have proposed the SBGA algorithm for optimizing the test suite. The proposed approach can be very helpful when the fault detection effectiveness is more important as compared to code coverage. The experiments are conducted on

different benchmark programs to evaluate the performance of the proposed approach. To evaluate the effectiveness of the proposed work two performance metrics were used i.e. SSR and FDL. And the experimental results show that the fault detection ability is highly improved by the proposed technique as compared to an existing technique.

The most essential stage of the research work is to validate the newly developed method. It is necessary to prove the approach to make it socially acceptable. In this chapter, the proposed work has been validated. Student T-test has been used to test the hypothesis. The observed values after t-test reveal that the proposed SBGA algorithm consistently outperformed the HGS algorithm in fault detection loss and quite competitive in suite size reduction.

Chapter 5 A New Similarity-Based Test Suite Optimization Framework Using Multiple Coverage Criteria

5.1 Introduction

Testing is the most commonly used but expensive method for demonstrating that a software program performs its intended function. It is a very expensive process as well as the important task of the software development life cycle (SDLC), through which we add some value to the software program. Adding some value means improving the quality and reliability of the program [116]. Due to some changes in customer requirements, the developer has to modify or update the existing software to release a new version of the software. Re-testing the modified software to verify whether new added features or changes have introduced errors into existing parts, compromising their behaviors is termed as regression testing. It requires additional test cases to check new features within the software program [3, 4]. Execute all the existing test cases together with newer ones are very time to consume and costly because of limited time and resources available. Additionally, the resulted test suite may contain obsolete and duplicate test cases which must be removed to get the optimal test suite. A test case in a test suite either said to be redundant or essential. Two test cases are termed as duplicate or redundant if their satisfied testing objectives (testing requirements or criteria) are same. On the other hand, some of the test cases are termed as essential if their testing objective is unique (not satisfied by remaining test cases). The main aim is to remove the duplicate test cases and find out the diverse test cases. With the aim of optimizing the test suite, many researchers proposed different regression testing techniques; such as test case selection, minimization, and prioritization by using different approaches.

Various approaches are coverage based, search-based, ILP based or similarity based. But, a recent study shows that a coverage criterion is not strong enough to solve regression testing problem [13]. For adequate regression testing, diversity-based test suite optimization is also one of the promising approaches with better results [14].

In this chapter, the similarity based test suite optimization framework is proposed with the help of clustering and prioritization to get an optimal representative test suite. The other major concern while performing minimization is coverage criteria. Most of the existing techniques are based on certain coverage criteria such as statement, branch, mc/dc, path etc. They used either one or two of these criteria to minimize the test suite. The study shows that no single coverage criteria test the whole program or performs best for all faults. Different kinds of faults are best identified by multiple coverage criteria [117]. In this paper, more than one coverage criteria are used to get an optimal result. Minimizing test suites while keeping all-uses coverage constant could result in minimum loss of fault detection effectiveness [43]. However, the previous empirical study recommends that after reducing test suites the fault detection capabilities can severely compromise [51]. The real challenge for researchers is to determine a minimal subset of essential test cases, which discovers a maximum number of faults similar to the original test suite. So, there must be an effective approach which must be capable of reducing the test suite size in such a manner that resultant test suite retains less FDE (Fault Detection Effectiveness) loss and maximum desirable code coverage.

The chapter introduces a novel approach that incorporates a similarity-based strategy with multiple coverage criteria to get an optimal result. The main contributions of the proposed approach are as follows:

- The proposed similarity-based approach is used to identify the most different test cases and eliminate the redundant one with the help of a similarity matrix. The approach uses three coverage information i.e. Statement coverage, MC/DC coverage and branch coverage to construct a similarity matrix by calculating the distance between test case pairs.
- The approach also utilizes an agglomerative hierarchical clustering method [92] which is applied to the obtained similarity matrix to create the desired number of clusters of similar test cases. And within every cluster, the duplicate and most similar test case pairs are identified according to their similarity degree.

- The chapter presents an optional use of prioritization to order the test cases belongs to minimized clusters with the help of their assigned weight. The weight is calculated according to their overall coverage detail.

To measure the quality and effectiveness of the proposed approach, we performed an experimental and statistical evaluation of different subject programs. We also implemented the well-known HGS and GRE algorithm [41, 114]; to compare the results of optimized test suites using our similarity-based test suite optimization algorithm with those of minimizing test suites using the HGS and GRE algorithm.

5.2 Background

Regression testing is a critical activity to ensure that the modified or updated software has sufficient quality and reliability. Thus, a fundamental problem of regression testing is to select a small set of diverse test cases having maximum coverage and higher FDE. For adequate regression testing, there must be an optimized test suite containing essential or diverse test cases have to be generated by discarding the duplicate or similar test cases from the original test suite. In this work to get an optimized result, we incorporated both the clustering and prioritization method with the similarity based strategy. This section provides a brief introduction about test suite minimization, coverage criteria, similarity-based test suite minimization and the clustering method used in the proposed approach for grouping of similar test cases.

5.2.1 Test suite optimization

To get an optimal test suite, we have considered both the test suite minimization and test case prioritization in our proposed similarity-based approach. Combining the two approaches provides the better result as compared to the existing one.

Test Suite Reduction

Test suite reduction aims to retain essential test cases and remove redundant ones from a test suite, generating an optimal test suite that is a minimal subset of the original test suite. Test suite reduction techniques aim to find duplicate test cases with respect to some testing criteria, such as statement, branch coverage, path, MC/DC etc. A test case

is either termed as essential or duplicate. Essential test cases are the reverse of duplicate test cases. If any test case satisfies the requirement r_i uniquely, the test case is termed as essean ntial test case. In contrary, if a test case covers equal requirements or only a subset of the test requirements covered by another test case, called as dupla icate test case. Most of testthe suite minimization problems can be stated as a minimal hitting set problem which is known as NP-complete [37].

According to Harrold et al. [41], the test suite minimization problem can be defined as follows:

Given: $\{t_1, t_2, t_3, \dots, t_n\}$ represents a test suite T containing n test cases and $\{r_1, r_2, r, \dots, r_k\}$ represents a set of testing requirements that must be satisfied in order to provide required coverage of the program and each subsets $\{T, T_2, T, \dots, T_n\}$ from T are associated to one of the r_i's such that each test case t_j related to T_i covers r_i.

Problem: Discover minimal test suite T' from T which satisfies all r_i's covered by the original test suite T.

Test Suite Prioritization

Test suite prioritization ranked the test cases within a test suite based on some criteria with the aim of maintaining fault detection effectiveness. The focus of test case prioritization is to maximize the probability of meeting certain objective (maximum coverage or higher fault detection rate) of the test cases when executed in a particular order. Under limited time and resource constraints, ordering the test cases may increase the testing effectiveness by executing the essential test cases as early as possible. The effectiveness of minimized test suites can be further improved or optimized by ordering the reduced test suite.

Test case prioritization problem can be defined as follows [2]:

Given: TS represents a test suite, PT represents a permutation of TS, and a function$f: PT \rightarrow R(real\ number)$.

Problem: to find $T' \in PT$ such that: $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

5.2.2 Test Coverage Criteria

The purpose of coverage criteria is to measure the adequacy of test suites and its quality. Given certain coverage criteria, *C* that is satisfied by one of the test cases *t* residing in the test suite *TS*, another test case t' of that suite is redundant with respect to *C* if the criteria *C* is also satisfied by that test case [83]. Thus, the process of removing test cases from a test suite that are redundant with respect to certain coverage criteria preserves the adequacy of the suite with respect to those criteria. But, redundant test cases that are selected to be removed from the test suite according to a specific criterion may not be redundant and may execute unique condition with respect to another criterion. The removal of such redundant test cases can be the reason of fault detection loss. This recommends that a combination of more than one testing criteria should be used for determining the optimal representative set, thus promoting the multi objective heuristics to solve the minimization problem effectively. The empirical study by Hemmati [98] concluded that a combination of control-flow coverage will always perform better than the best individual one. The effectiveness of test suite optimization techniques in identifying bug revealing test cases also increases from using single coverage criteria to using all coverage criteria together. Their finding suggests the use of multiple coverage criteria rather than focusing on maximizing just one. Inspired by their suggestions we have considered three coverage criteria i.e. statement, branch and MC/DC for the optimization process.

Statement Coverage: Every statement in the program has been executed at least once [19].

MC/DC Coverage: MC/DC requires that each condition in a decision be shown by execution to independently affect the outcome of the decision [19].

Branch Coverage: Every entry and exit point in the program has been called at least once, and every decision in the program has taken all possible outcomes at least once [19].

Selection of coverage criteria is a very important task for detecting a maximum fault as well as preserving maximum coverage. The tester or user can select other combination

of criteria's also such as def-use, path, function coverage etc. depending upon their requirement and the software to be tested.

5.2.3 Similarity-based Test Suite Optimization

The goal of the similarity based test suite optimization strategy is to select a number of different and essential test cases that are most different or similar based on the similarity values among them. The diversity or similarity degree of test cases is calculated by a certain distance measure between each pair of test cases. Therefore, this will maximize the fault detection rate if we choose the maximum number of diverse test cases in a test suite. The resultant test suite must satisfy the coverage requirements as the original test suite. Hence, the goal is to achieve maximum requirement coverage by the most diverse test cases. For this, the following inputs are required:

- **Test Suite:** The set of test cases that should be reduced;
- **Coverage Criteria:** the list of coverage criteria. For each test coverage criterion, a set of test requirements that should be covered from a reduced test suite is generated;
- **Similarity Function:** It presents the function used to calculate the similarity degree between two test cases. Cartaxo [31] proposed the Similarity Matrix to present the similarity degree between all pair of test cases of a test suite;

Cartaxo et al. [31] define a distance measure that computes the similarity degree between the test case pair specified as paths.

$$Similarity\ function[i,j] = \frac{nit}{Avg(|i|,|j|)} \tag{5.1}$$

Where, nit represents the number of identical transition and $Avg(|i|,|j|)$ represents the average transition between path lengths. With the help of the calculated similarity degree of each pair of test cases the square similarity matrix is created, where n is the number of paths and each path represented as i $(1 \leq i \geq n)$ is called a test case.

5.2.4 Agglomerative Hierarchical Clustering

The agglomerative hierarchical clustering [92] method is used in the proposed approach for clustering of similar test cases. The clustering algorithm is presented as Algorithm 1. It works based on the concept of pairwise similarity or diversity between two test cases. Initially, single test cases are placed in separate clusters. Then, the test case belonging to two clusters are selected as a pair and the similarity degree between their overall coverage capabilities is calculated. Based on the maximum similarity degree, clusters are merged. After grouping of all single test case clusters, the same process is continued for merging larger clusters until a preferred number of clusters are obtained or no other cluster pair are left of maximum similarity degree greater than or equal to the chosen threshold value. At each step, a table is maintained for keeping the updated similarity degree of test case pairs and then for merged clusters. The output of the clustering process is a dendrogram, which is a tree structure that represents the arrangement of clusters. Figure 5.1 shows an example of a dendrogram structure. It is possible to generate k number of clusters for any k in $[1 \leq k \geq n]$ by splitting the tree for different heights.

Algorithm 1: Agglomerative Hierarchical Clustering [92]
Input: A set of *n* test cases, *T* **Output:** A dendrogram, *D*, representing the clusters **Algorithm:** Form *n* clusters, each with one test case *C* Add clusters to *C* Insert *n* clusters as leaf node into *D* while there is more than one cluster Find a pair of clusters with the minimum distance Merge the pair into a new cluster, *new c*

Remove the pair of test cases from *C*

Add *new c* to *C*

Insert *new c* as a parent node of the pair into *D*

return *D*

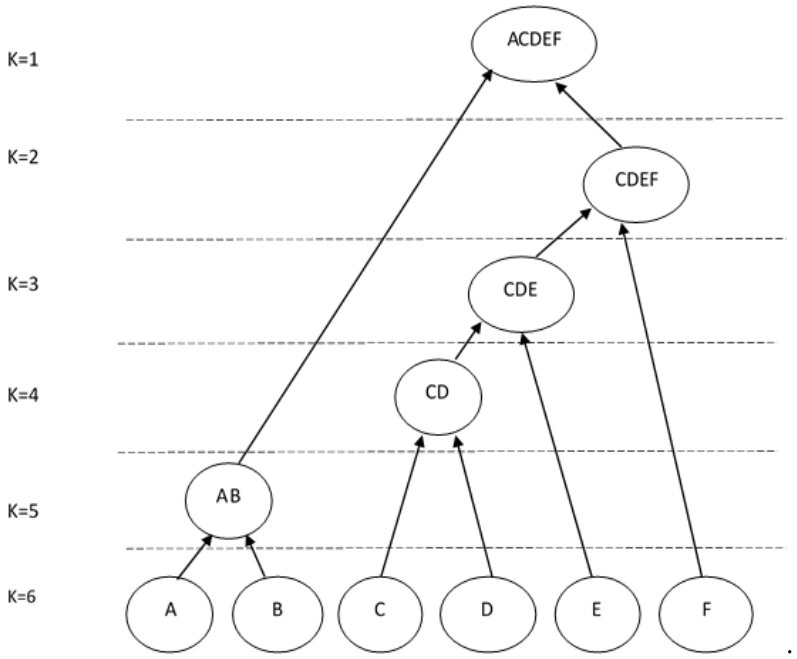


Figure 5.1 An example dendrogram for Agglomerative Hierarchical Clustering

5.3 Proposed Approach

Inspired by the similarity-based approach for test case selection and test suite reduction, we proposed the similarity based test suite optimization framework which is shown in Figure 5.2. The proposed approach for test suite optimization has been motivated by combining a similarity-based approach with multiple coverage criteria. The goal of this approach is to compare each test case pair by considering the similarity degree between them. To apply this strategy, multiple coverage criteria are selected. In this work, we are considering three important coverage criteria i.e. statement, MC/DC and branch coverage (must be specified by the tester) to compare, cluster, minimize and prioritize the test cases rather than using single criteria only. The user can choose other

combinations also depending upon their requirements. Selection of coverage criteria depends on the behavior of their ability to capture a test case execution and the behavior of the software program to be tested. The proposed SB_TSO (Similarity Based Test Suite Optimization) is described in Algorithm 2.

The stepwise procedure of our approach is as follows:

Step 1: Program is instrumented according to the three selected coverage criteria i.e. statement, MC/DC and branch coverage.

Step 2: Collect execution profile by executing the test cases over the instrumented program.

Step 3: For every test cases, the weight is calculated according to their total coverage information.

$$W_{Cov}T_i = \frac{TN_{Cov}}{Max_{Cov}} \times 10 \tag{5.2}$$

Where, TN_{Cov} – Total number of the selected coverage criteria (i.e. statement or MC/DC or branch) covered by the test case T_i and Max_{Cov} - Maximum number of selected criteria covered by any test case T_i.

Step 4: Calculate the overall weight for each test case.

$$TW_{oc}(T_i) = \frac{\sum_{k=1}^{m} WC_k T_i}{m} \tag{5.3}$$

Where, C_k represents the selected coverage criteria and m is the total number of criteria.

Step 5: Calculate the similarity level of each test case pair according to every individual coverage criteria.

$$Similarity\ Degree\ (T_i, T_j) = \frac{TN_{CoCov}(T_i, T_j)}{Total_{Cov}(T_i, T_j)} \tag{5.4}$$

Where, TN_{CoCov} – Total number of common coverage criteria covered by test case pair. $Total_{Cov}$ - represents the total number of coverage criteria covered by both the test cases.

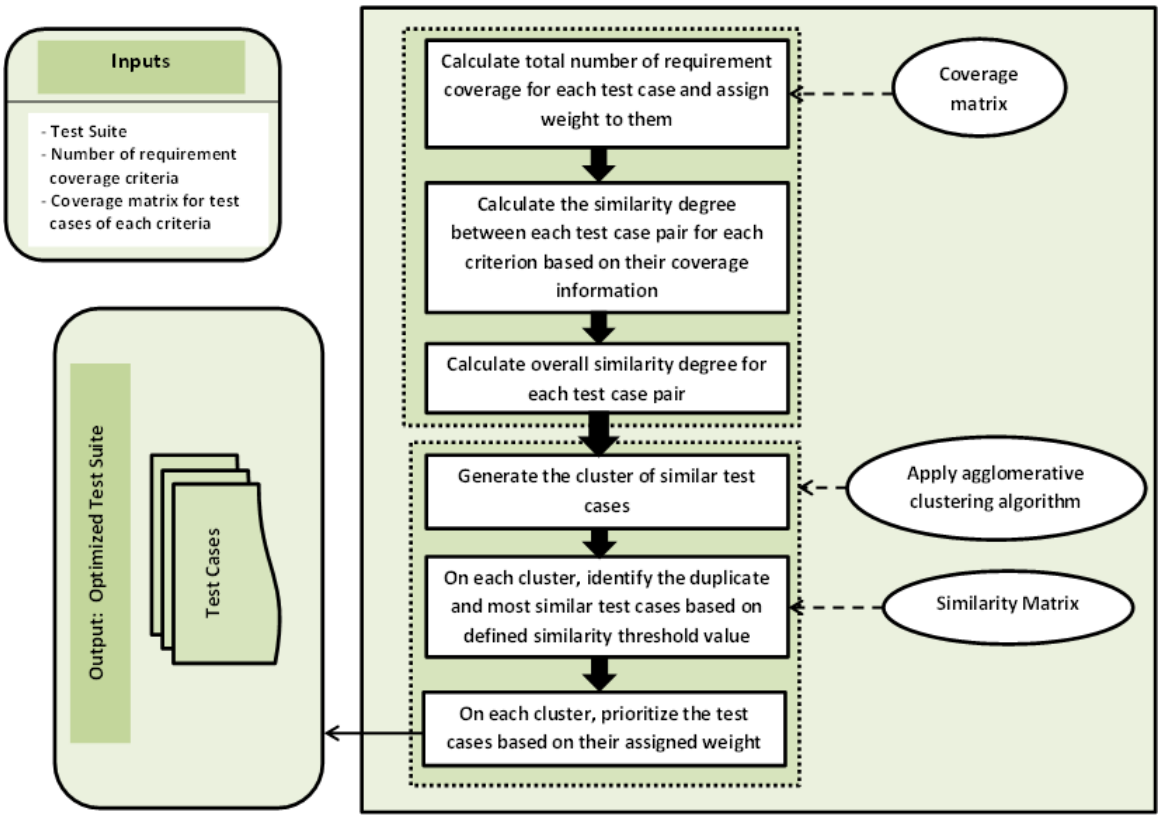


Figure 5.2 Similarity-based test suite optimization process framework

Step 6: Calculate the overall similarity level of test case pair.

$$SD_{Overall}(T_i, T_j) = \frac{\sum_{k=1}^{m} SD_k(T_i, T_j)}{m} \tag{5.5}$$

Where, SD_k - represents the similarity degree of test case pair associated with some specific coverage criteria. M is the total number of coverage criteria.

Step 7: For clustering, calculate the diversity values for each test case pair.

$$Diversity\ (T_i, T_j) = 1 - SD_{Overall}(T_i, T_j) \tag{5.6}$$

Step 8: Create the group of similar test cases with the help of agglomerative hierarchical clustering method.

Step 9: Within each cluster, duplicate and similar test cases are identified and removed from the cluster with the help of SelectOptimal( ) function.

Step 10: Prioritize the test cases after minimization with the help of their calculated overall weight.

5.3.1 Algorithm of SB_TSO

Algorithm 2: SBTSO (Similarity Based Test Suite Optimization)

Input: Source Code

Set of all test cases presented in the test suite TS ={ $\boldsymbol{T_1, T_2, T_3, \ldots, T_n}$}

Set of requirement coverage information for each test cases in TS for coverage criteria:

$\boldsymbol{Cov_k}$ ={$\boldsymbol{Cov_1, Cov_2, Cov_3, \ldots, Cov_k}$ }, where (k $\geq$ 2)

Output: Optimized test suite that satisfies all coverage requirements for k criteria

begin SBTSO

1. **for** each coverage criteria $\boldsymbol{Cov_k}$ **do**
2. **for** each test cases $\boldsymbol{T_i}$ **do**
3. Extract the selected coverage requirements from the source code
4. **if** (k= =1) **then** $\boldsymbol{R_{st} \leftarrow \{ S_1, S_2, S_3, \ldots, S_n\}}$;
5. **else if** (k= =2) **then** $\boldsymbol{R_{br} \leftarrow \{ B_1, B_2, B_3, \ldots, B_n\}}$;
6. **else if** (k= =3) **then** $\boldsymbol{R_{m/d} \leftarrow \{ P_1, P_2, P_3, \ldots, P_n\}}$;
7. **endif**
8. Generate Coverage Matrix and corresponding covered requirements are marked as visited (×);
9. Calculate total coverage for each test case as $\boldsymbol{TN_{k,i}}$ and $\boldsymbol{Max(TN_{k,i})}$, where (k, i) represents the i^{th} test the case for k^{th} criteria;
10. Compute weight for each test cases using Eq. 2;
11. **endfor**
12. **endfor**
13. **for** each test cases $\boldsymbol{T_i}$ **do**
14. Compute overall weight for each test cases using Eq. 3;
15. **endfor**

16. **for** each criterion $\boldsymbol{Cov_k}$ **do**
17. **for** $\boldsymbol{i \leftarrow 1\ to\ n-1}$ **do**
18. **for** $\boldsymbol{j \leftarrow 1\ to\ n}$ **do**
19. Find out the common requirements ($\boldsymbol{TN_{CoCov(k)}(T_i, T_j)}$) covered by test case pair and the total requirements ($\boldsymbol{Total_{Cov(k)}(T_i, T_j)}$) covered by them;
20. Generate Similarity Matrix [$\boldsymbol{T_i, T_j}$] by calculating the distance between test case pair using Eq. 4;
21. **endfor**
22. **endfor**
23. **endfor**
24. **for** $\boldsymbol{i \leftarrow 1\ to\ n-1}$ **do**
25. **for** $\boldsymbol{j \leftarrow 1\ to\ n}$ **do**
26. Generate overall similarity matrix for test case pair by calculating an overall similarity degree between them using Eq. 5;
27. Calculate diversity values for each test case pair for further clustering by using Eq. 6;
28. **endfor**
29. **endfor**
30. while (stopping criteria not reached) do
31. Apply Agglomerative Clustering Algorithm (Algorithm 1)
32. Apply function SelectOptimal (Clusters)
33. On each minimized cluster prioritize the test cases based on their calculated weight: $\boldsymbol{TW_{oc}(T_i)}$;
34. **end** SBTSO

function SelectOptimal (clusters)

begin

1. **if the** number of test cases in the cluster is > 1 **do**
2. **for** each pair of test cases in a cluster having diversity, values are $\leq$ *selected thresold value* **do**
3. choose the pair with diversity value of zero that represents the redundant pair

4. remove one of the test case from redundant cases randomly;
5. **for** each remaining test case pair in a cluster **do**
6. select the pair with minimal diversity value among others;
7. select the test case from that pair which are also paired withisother test case;
8. keep that test case with their new pair and remove the remaining one;
9. start with new pair and repeat a the step 7 until there are no pairs are left to be analyzed;
10. get the essential test cases on that cluster;
11. put that cluster with essential test cases into a set called *minimized cluster*;
12. **else** put that cluster into *minimized cluster*;
13. **endif**
14. Generate the representative test suite by merging all the clusters of *minimized clusters;*
15. **end** SelectOptimal

5.4 Experimental Results and Discussion

The experiments were conducted using ten different subject programs covering a wide range of applications. The performance of the proposed SB_TSO algorithm is evaluated against the coverage based state-of-the-art algorithms: HGS, GRE (Coverage–based).

5.4.1 Test Artifacts

The subject programs considered for this work have been well structured and without any compilation errors. The program description is shown in Table 5.1. The selection of test cases was done using control and data flow criterion. Each program considered for experimentation used three coverage information i.e. statement, branch, and MC/DC coverage that was hand instrumented. All the test suite reduction approaches considered in this work had been implemented using the C++ programming language.

Table 5.1 Subject program description

Program No.	Subject Programs	Program Description
Pr. 1	Triangle	Return the type of a triangle by three integers
Pr. 2	Pushdown	It pushes the element down through its descendants by a sequence of swaps to its proper position
Pr. 3	Prime number	To determine whether a number is prime or not
Pr. 4	Leap Year	To determine whether the year is a leap year or not
Pr. 5	Greatest Number	To find the largest number among three numbers
Pr. 6	Number of digits	To calculate the number of the digits of a given number
Pr. 7	Odd/even	To determine whether a number is odd/even
Pr. 8	Quadratic equation	Find all roots of a Quadratic equation
Pr. 9	Calc	Simple calculator to add, subtract, multiply and divide
Pr. 10	Bubble sort	Sort by repeatedly stepping through the list

5.4.2 Test Measures

Software metrics play an important part in measuring attributes that are very essential for software success. Measuring these attributes helps in better understanding of the features of the attributes. Measurement of software testing process features enables you to gain better insight into the software testing process. Generally, the test metric can be broadly classified to assess the test activity as:

- Independent test variables
- Dependent test variables

This experiment employed two independent variables which are the ten subject program and the test suites generated. Whereas, two dependent variables were also measured during experimentation include test suite size and fault detection loss percentage.

In this chapter, the following measures were used to assess the performance of proposed algorithms and state-of-the-art algorithms.

a) Suite Size Reduction (SSR) Measurement

Suite size reduction (SSR) percentage implies the number of test cases removed from the original suite.

$$SSR = \frac{|TS| - |MS|}{|TS|} \times 100\ \% \tag{5.7}$$

Where, $|TS|$ represent the number of test cases in the original test suite and $|MS|$ is the number of test cases in the minimized test suite.

b) Fault Detection Loss (FDL) Measurement

Fault detection loss (FDL) percentage signifies the total number of faults revealed by the minimized test suite.

$$FDL = \frac{|F| - |F_m|}{|F|} \times 100\ \% \tag{5.8}$$

5.4.3 Case Study

This section presents the experimentation and evaluation of the proposed optimization approach with the help of a sample case study, observing optimized test suite size and fault coverage. The experimentation compares the relative performance and effectiveness of the proposed SB_TSO algorithm with the state-of-the-art HGS and GRE algorithms. Table 5.2 presents a standard pushdown procedure, a case study on which the proposed approach has been performed to evaluate the performance. Figure 5.3 shows the control flow graph of the pushdown module. The module pushdown accepts an array $A[1], A[2], \ldots, A[n]$ and two integers 'first' and 'last' representing the first and last items of the array 'A'. By a sequence of swapping process, it pushes the element A[first] down through its descendants to its appropriate position in the tree until 'A' fulfils the property of partially ordered tree i.e. if $A[i].key \leq A[2 \times i].key\ and\ A[i].key \leq A[2 \times i + 1].key$ for $1 \leq i \leq \lfloor n/2 \rfloor$.

Table 5.2 Pushdown procedure and the injected faults

St. No.	Original	Fault ID	Injected fault(s)
	procedure *pushdown* (*first*, *last*: integer);		
	var: *r*: integer;		
	begin		
S1	*r* = *first*;		**None**
S2	**while** *r* < = *last* **div** 2 **do**	**F1**	**while** *r* >= *last* **div** 2 **do**
S3	**if** *last* = 2**r* **then begin**	**F2**	**if** *last* = 2**r+1* **then begin**
S4	**if** *A*[*r*].*key* > *A*[2**r*].*key* **then**	**F3**	**if** *A*[*r*].*key* > *A*[2+*r*].*key* **then**
S5	*swap*(*A*[*r*], *A*[2**r*]);		**None**
S6	*r* = *last*		**None**
	end		**None**
	else { *r* has two children, elements at 2**r* and 2**r* + 1 }		**None**
S7	**if** *A*[*r*].*key* > *A*[2**r*].*key* **and** *A*[2**r*].*key* < = *A*[2**r* + 1].*key* **then**	**F4**	**if** *A*[*r*].*key* < *A*[2**r*].*key* **and** *A*[2**r*].*key* > = *A*[2**r* + 1].*key* **then**
S8	**begin** *swap*(*A*[*r*], *A*[2**r*]);		**None**
S9	*r* = 2**r*	**F5**	*r* = 2**r+1*
	end		**None**
S10	**else if** *A*[*r*].*key* > *A*[2**r* + 1].*key* **and** *A*[2**r* + 1].*key* < *A*[2**r*].*key* **then**	**F6**	**else if** *A*[*r*].*key* > *A*[2**r*].*key* **and** *A*[2**r*].*key* < *A*[2**r+1*].*key* **then**
S11	**begin** *swap*(*A*[*r*], *A*[2**r* + 1]);		**None**
S12	*r* = 2**r* + 1	**F7**	*r* = 2**r*
	end		**None**
	else { *r* does not violate partially ordered tree property }		**None**
S13	*r* = *last*		**None**
	end *pushdown*		**None**

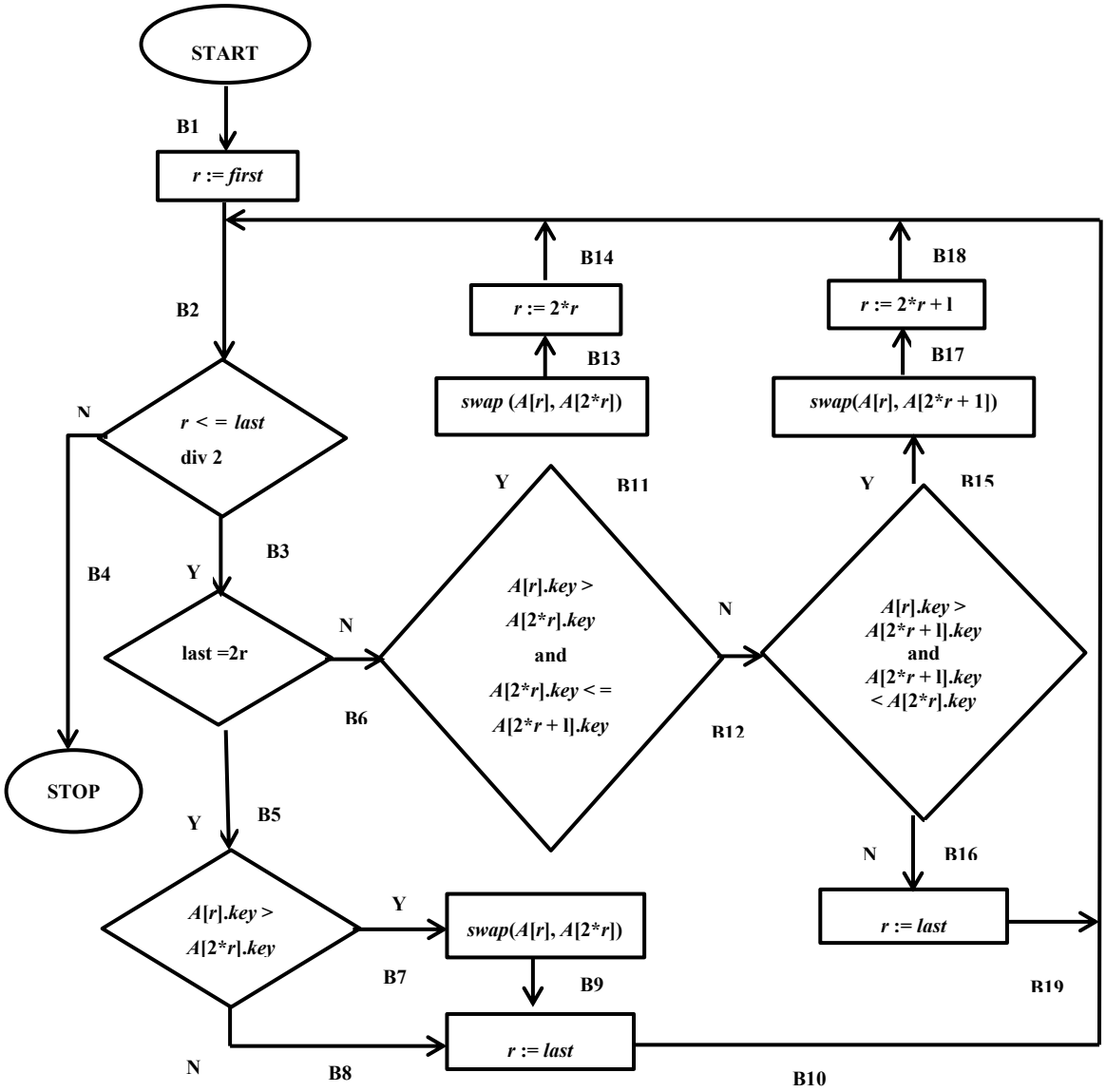


Figure 5.3 Control-flow graph of the pushdown procedure

Table 5.3 Test Cases for the pushdown procedure

Test cases	A[i].key	First	Last
T1	2 1	1	2
T2	2 1 3 4	1	4
T3	5 1 3 4 2	1	5
T4	3 2 1 4 5 6 7	1	7
T5	1 2	1	2
T6	1 2 3	1	3
T7	1 4 3 2	2	4

T8	2 1 3 4 5	1	5
T9	3 2 1 4 5 6	1	6
T10	4 1 3 2	1	4
T11	1 2 3 4	2	4
T12	6 2 1 4 5 3	1	6

Table 5.4 Fault Matrix

Faults / Test Cases	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**
T1		×	×				
T2	×	×		×			
T3	×	×		×	×	×	
T4	×					×	
T5		×	×				
T6							
T7		×	×				
T8	×			×			
T9	×					×	
T10	×	×		×	×		
T11		×	×				
T12	×	×				×	×
***The* × designate fault is revealed and a blank cell indicates fault is not revealed**							

The proposed approach also uses hand-seeded faults [36] for the subject program that are shown in Table 5.2. We have injected different types of errors in the faulty version of the subject program i.e. categorized as: (1) changing the operator in an expression,

(2) changing the value of a constant, (3) modifying code, (4) and changing the logical behavior of the code in which all the faults are treated as equally severe. Faults are represented as $F_1, F_2, F_3, \ldots, F_n$ and where 'n' signifies the number of injected faults. Table 5.3 listed all the test cases of the pushdown procedure and Table 5.4 shows the fault matrix to represent the seeded faults. In the fault matrix, each column represents an injected fault, F_i and each row signified a test case T_j . If a cell in the matrix is × then it represents a fault F_i corresponding to a test case T_j and blank cell otherwise.

Empirical studies have shown that the structural testing based on either control flow or data flow coverage criteria can significantly improve the fault detection rate rather than random testing [66]. Hence the proposed approach uses the statement, branch and MC/DC criteria to evaluate the test case adequacy and accesses the fault detection capability and requirement coverage by using it. In order to collect the coverage information of test cases for each selected criterion, the subject program and the source code were instrumented. The total number of statements are thirteen which are annotated as $\{S_1, S_2, S_3, \ldots, S_n\}$ (where, n=13) in the procedure (Table 5.2). The statement coverage information for each test case is shown in Table 5.5.

Table 5.5 Statement coverage information of test cases

Test Cases	Statement Coverage	Total Coverage
T1	S1, S2, S3, S4, S5, S6	6
T2	S1, S2, S3, S4, S6, S7, S8, S9	8
T3	S1, S2, S3, S7, S8, S9, S10, S11, S12	9
T4	S1, S2, S3, S7, S10, S11, S12, S13	8
T5	S1, S2, S3, S4, S6	5
T6	S1, S2, S3, S7, S10, S13	6
T7	S1, S2, S3, S4, S5, S6	6
T8	S1, S2, S3, S7, S8, S9, S10, S13	8
T9	S1, S2, S3, S4, S6, S7, S10, S11, S12	9
T10	S1, S2, S3, S4, S5, S6, S7, S8, S9	9

T11	S1, S2, S3, S4, S6	5
T12	S1, S2, S3, S4, S5, S6, S7, S10, S11, S12	7

Each row represents the test cases and the column or field 'Statement coverage' signifies the statements covered by them. Total coverage field represents the total number of statements covered by each test case. The procedure contains five decision statements $S_{2,}$ S_3, S_4, S_7 and S_{10}, which are shown in Table 5.6 with their MC/DC pairs and truth vectors. For MC/DC coverage, execution of each condition in a decision independently affects the result of the decision [31]. For example, the decision statement, S_{10}, that has two conditions: F, and G. Each possible evaluation of those decision statements generates a truth vector i.e. a vector of the Boolean values comes out after the evaluation of the conditions in those decisions. For example, "TF" in Table 5.6 for statement S7 is a truth vector in which condition D evaluates to true, and E evaluated to false. MC/DC pair coverage detail of every test cases and their calculated weight are shown in Table 5.7. For each condition of the decision statements, the truth and false coverage by test cases are shown in Table 5.7. The covered MC/DC pair is marked as × and blank cell otherwise.

Table 5.6 MC/DC Pairs for decision statements and their truth vectors

S2: **while** *r* < = *last* **div** 2 (A)		S3: **: if** *last* = 2**r* *(B)*		S4: **if** *A*[*r*].*key* > *A*[2**r*].*key (C)*	
A(T)	A(F)	B(T)	B(F)	C(T)	C(F)
T	F	T	F	T	F

S7: **if** *A*[*r*].*key* > *A*[2**r*].*key* **and** *A*[2**r*].*key* < = *A*[2**r* + l].*key* *(D AND E)*				S10: **if** *A*[*r*].*key* > *A*[2**r* +l].*key* **and** *A*[2**r* + l].*key* < *A*[2**r*].*key* *(F AND G)*			
D(T)	D(F)	E(T)	E(F)	F(T)	F(F)	G(T)	G(F)
TT	FT	TT	TF	TT	FT	TT	TF

Table 5.7 MC/DC pair coverage matrix

Test Cases	A		B		C		D		E		F		G		Total Coverage	
	T	F	T	F	T	F	T	F	T	F	T	F	T	F	N(T)	N(F)
T1	X	X	X		X										3	1
T2	X	X	X	X		X	X		X						4	3
T3	X	X		X			X		X	X	X		X		5	3
T4	X	X		X				X		X	X		X		3	4
T5	X	X	X			X									2	2
T6	X	X		X				X							1	3
T7	X	X	X		X										3	1
T8	X	X		X			X	X	X						3	3
T9	X	X	X	X		X				X	X		X		4	4
T10	X	X	X	X	X		X		X						5	2
T11	X	X	X			X									2	2
T12	X	X	X	X	X					X	X		X		5	3

Table 5.8 Branch coverage and weight for each test case pair

Test Cases	Branches Covered by Test Cases																		
	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B 8	B 9	B 10	B 11	B 12	B 13	B 14	B 15	B 16	B 17	B 18	B 19
T1	X	X	X	X	X		X		X	X									
T2	X	X	X	X	X	X		X		X	X		X	X					

T3	X	X	X	X		X					X	X	X	X	X		X	X	
T4	X	X	X	X		X						X			X	X	X	X	X
T5	X	X	X	X	X			X		X									
T6	X	X	X	X		X						X				X			X
T7	X	X	X	X	X		X		X	X									
T8	X	X	X	X		X					X	X	X	X		X			X
T9	X	X	X	X	X	X		X		X		X			X		X	X	
T10	X	X	X	X	X	X	X		X	X	X		X	X					
T11	X	X	X	X	X			X		X									
T12	X	X	X	X	X	X	X		X	X		X			X		X	X	

Control flow graph of the pushdown procedure is represented by Figure 5.3, in which the branches are annotated as $B_1, B_2, B_3, \ldots, B_n$, (where n= 19). The collected branch coverage information for each test case is shown in Table 5.8 and the branches covered by each test case are marked with $\times$ in the corresponding columns.

5.4.4 Results and Discussion

The experimentation is carried out using the proposed and state-of-the-art algorithms focused on two different aspects to analyze the outcome of the test suite optimization:

- The size of the test suite
- The fault detection loss of representative test suite

We divided the test suite TS into three sub-suites TS-1, TS-2, and TS-3 for evaluating the proposed work. Where TS1 comprises of $\{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$, TS2 comprises of $\{T_1, T_2, T_3, T_4, T_8, T_9\}$, and TS3 encloses $\{T_1, T_3, T_4, T_5, T_6, T_8, T_{10}, T_{11}, T_{12}\}$. For all the three selected criteria i.e. statement, MC/DC, and branch coverage, the computed similarity values are presented in Table 5.9. For clustering, the calculated diversity value for each pair of test cases is shown in Table 5.10. These values disclose how much the test case pair is identical or different from each other. The smaller diversity value between two test cases shows that the test case pair is less diverse and maximum value represents that the pair are highly different. In which we can clearly see that the T1-T7 and T5-T11 pairs are duplicate, because their associated distance values are zero. The value zero demonstrates that the test case pair is redundant in terms of each criterion. After clustering, one of the duplicate test cases must be removed randomly to optimize the test suite.

Table 5.9 Criteria-wise similarity values for each test case pair

Test Cases		**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**T11**	**T12**
T1	**Stmt.**		0.55	0.25	0.27	0.83	0.33	1.00	0.27	0.50	0.67	0.83	0.60
	MC/DC		0.37	0.20	0.22	0.60	0.33	1.00	0.25	0.33	0.57	0.60	0.50
	Branch		0.46	0.25	0.26	0.66	0.33	1.00	0.26	0.42	0.66	0.66	0.61
T2	**Stmt.**			0.55	0.33	0.62	0.40	0.55	0.60	0.54	0.88	0.62	0.50
	MC/DC			0.50	0.27	0.57	0.37	0.37	0.62	0.50	0.75	0.57	0.36
	Branch			0.53	0.29	0.54	0.41	0.46	0.57	0.53	0.76	0.63	0.41
T3	**Stmt.**				0.70	0.27	0.50	0.25	0.70	0.63	0.50	0.27	0.58
	MC/DC				0.66	0.20	0.33	0.20	0.55	0.60	0.50	0.20	0.60
	Branch				0.64	0.26	0.42	0.25	0.64	0.60	0.50	0.26	0.56

T4	**Stmt.**					0.30	0.75	0.27	0.60	0.70	0.30	0.30	0.63
	MC/DC					0.22	0.57	0.22	0.44	0.66	0.27	0.22	0.66
	Branch					0.28	0.72	0.26	0.57	0.64	0.27	0.28	0.60
T5	**Stmt.**						0.37	0.83	0.30	0.55	0.55	1.00	0.50
	MC/DC						0.33	0.60	0.25	0.50	0.37	1.00	0.33
	Branch						0.36	0.66	0.28	0.58	0.46	1.00	0.42
T6	**Stmt.**							0.33	0.75	0.50	0.36	0.37	0.45
	MC/DC							0.33	0.66	0.33	0.37	0.33	0.33
	Branch							0.33	0.72	0.42	0.33	0.36	0.40
T7	**Stmt.**								0.27	0.50	0.66	0.83	0.60
	MC/DC								0.25	0.33	0.57	0.60	0.50
	Branch								0.26	0.42	0.66	0.66	0.61
T8	**Stmt.**									0.41	0.54	0.30	0.38
	MC/DC									0.27	0.62	0.25	0.27
	Branch									0.35	0.53	0.28	0.33
T9	**Stmt.**										0.50	0.55	0.90
	MC/DC										0.36	0.50	0.77
	Branch										0.41	0.63	0.78
T10	**Stmt.**											0.55	0.58
	MC/DC											0.37	0.50
	Branch											0.46	0.56
T11	**Stmt.**												0.40
	MC/DC												0.33
	Branch												0.42

Table 5.10 Overall diversity values for each test case pair

Test Cases	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**T11**	**T12**
T1	0.00	0.54	0.77	0.75	0.31	0.67	0.00	0.74	0.59	0.37	0.31	0.43
T2	0.54	0.00	0.49	0.71	0.43	0.61	0.54	0.41	0.48	0.21	0.40	0.58
T3	0.77	0.49	0.00	0.34	0.76	0.58	0.76	0.37	0.39	0.50	0.76	0.42
T4	0.75	0.71	0.34	0.00	0.74	0.32	0.75	0.47	0.34	0.72	0.74	0.27
T5	0.31	0.43	0.76	0.74	0.00	0.65	0.31	0.73	0.46	0.54	0.00	0.59

T6	0.67	0.61	0.58	0.32	0.65	0.00	0.67	0.29	0.59	0.65	0.65	0.61
T7	0.00	0.54	0.76	0.75	0.31	0.67	0.00	0.74	0.59	0.37	0.31	0.43
T8	0.74	0.41	0.37	0.47	0.73	0.29	0.74	0.00	0.66	0.44	0.73	0.68
T9	0.59	0.48	0.39	0.34	0.46	0.59	0.59	0.66	0.00	0.58	0.44	0.19
T10	0.37	0.21	0.50	0.72	0.54	0.65	0.37	0.44	0.58	0.00	0.54	0.46
T11	0.31	0.40	0.76	0.74	0.00	0.65	0.31	0.73	0.44	0.54	0.00	0.62
T12	0.43	0.58	0.42	0.27	0.59	0.61	0.43	0.68	0.19	0.46	0.62	0.00

For grouping, we have considered the agglomerative hierarchical clustering method that helps in extracting the cluster of similar test cases. To begin with, single test cases are placed in separate clusters and then the test case belonging to two clusters are selected as a pair and the difference between their overall coverage capabilities in terms of distance is calculated. Accordingly, the clusters are merged based on the minimum distance value. Practically, the dendrogram is generated for this using the MatLab Tool to represent the clustering process graphically. Figure 5.4, 5.5, and 5.6 displays a tree-like structure called dendrogram for test suite TS-1, TS-2, and TS-3 with considering multiple coverage criteria that comprise of test cases as leaf nodes and cluster at a higher level. Where X-axis represents the cluster distances and Y-axis displays the test cases. Cutting this dendrogram horizontally creates several clusters of test cases. It can be observed that there are many clusters are available at different levels depending upon their similarity degree. Fig. 5.7 (a) to 5.7 (c) shows the dendrograms containing clusters of test cases for TS-1 by considering every single criterion separately with their diversity values. The analysis illustrates that the clusters formed after using multiple criteria is more optimized rather than using any single criteria.

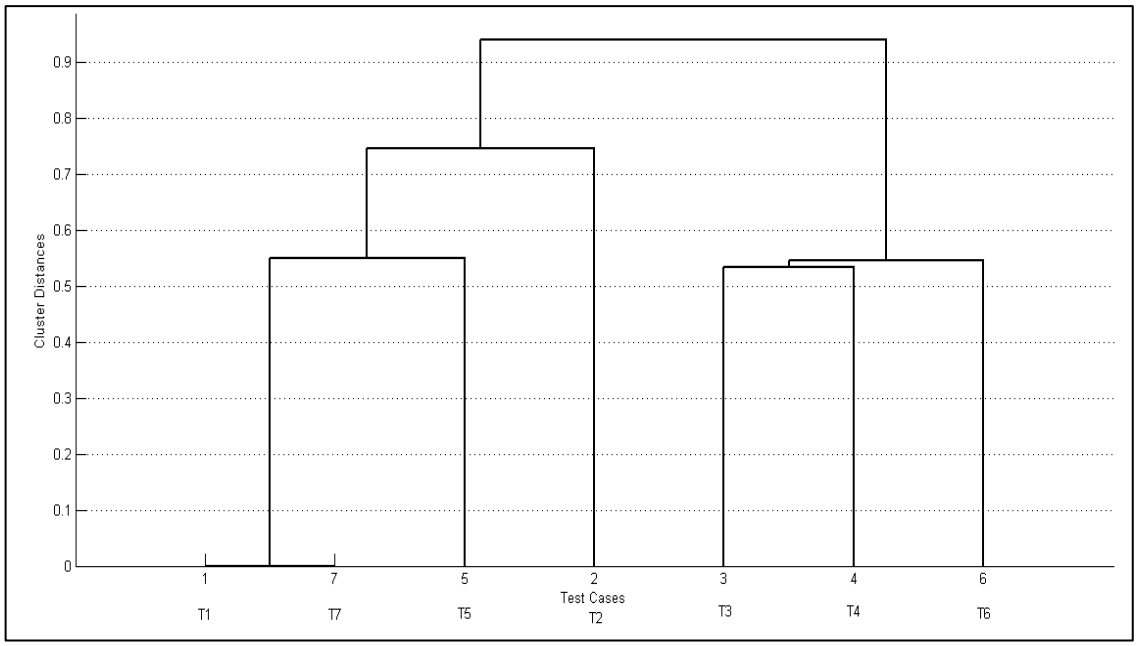


Figure 5.4 Dendrogram containing clusters of test cases for Test Suite-1 considering multiple coverage criteria

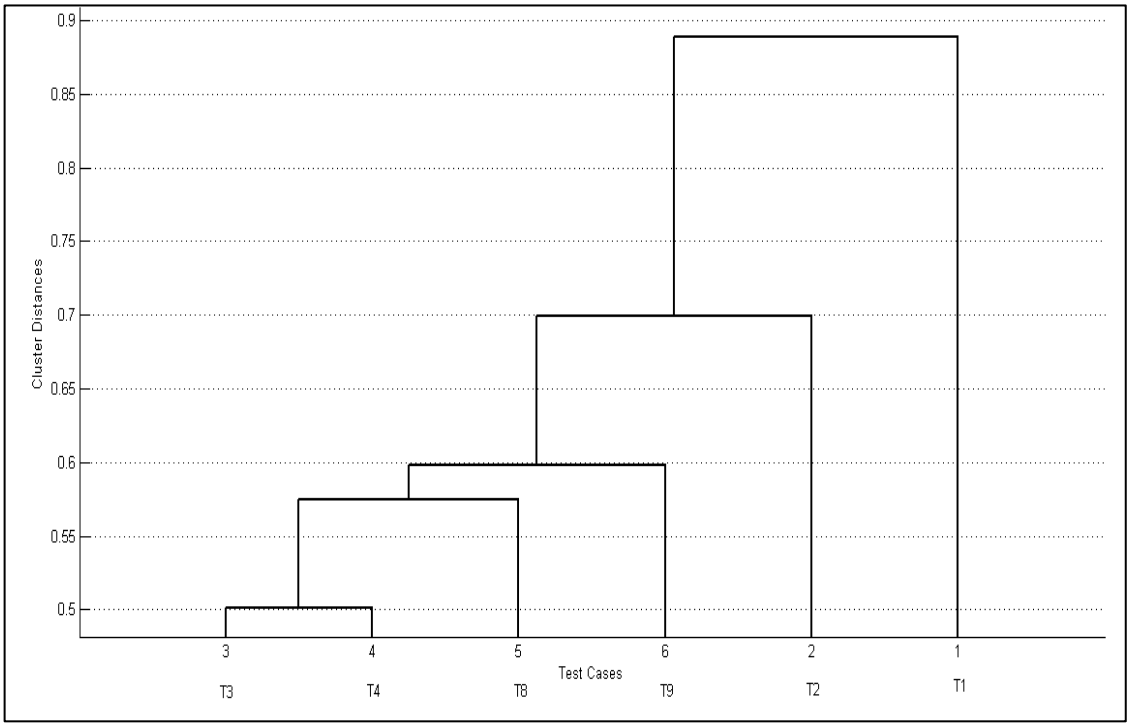


Figure 5.5 Dendrogram containing clusters of test cases for Test Suite-2 considering multiple coverage criteria

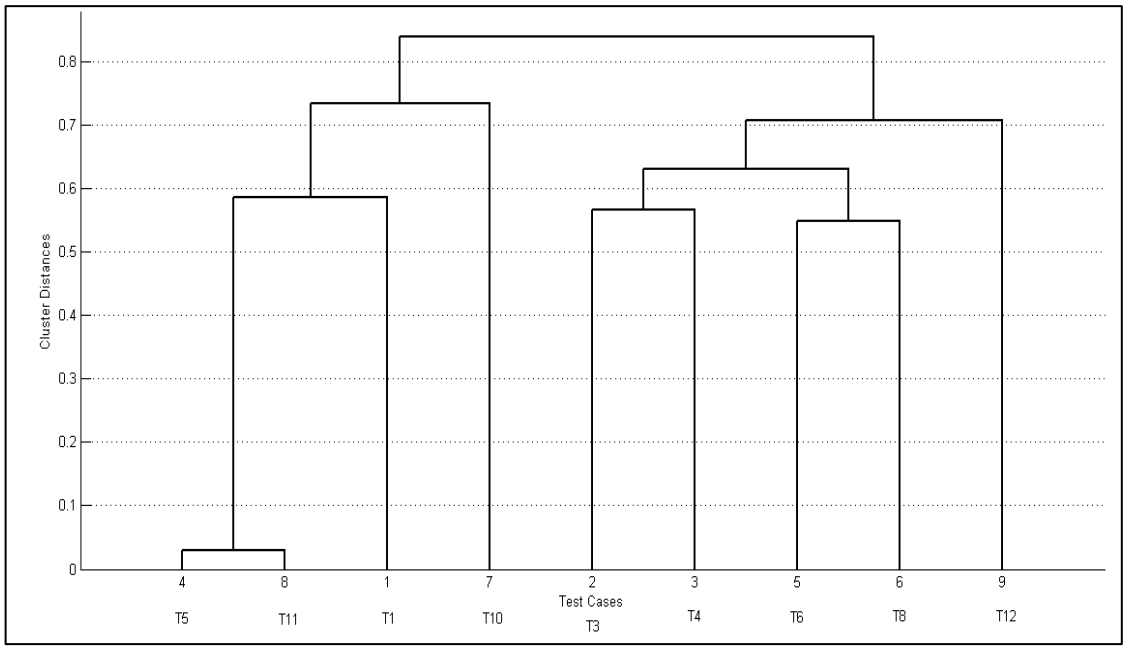


Figure 5.6 Dendrogram containing clusters of test cases for Test Suite-3 considering multiple coverage criteria

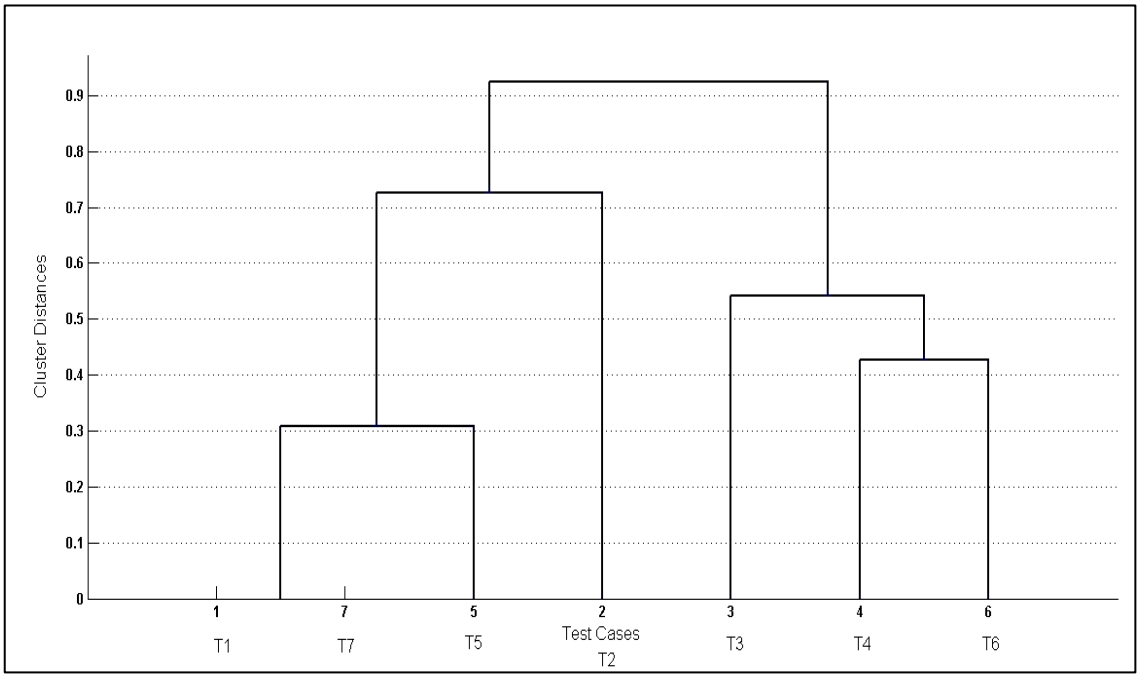


Figure 5.7 (a) Dendrogram containing clusters of test cases for Test Suite-1: Considering only statement coverage

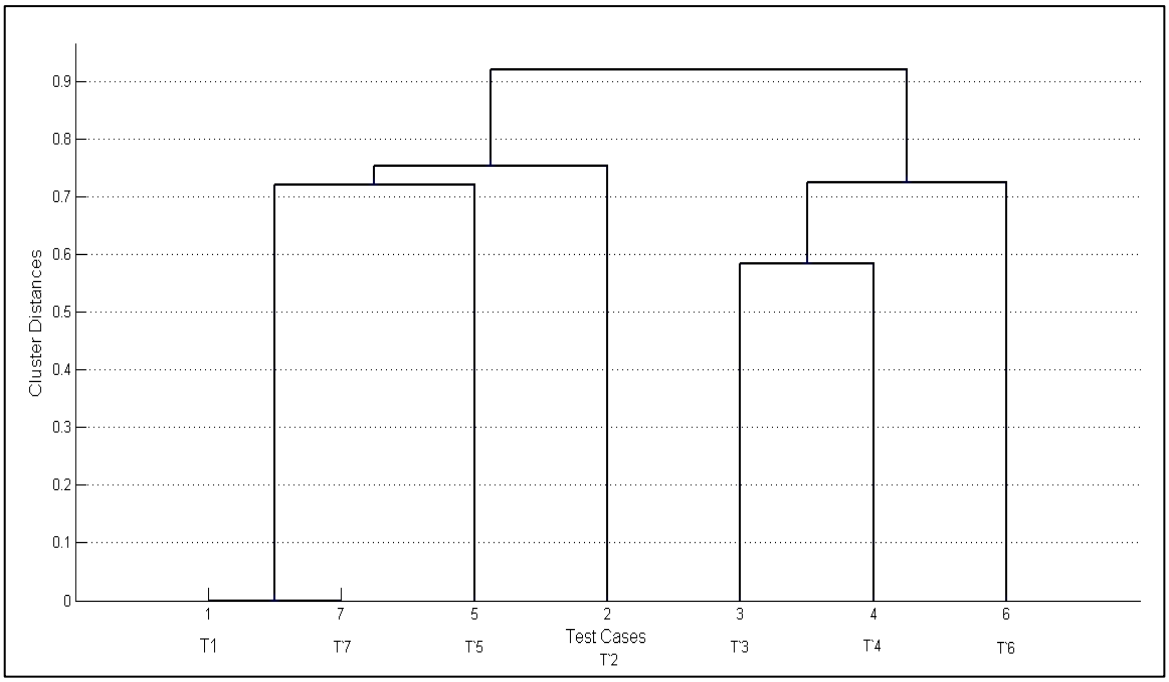


Figure 5.7 (b) Dendrogram containing clusters of test cases for Test Suite-1: Considering only branch coverage

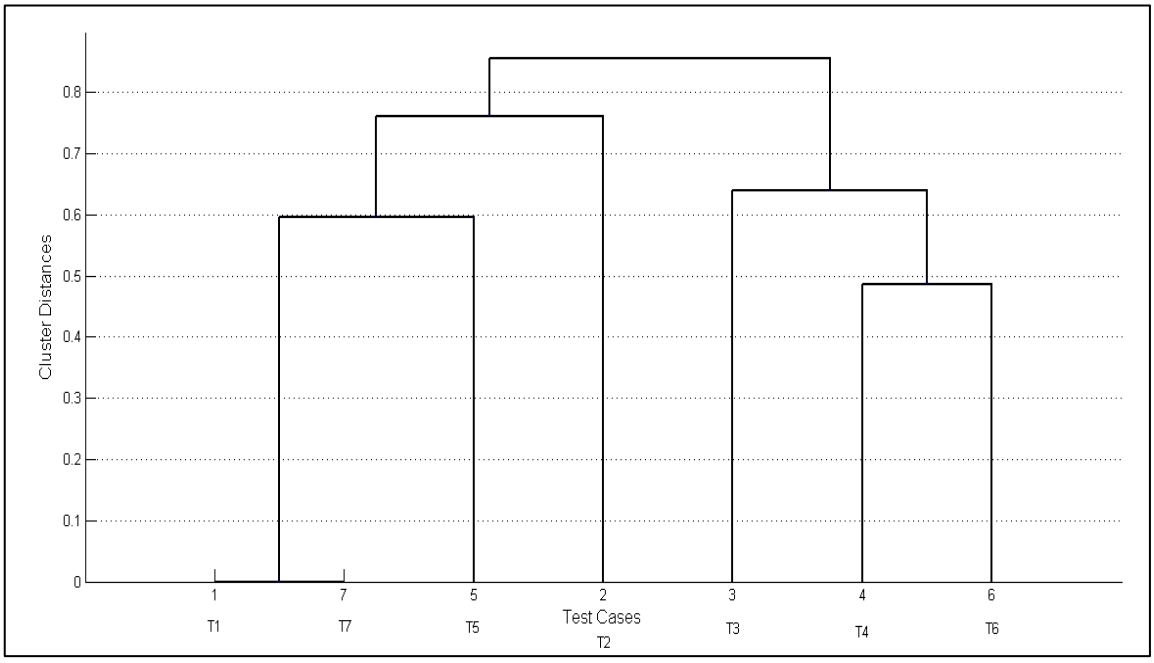


Figure 5.7 (c) Dendrogram containing clusters of test cases for Test Suite-1: Considering only MC/DC coverage

Table 5.11 List of clusters generated for the test suite of case study

Test Suite	Test Cases	Clusters
TS-1	T1, T2, T3, T4, T5, T6, T7	C1: {T1, T7, T5} C2: {T2} C3: {T3, T4, T6}
TS-2	T1, T2, T3, T4, T8, T9	C1: {T3, T4, T8, T9} C2: {2} C3: {1}
TS-3	T1, T3, T4, T5, T6, T8, T10, T11, T12	C1: {T1, T5, T11} C2: {T10} C3: {T3, T4, T6, T8} C4: {T12}

After implementing the cluster development process, different clusters covering similar test cases are generated. The desired similarity degree or the diversity value for test case pair is ≤0.35 for constructing a cluster. The diversity threshold value can be adjusted depends on the testers (users) requirement. Some of the test cases that are not included in the clustering process are treated as individual clusters. For the above experimental subject program, the clusters generated are presented in Table 5.11. On each cluster having more than one test case are available, we apply a further minimization process to get the representative optimal cluster with the help of SelectOptimal function. For each cluster, we must extract the similarity matrix from the Table 5.10. The test case pairs of each cluster are analyzed individually. Clusters of test suite TS-1 are C1, C2, and C3 respectively. In which, C2 represents the individual cluster because the test case T2 is not similar to any test cases presented in that cluster.

On the other hand, after analysis, we found that the test case pair T1-T7 are redundant due to their diversity value which is zero (0.00). So, one of the test cases of that pair has to be removed to optimize the test suite size. Basically, the SB_TSO algorithm analyzes

the diversity values of the similarity matrix from the lowest value and verifies whether the resultant test suite still keeps 100 % of the requirements coverage of each criterion after removal of the duplicate and similar test cases. Similarly, the other clusters of remaining test suites are analyzed and minimized accordingly.

The percentage of suite size reduction and the percentage of faults detection loss by the optimal test suite generated by the state-of-the-art algorithm and by the proposed technique is shown in Figure 5.8 to Figure 5.10. These outcomes reveal that the test suite minimization for the proposed algorithm is competitive with respect to the HGS and GRE algorithms. For state of the art algorithms, we have considered the branch coverage as a testing criterion.

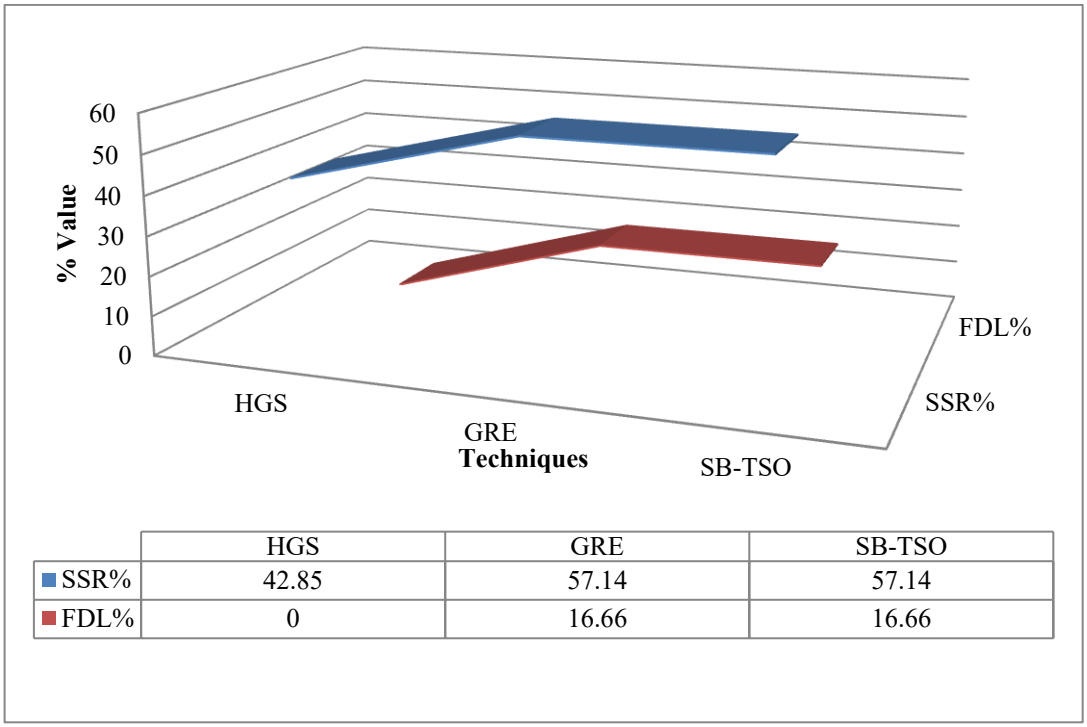
For each minimized test suite, the reduction in test suite size (SSR) is calculated using Eq. 5.7. The objective of regression testing is to satisfy the test requirements and also detect a maximum number of injected faults as soon as possible. With the help of fault matrix (see Table 5.4), the fault detection loss (FDL) in the representative test suite is evaluated using Eq. 5.8. As we can clearly see that the test case T12 of test suite TS-3 are only the test case that discovers the fault F7. However, the HGS and GRE did not include the test case T12 in the minimized test suite of TS-3, but the proposed SBTSO does.

And results inferred that requirement coverage directly affected the fault detection loss of the representative test suite. Finally, after reducing the test suite size, the test cases are ranked according to the calculated overall weight value (see Table 5.12). The ordering of test cases will significantly improve the fault detection rate and make the optimized test suite more effective. But, in this work the prioritization is optional and our main concentration is on test suite minimization.

Table 5.12 The overall calculated weight for test cases

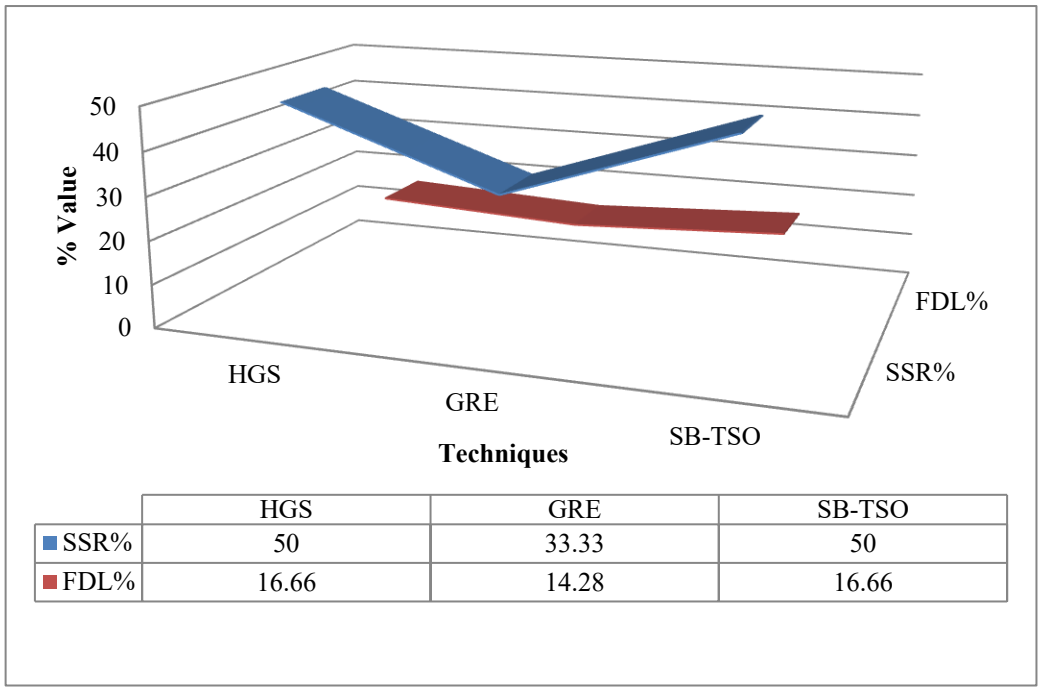
Test Cases	$W_{bc}(T_i)$	$W_{st}(T_i)$	$W_{mc/dc}(T_i)$	$TW_{oc}(T_i)$
T1	6.1	6	4.72	5.6
T2	8.4	8	8.61	8.33

T3	9.2	9	9.72	9.30
T4	8.4	8	8.88	8.42
T5	5.3	5	5.00	5.10
T6	6.1	6	5.27	5.79
T7	6.1	6	4.72	5.60
T8	8.4	8	7.5	7.96
T9	9.2	9	10	9.40
T10	9.2	9	8.33	8.84
T11	5.3	5	5.00	5.10
T12	10	7	9.72	8.90



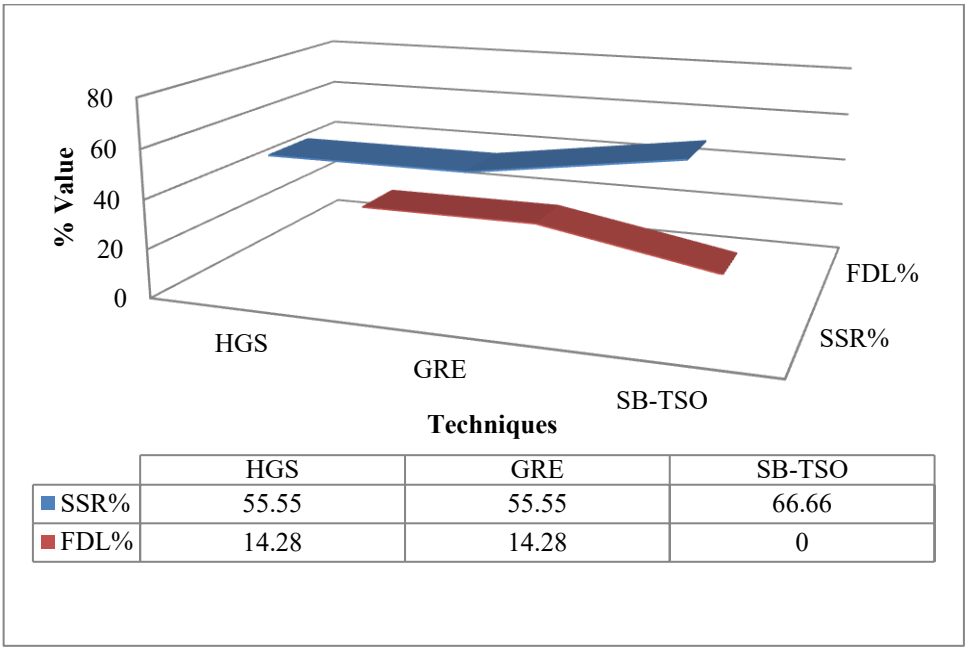
	HGS	GRE	SB-TSO
SSR%	42.85	57.14	57.14
FDL%	0	16.66	16.66

Figure 5.8 SSR and FDL percentage of TS-1 (SP-1) using SB_TSO and state-of-the-art algorithms



	HGS	GRE	SB-TSO
SSR%	50	33.33	50
FDL%	16.66	14.28	16.66

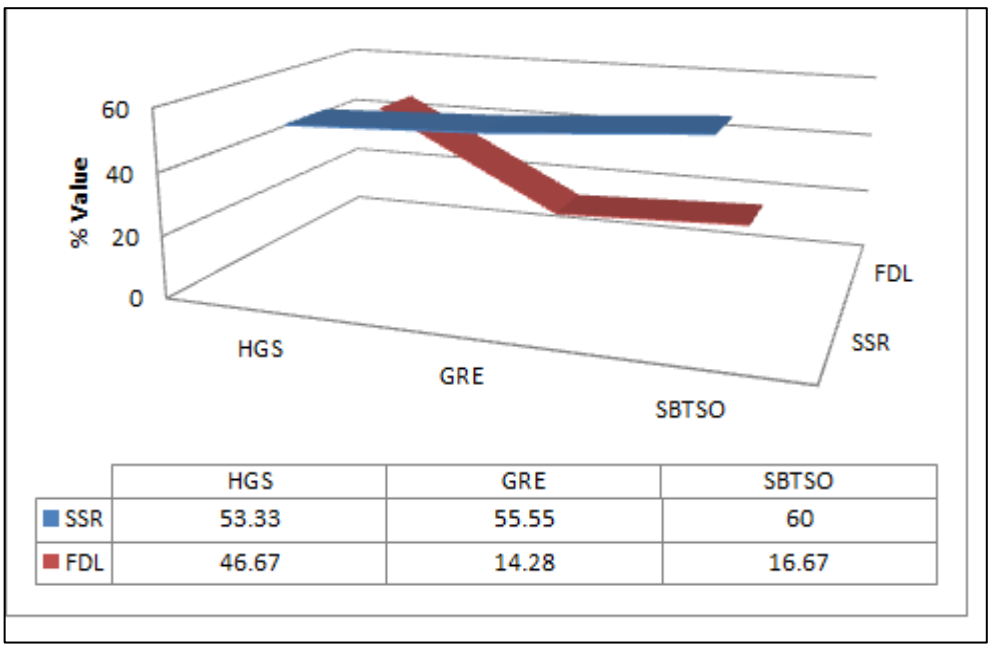
Figure 5.9 SSR and FDL percentage of TS-2 (SP-1) using SB_TSO and state-of-the-art algorithms



	HGS	GRE	SB-TSO
SSR%	55.55	55.55	66.66
FDL%	14.28	14.28	0

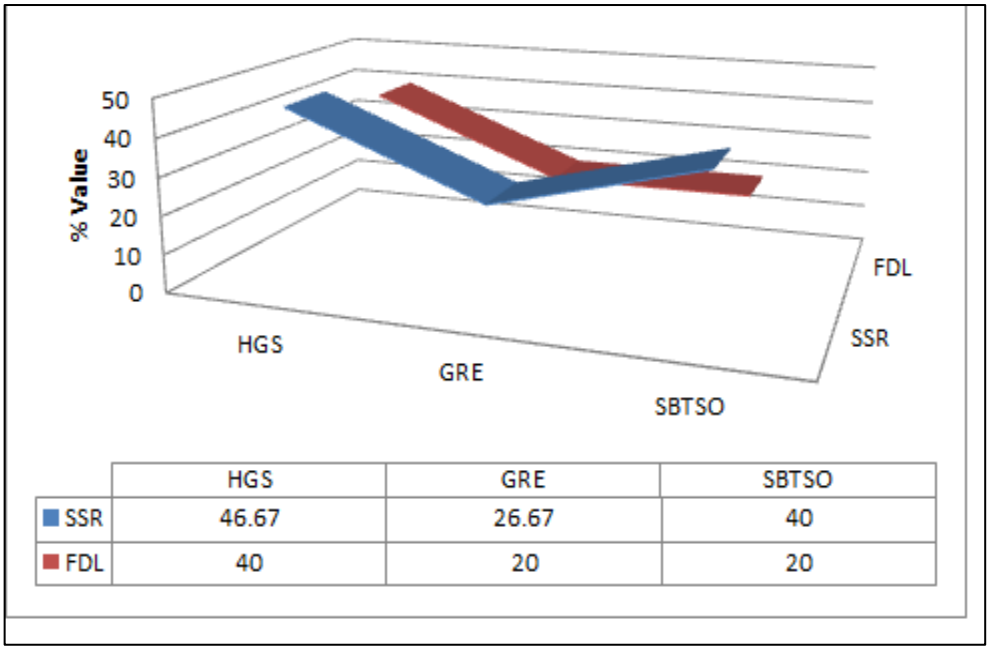
Figure 5.10 SSR and FDL percentage of TS-3 (SP-1) using SB_TSO and state-of-the-art algorithms

The proposed optimization approach is carried out systematically using the procedure described in this chapter for other subject programs (SP-2 to SP-10). The suite size reduction (SSR) and fault detection loss (FDL) for the SP-2 and SP-3 program is illustrated in Figure 5.11 and 5.12. The y-axis in the graph represents the SSR and FDL percentage against the techniques considered in the x-axis. And for the remaining subject programs, the observed values are shown in Figure 5.13. Where the y-axis in the graph represents the SSR and FDL percentage against the subject programs considered in the x-axis. From Figure 5.8 to Figure 5.13 it can be concluded that test suite size reduction and fault detection loss using SB_TSO is either similar or improved as compared to the state-of-the-art algorithms i.e. HGS and GRE. The result of comparative analysis also reveals that when the proposed algorithm is used the SSR and FDL for the subject programs ranges from 40 % to 52% and 0% to 18% respectively. The experimental results indicate that the FDL obtained by applying our proposed approach is in an average less than that one obtained by applying the state-of-the-art algorithms. Considering suite size reduction, it can be observed that in an average of the cases our strategy presents a similar behavior as compared to other heuristics.



	HGS	GRE	SBTSO
SSR	53.33	55.55	60
FDL	46.67	14.28	16.67

Figure 5.11 SSR and FDL percentage for SP-2 using SB_TSO and state-of-the-art algorithms



	HGS	GRE	SBTSO
SSR	46.67	26.67	40
FDL	40	20	20

Figure 5.12 SSR and FDL percentage FOR SP-3 using SB_TSO and state-of-the-art algorithms

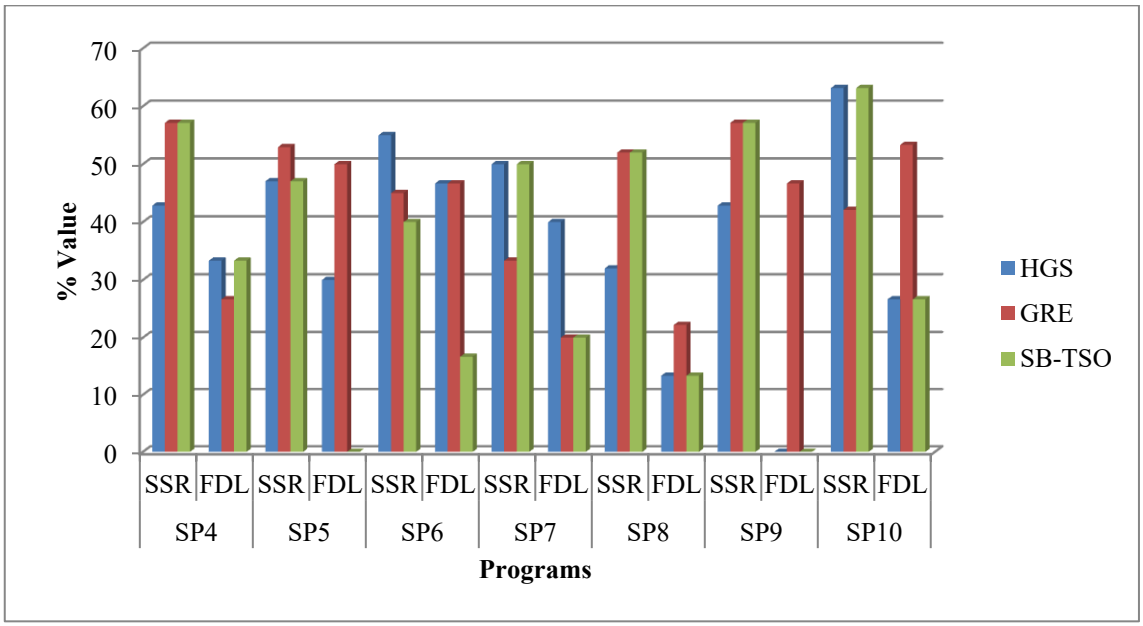


Figure 5.13 SSR and FDL percentage using SB_TSO and state-of-the-art algorithms for subject programs SP-4 to SP-10

The average, maximum and minimum test suite size reduction and fault detection loss attained for the proposed SB_TSO algorithm with respect to the HGS and GRE algorithms are presented in Table 5.13.

Table 5.13 states that when SB_TSO algorithm is used, the obtained average, maximum and minimum suite size reduction percentage is 51.64%, 63.15%, and 40% respectively. These values are marginally greater than HGS and GRE. So, overall the proposed approach is quite effective and competitive against HGS and GRE algorithm in suite size reduction.

The FDL values evaluated using SB_TSO and state-of-the-art algorithms are graphically represented in Figure 5.8 to Figure 5.13. The details shown in the graph are tabulated in three ranges: average, maximum and minimum FDL values. The table 5.13 also summarizes the fault detection loss by the representative test set after experimentation with the proposed algorithm and state-of-the-art algorithms. The observations made after experimentation show that SB_TSO algorithm provided an average of 17.66% fault detection loss against HGS (29.33%) and GRE (24.93%) algorithms. Further, the obtained results recommend that the proposed SBGA algorithm consistently outperformed the HGS and GRE algorithm in fault detection loss.

Table 5.13 Suite size reduction values

	HGS		**GRE**		**SB-TSO**	
	SSR%	FDL%	SSR%	FDL%	SSR%	FDL%
Average (%)	48.29	29.33	45.52	24.93	51.64	17.66
Maximum (%)	63.15	46.67	57.14	53.33	63.15	40
Minimum (%)	32	0	26.67	14.28	40	0

5.5 Statistical Validation

Statistical Hypothesis testing is conducted to assess whether or not the suite size reduction and fault detection loss are improved in the optimal test set as compared to the test suite generated by HGS and GRE. Since the sample size is small, the student T-test is applied to find out the level of significance and reject the null hypothesis. Meanwhile, the rejection or acceptance of a null hypothesis is based on either (0.05) alpha ($\propto$) or (0.01) alpha ($\propto$) level of significance for one tailed or two tailed test, (0.05) alpha ($\propto$) level of significance for a one-tailed test is taken for rejection of the null hypothesis.

The formulated hypothesis is mentioned below:

Null Hypothesis **(H_{01}):** Percentage of Suite Size Reduction (SSR %) cannot be improved using the proposed approach as compared to HGS.

Alternative Hypothesis **(H_{11}):** Percentage of Suite Size Reduction (SSR %) can be improved using the proposed approach as compared to HGS.

Null Hypothesis **(H_{02}):** Percentage of Suite Size Reduction (SSR %) cannot be improved using the proposed approach as compared to GRE.

Alternative Hypothesis **(H_{12}):** Percentage of Suite Size Reduction (SSR %) can be improved using the proposed approach as compared to GRE.

Null Hypothesis **(H_{03}):** Percentage of Fault Detection Loss (FDL %) cannot be improved using the proposed approach as compared to HGS.

Alternative Hypothesis **(H_{13}):** Percentage of Fault Detection Loss (FDL %) can be improved using the proposed approach as compared to HGS.

Null Hypothesis **(H_{04}):** Percentage of Fault Detection Loss (FDL %) cannot be improved using the proposed approach as compared to GRE.

Alternative Hypothesis **(H_{14}):** Percentage of Fault Detection Loss (FDL %) can be improved using the proposed approach as compared to GRE.

In general, the level of significance of proposed approach must be calculated to make it acceptable and for this purpose, t-test is found appropriate.

Level of Significance of SSR

To find out the significance of the difference between the HGS and GRE algorithm against the proposed SB_TSO approach in terms of SSR, the means of both old and new SSR are calculated as shown in Table 5.14 (a) and 5.14 (b). Pearson coefficient of correlation shows that the old SSR values before treatment and new values of SSR after treatment are highly correlated. The degree of freedom for both SSR values is 9.

The t value comes out to be 1.002 while considering data of HGS and the proposed approach. As the t value is less than the t critical value i.e. 2.262, the null hypothesis H_{01} is not strongly rejected and the alternate hypothesis H_{11} is rejected. But in the case of comparison with GRE, the t value comes out to be 1.959. As the t value is greater than the t critical value i.e. 1.882, the null hypothesis H_{02} is strongly rejected and the alternate hypothesis H_{12} is accepted.

Hence it is validated that SSR value is improved by applying the proposed optimization approach with multiple criteria as compared to GRE. And as compared to HGS the SSR is marginally less improved by the proposed approach. Overall our approach is quite competitive with the state-of-the-art algorithm in suite size reduction.

Table 5.14 (a) T-Test: Paired Two Sample for Means in terms of SSR (HGS vs. SB_TSO)

Statistical Observation	HGS	SB_TSO	
Mean	48.29	51.648	
Variance	69.53409	62.26946	
Observations	10	10	
Pearson Correlation	0.149108		
Hypothesized Mean Difference	0		
df	9		
t Stat	-1.00259		
P(T<=t) one-tail	0.171126		
t Critical one-tail	1.833113		
P(T<=t) two-tail	0.342252		
t Critical two-tail	2.262157		

Table 5.14 (b) T-Test: Paired Two Sample for Means in terms of SSR (GRE vs. SB_TSO)

Statistical Observation	GRE	SB_TSO
Mean	45.52	51.648
Variance	125.7976	62.26946
Observations	10	10
Pearson Correlation	0.509734	
Hypothesized Mean Difference	0	
df	9	
t Stat	-1.95914	
P(T<=t) one-tail	0.040879	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.081757	
t Critical two-tail	1.882157	

Level of Significance of FDL

To find out the significance of the difference between the HGS and GRE algorithm against the proposed SB_TSO approach in terms of FDL, the means of both old and new FDL are calculated as shown in Table 5.15 (a) and 5.15 (b). Pearson coefficient of correlation shows that the old FDL values before treatment and new values of FDL after treatment are highly correlated. The degree of freedom for both FDL values is 9.

In the case of statistical analysis with HGS and GRE both against the proposed technique, the t value comes out to be 2.299 and 2.277 respectively. As the values exceed the t critical value for a two-tailed test at the 0.05 level for 9 degrees of freedom, the null hypothesis H_{03} and H_{04} are strongly rejected and the alternate hypothesis H_{13} and H_{14} are accepted. Hence it is validated that the performance of the test suite optimization in terms of fault detection loss (FDL) can be improved using the proposed test suite optimization approach

Table 5.15 (a) T-Test: Paired Two Sample for Means in terms of FDL (HGS vs. SB_TSO)

Statistical Observation	HGS	SB-TSO
Mean	29.332	17.665
Variance	236.6005	155.6729
Observations	10	10
Pearson Correlation	0.351162	
Hypothesized Mean Difference	0	
df	9	
t Stat	2.299233	
P(T<=t) one-tail	0.023529	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.047058	
t Critical two-tail	2.262157	

Table 5.15 (b) T-Test: Paired Two Sample for Means in terms of FDL (GRE vs. SB_TSO)

Statistical Observation	GRE	SB-TSO
Mean	31.65	16.331
Variance	241.037	107.2575
Observations	10	10
Pearson Correlation	-0.32403	
Hypothesized Mean Difference	0	
df	9	
t Stat	2.27732	
P(T<=t) one-tail	0.024388	
t Critical one-tail	1.833113	
P(T<=t) two-tail	0.048776	
t Critical two-tail	2.262157	

5.6 Summary

In this chapter, we proposed a similarity-based test suite optimization approach based on the comparison of similarity between test cases in a given test suite. A single coverage criterion is not sufficient to select the diverse test cases, thus the key idea behind this approach is to use more than one coverage criteria i.e. statement, MC/DC and branch coverage to measure the similarity degree for each pair of test cases. So that the minimized test suites have the maximum diversity as well as better requirement and fault coverage. The proposed work also utilizes a clustering approach to speed up the minimization process. On the other hand, the test cases are also ranked based on their weight to maximize the fault detection effectiveness. The experimental results indicate that the minimized test suite sizes obtained by applying our proposed approach are in an average greater than that one obtained by applying the state-of-the-art algorithms. Considering the fault detection loss, it can be observed that in an average of the cases our strategy presents a similar behavior as compared to other heuristics.

The most important and essential part of the research work is to statistically validate the newly proposed approach. It is necessary to prove the approach to make it socially acceptable. In this chapter, the proposed work has been validated using the Student T-test to test the hypothesis. The observed values after t-test reveal that the proposed SB_TSO algorithm consistently outperformed the HGS and GRE algorithm in fault detection loss and behaves nearly similar in suite size reduction.

Chapter 6 Pair-Wise Selection Approach for Test Case Prioritization in Regression Testing

6.1 Introduction

Any software system with high quality cannot be achieved without a rigid development process [106, 107]. With the increase in the size and complexity of the recent software product, the importance and necessity of regression testing are increasing rapidly. Regression testing is a very complex activity in perspective of time and cost. More than 50% of the cost is used in the software maintenance phase [34]. With the continuous advancement in software systems, test suites often grow very large and it is impractical to re-execute all test cases within limited time and resources. Various optimization techniques have been proposed to solve the above problem, where, the test case prioritization is one of them that reorders test cases according to some selected specific criteria, such that the test cases with higher coverage are executed earlier [2, 70]. Test case prioritization techniques help to improve the effectiveness of some performance objectives. These techniques determine test cases in a particular execution sequence to increase efficiencies in achieving given objectives. After every modification in the software program, it is considered incompetent to re-run each test cases during regression testing. Having limited resource forces to select a highly effective prioritization technique, which schedules the order of cases; so that the most appropriate test case can be executed first that further enhances the effectiveness of regression testing. The aim of any prioritization technique is to increase the possibility of meeting certain criteria by the resultant test suite with earlier fault detection ability. Any test case prioritization technique effectiveness is measured using APFD (Average percentage of Faults Detected). Utilizing code coverage information as selection criteria, coverage-based test case prioritization reorders the test cases to maximize code

coverage as early as possible [2]. However, previous studies show that in some cases, code coverage as selection criteria is not sufficient to guarantee a high fault detection rate [29].

In recent years, similarity-based test case prioritization techniques, combining clustering approach [92], ART-approach [93] and other similarity-based prioritization [94], have gained more attention. Similarity-based test case prioritization techniques primarily concentrated on test case diversity that helps to detect more faults [72, 118]. But, most of the techniques use single criteria to measure the diversity between test case pairs that are not sufficient to get an optimal result. However, the optimal test suite is best generated by multiple coverage criteria [73, 119].

Generally, test cases are ordered to achieve some testing goals based on a certain criterion for effective regression testing [105]. Some of the testing goals used in prioritization are statement coverage, fault coverage, branch coverage, path coverage, MC/DC and def/use coverage. Additionally, certain other factors have been also used during prioritization of test cases such as execution time, implementation complexity etc. The empirical study reveals that; a number of researches have been carried out to prioritize the test cases using single coverage criteria. A test case which covers multiple criteria rather than covering single criteria is considered as a strong candidate for fault detection compared to the test case, which includes the single coverage criteria. Considering multiple coverage criteria to conduct regression testing is quite effective in terms of time, cost and effort required to make the regression process more effective.

In this chapter, several similarity-based prioritization techniques are proposed based on pair-wise selection strategy. Pairwise comparison of test cases is a fundamental strategy to inspect test cases and choose an association between them in a finite test set. The technique evaluates the similarity degree for each test case pair in three levels and accordingly assigns the execution order to the test cases in a test suite. We also empirically evaluate their improvement in fault detection rate that let developers initiate debugging and amending faults before the release of any software product. The results show that some of the proposed techniques can attain higher fault detection rate, in terms of APFD, in comparison with the other considered techniques and random ordering.

6.2 Background

a) Test Case Prioritization

The entire testing method consumes nearly half of the development cost. In the constraints of the given resources, the re-run of a set of test cases is unfeasible. It is difficult for the tester to make 100% fault-free software product in limited time [99]. Moreover, the tester is required to target creating such functionalities bug-free which are highly used and applicable in the organization [100, 101]. Therefore, the sample of test cases is must be chosen which may find the maximum number of faults within the given period of time [76, 102]. Such a selection method, which can be done to reduce test cases, but not always, can be effective in finding faults. So, there is a need to prioritize the test cases based on some criteria. Test case prioritization is one of all these approaches which rank the test cases so that test cases having the highest priority corresponding to some criteria are executed earlier [104]. Test case prioritization represents one of the regression testing techniques with the aim of improving the rate of fault detection of a test suite. The technique encourages software quality assurance by improving the chance of earlier execution of the essential test cases. That results in earlier detection of faults as well as earlier initiation of the debugging process. The test case prioritization problem is framed as follows [2]:

Given: a test suite TS; the set of permutations PT of TS; and, a function f from PT to the set of all real numbers.

Problem: To find $TS' \in PT$ such that, $\forall TS'' \in PT \Rightarrow f(TS') \geq f(TS'')$.

b) Coverage Similarity

The test case pair is said to be similar, if they are identical in terms of some selected criteria, where, the criteria may be coverage also. Coverage might be a branch, statement, def-use, control-flow, MC/DC, function, path, and data flow etc. We can apply more than one criterion to measure the similarity value, which represents how much the test case pair is identical to each other [120]. With the help of coverage similarity, any prioritization technique can improve the ordering of test cases which may also increase the fault detection rate.

6.3 Proposed Approach

In this section, we present our proposed test case prioritization techniques, mainly includes four steps:

- ***Instrumentation:*** Collect coverage information (statement, MC/DC, and branch coverage) for test suite by executing the test cases over software system.
- ***Distance calculation:*** Calculate the distance between test case pair for each criterion.
- ***Calculate integrated similarity values for each test case pair:*** integrate the similarity values of test case pairs for each criterion.
- ***Apply prioritization techniques and calculate APFD:*** Apply techniques to order the test cases and then evaluate fault detection effectiveness of prioritized test suite.

The block diagram of the prioritization process is illustrated in Figure 6.1, whereas the process flow diagram is shown in Figure 6.2 (a) and (b).

6.3.1 Test Case Similarity

The objective of the similarity based test suite prioritization strategy is to order the test cases based on the similarity values among them. The similarity degree between each pair of test cases is calculated by certain distance measure. The resultant ordered test suite must satisfy the coverage requirements as the original one. Hence, the goal is to achieve maximum requirement coverage and higher fault detection rate by the ordered test cases. Cartaxo et al. [31] define a distance measure that calculates the similarity degree between a pair of test cases stated as paths.

$$Similarity\ function[i,j] = \frac{nit}{Avg(|i|,|j|)} \tag{6.1}$$

Where nit represents the number of identical transition and ($|i|$, $|j|$) represents the average transition between path lengths.

With the help of the calculated similarity degree of each pair of test cases the $n \times n$ square similarity matrix is created, where n is the number of paths and each path represented as i $(1 \leq i \geq n)$ is called a test case.

Definition 1: Coverage - Set; a Coverage-Set of any test case T is a set of coverage triples $\{\langle s_1, m_1, b_1\rangle, \langle s_2, m_2, b_2\rangle, \ldots, \langle s_n, m_n, b_n\rangle\}$, where a triple $\langle s_i, m_i, b_i\rangle$ signifies a statement coverage specification s_i, modified decision/condition coverage (MC/DC) m_i, and a branch specification b_i. Let $(T) = \{s_1, s_2, \ldots, s_n\}$, $M(T) = \{m_1, m_2, \ldots, m_n\}$, and $B(T) = \{b_1, b_2, \ldots, b_n\}$, represent the set of statement coverage specifications, the set of MC/DC specification, and the set of branch specification, exercised in the execution of test case t to validate the software to be tested, respectively. Here coverage set may be of more than three coverage criteria which absolutely depend on user's choice. We further explain the coverage similarity in Definition **2.**

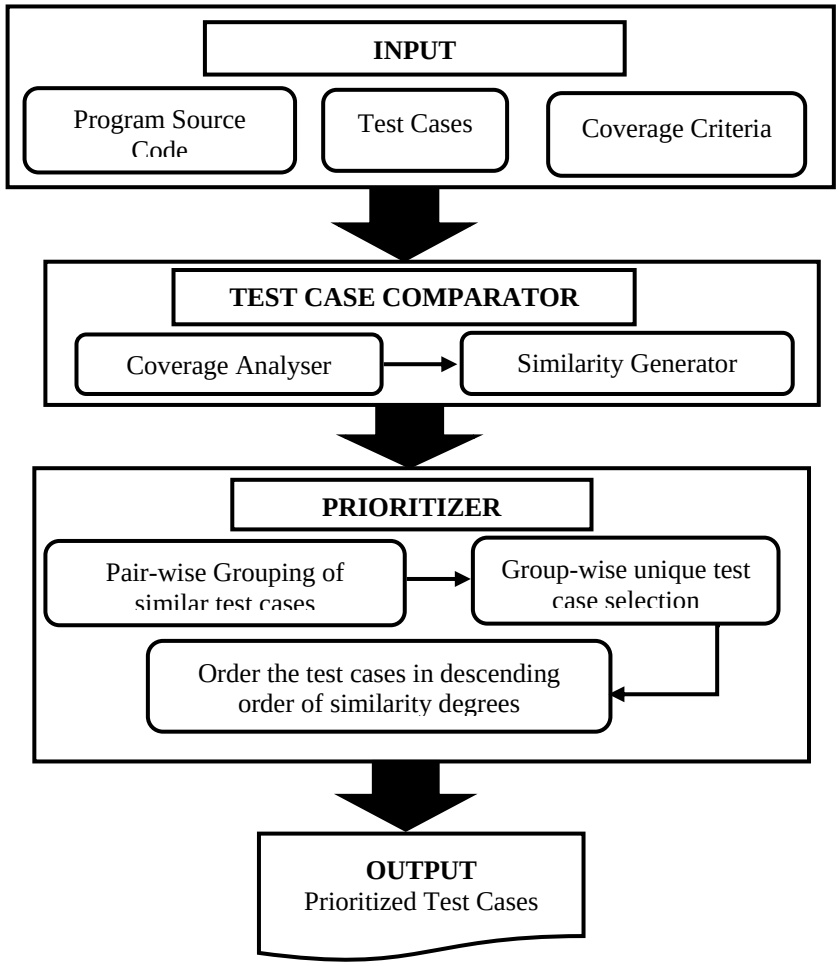


Figure 6.1 Block diagram for similarity based test case prioritization

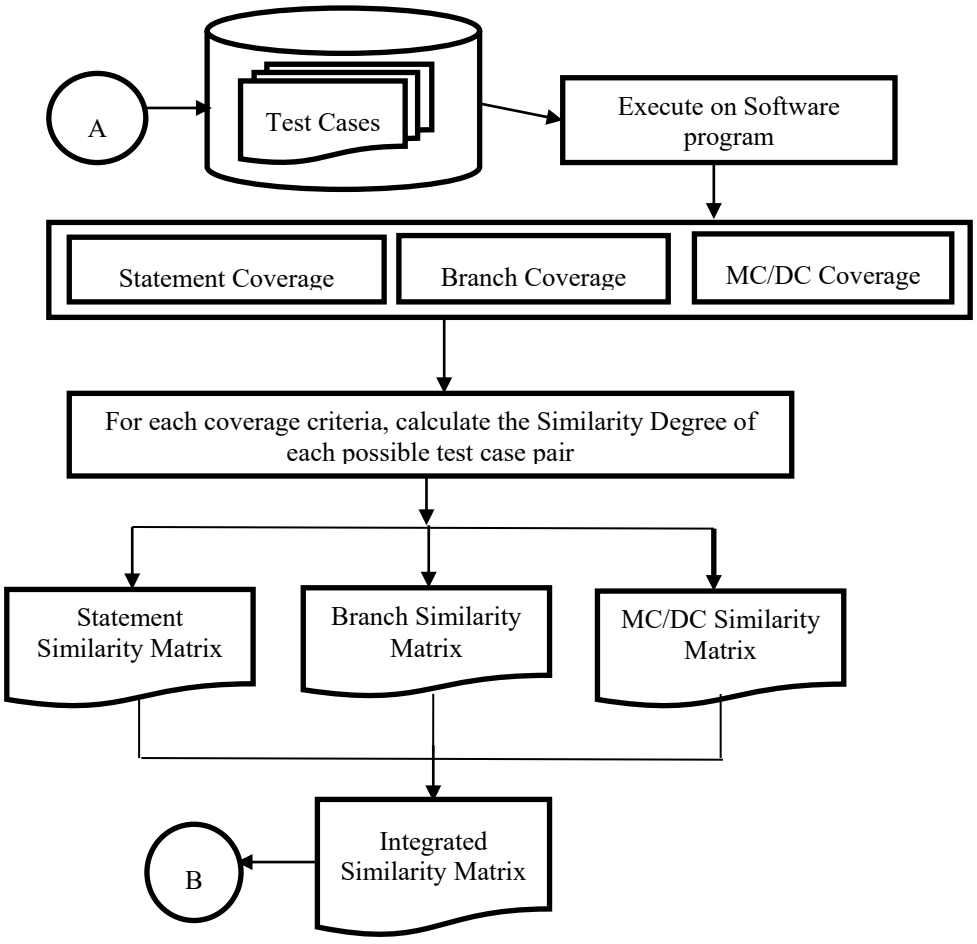


Figure 6.2 (a) Process flow diagram for similarity based test case prioritization

Definition 2: Coverage Similarity; we represent three steps of evaluating similarity degree between two test cases T_i and T_j i.e. (i) Statement similarity specification (CS-I), (ii) MC/DC similarity specification and statement similarity specification (CS-II), and (iii) Branch, statement, and MC/DC similarity specification (CS-III).

Let the Coverage Set of any test case pair T_i and T_j be $\langle S_i, M_i, B_i \rangle$ and $\langle S_j, M_j, B_j \rangle$ respectively. Let the distance between the statement, MC/DC and branch coverage of the test case pair T_i and T_j be $\pi(S_i, S_j)$, $\pi(M_i, M_j)$ and $\pi(B_i, B_j)$. Where, the distance is calculated by applying Eq.6.1.

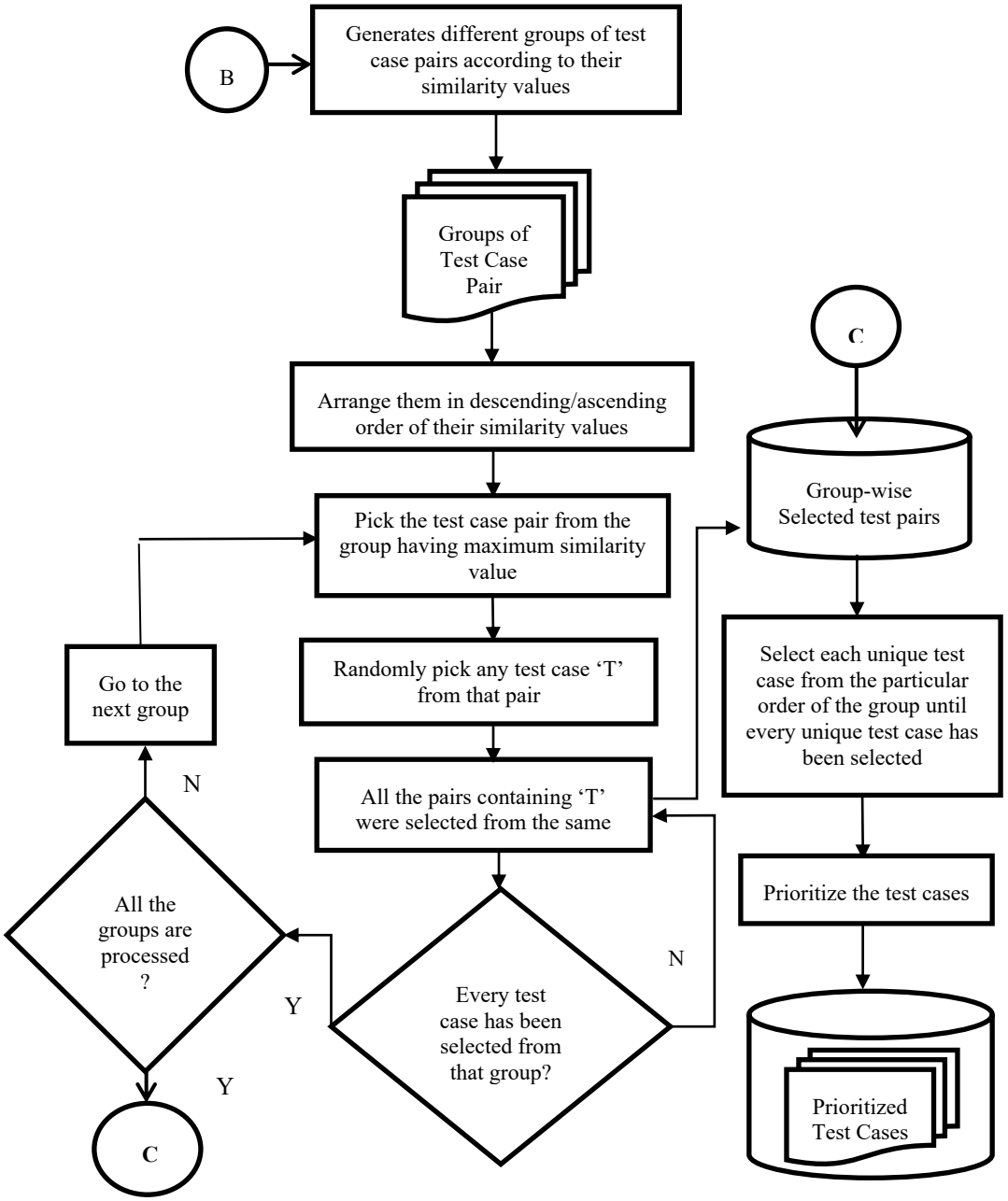


Figure 6.2 (b) Process flow diagram for similarity based test case prioritization

The proposed techniques use the 'Geometric Mean' to evaluate the integrated similarity degree between each pair of test cases T_i and T_j, which is calculated in three steps that is represented by Eq. 6.2, Eq. 6.3, and Eq. 6.4 respectively:

$$CS - I = \pi(S_i, S_j) \tag{6.2}$$

$$CS - II = \sqrt[2]{\pi(S_i, S_j) \times \pi(M_i, M_j)} \tag{6.3}$$

$$CS - II = \sqrt[3]{\pi(S_i, S_j) \times \pi(M_i, M_j) \times \pi(B_i, B_j)} \tag{6.4}$$

However, we demonstrate our techniques using three criteria, but it is also flexible to simplify the above equations to function on more than three criteria.

6.3.2 Test Case Prioritization

Our proposed technique uses similarity degree between each pair of test cases to rank the test cases.

Suppose we have a test suite 'TS' of 'm' number of test cases. Select all possible pairs of different test cases and divide them into K different groups. Each group comprising test case pairs with the same similarity value and it is designated by G_k $(1 \leq k \leq K)$, where k represents the group index. A smaller k specifies all test case pairs of that group having maximum similarity value.

The proposed similarity based prioritization techniques are illustrated as follows:

a) ***PS1 (Total CS-i similarity prioritization):*** The technique generates K different groups of test case pairs using their similarity values CS-i. At first, a test case pair of maximum similarity value is selected and chooses any one test case T from that pair randomly. Continuously all the pairs containing T were selected from the same group. In case of a tie, the technique arbitrarily picks one pair to break this. Repetitively, the above process is executed initially for the group, and once every test case have been selected from that group, then the other remaining groups are processed in the ascending order until every single test case has been selected.

b) ***PS2 (Total CS-i dissimilarity prioritization):*** The technique is similar to PS1, apart from the selection process for test case pair having minimum similarity values, and the above process is executed in ascending order of the index for remaining groups.

c) ***PS3 (Iterative CS-i dissimilarity prioritization):*** In descending order (G_m to G_1) of the group index, the technique selects one test case pair from each group. Excluding the selected pairs, the above selection process repeats on the remaining groups until every test case has been included for prioritization purpose.

d) ***PS4 (Iterative CS-i similarity prioritization):*** This technique is equivalent to the above technique i.e. PS3, apart from the test pair selection process from each group is in ascending order of the group index (G_1 to G_m) instead of in descending order.

e) ***PS5 (Advanced iterative CS-i dissimilarity prioritization):*** The technique selects one test case pair (T_a, T_b) from group G_m and then the further test case pairs are selected containing T_a or T_b, if available, from each group in descending order of the remaining groups (G_{m-1} to G_1). Afterward, the selected test case pair from the group is to be removed. For each remaining group, this technique reiterates the above-discussed selection process until all the test cases have been included for prioritization purpose.

f) ***PS6 (Advanced Iterative CS-i similarity prioritization):*** This technique is quite similar to the above technique i.e. PS3, apart from that the selection process of test case pair from each group is executed in ascending order of the group index (G_1 to G_m), instead of in descending order.

6.4 Experimental Results and Discussion

This section presents the experimentation and evaluation of the proposed prioritization techniques. The experimentation compares the relative performance and effectiveness of the proposed techniques with some standard state-of-the-art prioritization techniques [2].

6.4.1 Subject Program

The subject programs considered for this work have been well structured and without any compilation errors. The proposed algorithm and state-of-the-art algorithms are independent of the programming language. The program description is shown in Table 6.1.

Table 6.1 Subject program description

Program No.	Subject Programs	Program Description
Pr. 1	Pushdown	It pushes the element down through its descendants by a sequence of swaps to its proper position
Pr. 2	Triangle	Return the type of a triangle by three integers
Pr. 3	Prime number	To determine whether a number is prime or not
Pr. 4	Leap Year	To determine whether the year is a leap year or not
Pr. 5	Greatest Number	To find the largest number among the three numbers

6.4.2 Effectiveness Measure

The performance metric "average percentage of faults detected" (APFD) introduced by Elbaum et al. 2002 has been widely used to evaluate any prioritization techniques. A Higher value of APFD concludes improved fault detection rates. APFD values range from 0 to 100. Higher the APFD numbers better the fault detection rates. The APFD values represent the area under the curve by plotting percentage of faults detected on the y-axis, and the percentage of test suite run on the x-axis of the graph.

Let TS be a test suite of n test cases, F represents a set of k faults exposed by the test suite, and TF_i signifies that the fault i reveals by the first test case of the prioritized test suite TS'. The equation to calculate APFD value for test suite TS' is as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \tag{6.5}$$

The APFD value ranges from 0 to 1. The ordered test suite with high APFD value detects more faults as early as possible. For example, consider a test suite consisting of 4 test cases, T1 to T4, and 5 faults detected by those test cases (see Table 6.2). Consider two possible prioritized order of these test cases, P1: T1, T2, T3, T4, and P2: T3, T2, T4, T1.

According to the discussed APFD equation 6.5, P1 produces an APFD of 58% ($1 - \frac{1+1+2+3+4}{4\times5} + \frac{1}{2\times4} = 0.58$) and P2 an APFD of 78% ($1 - \frac{1+1+1+1+3}{4\times5} + \frac{1}{2\times4} = 0.78$). The APFD value reveals that P2 much faster in detecting faults than P1.

Table 6.2 Test suite and faults exposed

Tests/Faults	F1	F2	F3	F4	F5
T1	X	X			
T2	X		X		
T3	X	X	X	X	
T4					X

6.4.3 Case Study

Figure 6.3 presents a control flow diagram of the standard pushdown procedure, a case study on which the proposed techniques have been performed to evaluate the performance. The module pushdown accepts an array $A[1], A[2], \ldots, A[n]$ and two integers 'first' and 'last' representing the first and last items of the array 'A'.

By a sequence of swapping process, it pushes the element A[first] down through its descendants to its appropriate position in the tree until 'A' fulfills the property of partially ordered tree i.e. if $A[i].key \leq A[2 \times i].key \; and \; A[i].key \leq A[2 \times i + 1].key$ for $1 \leq i \leq \lfloor n/2 \rfloor$. The proposed approach also uses hand seeded faults for the subject program which is represented by fault matrix (see Table 6.3).

In the proposed work, three different coverage criteria i.e. statement, MC/DC, and branch coverage are used to evaluate the distance between the pair of test cases. Based on the calculated distance (similarity degree) values, further prioritization will be processed. To collect the coverage information of test cases for each selected criterion, the subject program and the source code were instrumented.

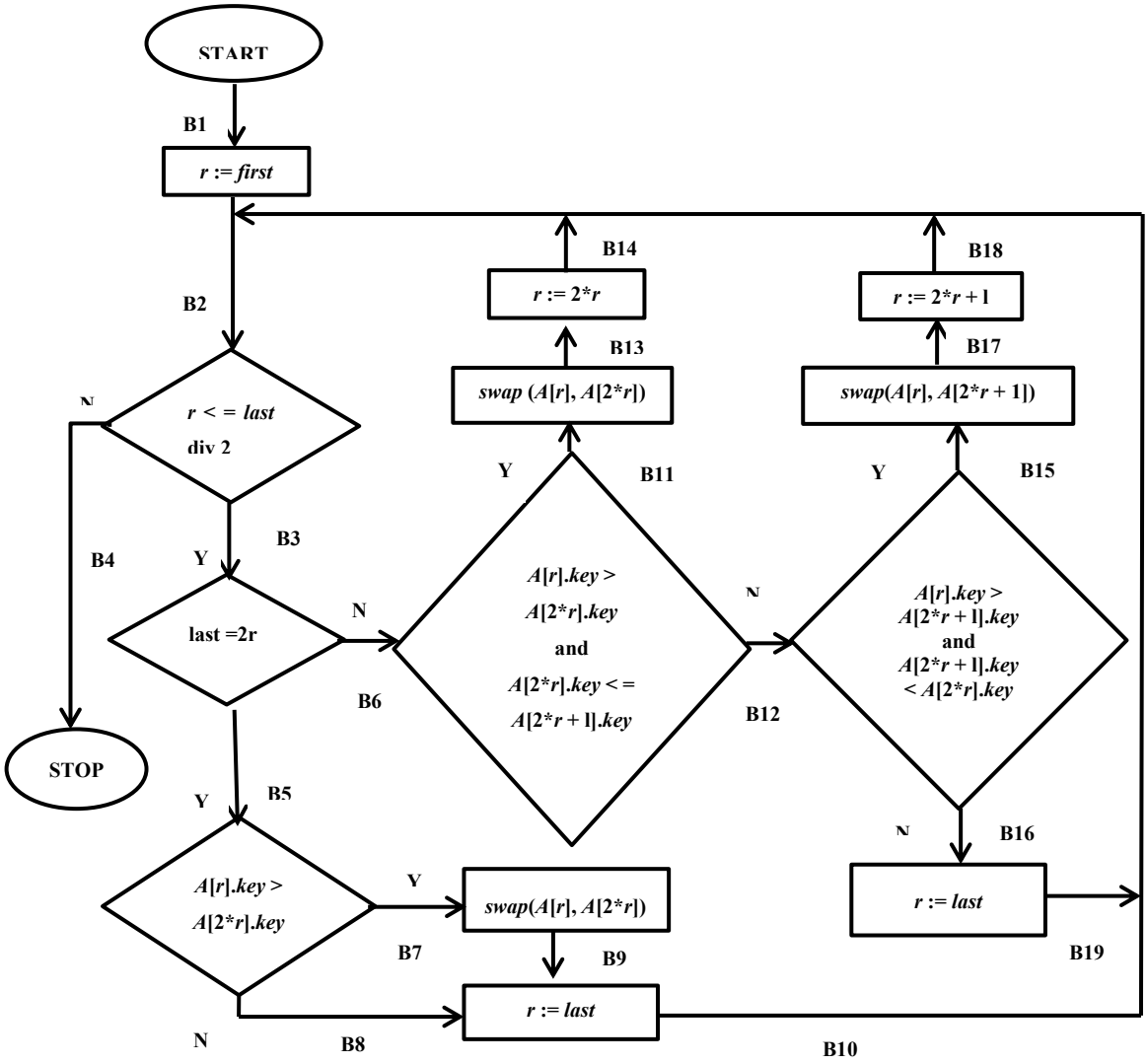


Figure 6.3 Control flow diagram of the pushdown procedure

6.4.4 Results and Discussion

Before applying any prioritization techniques, for each selected criterion the distance between each pair of test cases must be calculated by using Eq. 6.1. We evaluate the similarity values of every possible test case pairs in three levels using Eq. 6.2, Eq. 6.3, and Eq. 6.4. With the help of these coverage similarity values, further prioritization techniques are applied to get ordered test cases. Table 6.6 shows the process of ordering test cases using proposed pair-wise test case prioritization techniques (PS1 to PS6) by considering C-I level similarity values, the different test case pairs of T1 to T12 are categorized into different groups.

Table 6.3 Fault matrix for pushdown procedure

Test Cases	F1	F2	F3	F4	F5	F6	F7
T1		×	×				
T2	×	×		×			
T3	×	×		×	×	×	
T4	×					×	
T5		×	×				
T6							
T7		×	×				
T8	×			×			
T9	×					×	
T10	×	×		×	×		
T11		×	×				
T12	×	×				×	×

By implementing each technique from PS1 – PS6, the possible ordering of test case pairs is shown in the rightmost columns of the tables. Similarly, all the mentioned prioritization techniques are also applied to CS-II and C-III similarity values. And the ordering of test cases are shown in Table 6.7 and Table 6.8. We also analyze the result of proposed and conventional techniques on the subject program that is presented in Table 6.9. For each technique, we deliver a valid ordering of test cases in Table 6.9. For single test case prioritization techniques i.e. P2, P3, and P4, we use three coverage metrics (i.e., Statement, MC/DC, and Branch) to order the test cases. And, for similar test pair prioritization techniques i.e. from PS1 to PS6, we use the coverage similarity metrics i.e. CS-I, CS-II, and CS-III to order the test cases.

Table 6.4 Similarity values of test case pairs for each criterion

Test Cases		T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
T1	Stmt.		0.55	0.25	0.27	0.83	0.33	1.00	0.27	0.50	0.67	0.83	0.60
	MC/DC		0.37	0.20	0.22	0.60	0.33	1.00	0.25	0.33	0.57	0.60	0.50
	Branch		0.46	0.25	0.26	0.66	0.33	1.00	0.26	0.42	0.66	0.66	0.61

T2	Stmt.			0.55	0.33	0.62	0.40	0.55	0.60	0.54	0.88	0.62	0.50
	MC/DC			0.50	0.27	0.57	0.37	0.37	0.62	0.50	0.75	0.57	0.36
	Branch			0.53	0.29	0.54	0.41	0.46	0.57	0.53	0.76	0.63	0.41
T3	Stmt.				0.70	0.27	0.50	0.25	0.70	0.63	0.50	0.27	0.58
	MC/DC				0.66	0.20	0.33	0.20	0.55	0.60	0.50	0.20	0.60
	Branch				0.64	0.26	0.42	0.25	0.64	0.60	0.50	0.26	0.56
T4	Stmt.					0.30	0.75	0.27	0.60	0.70	0.30	0.30	0.63
	MC/DC					0.22	0.57	0.22	0.44	0.66	0.27	0.22	0.66
	Branch					0.28	0.72	0.26	0.57	0.64	0.27	0.28	0.60
T5	Stmt.						0.37	0.83	0.30	0.55	0.55	1.00	0.50
	MC/DC						0.33	0.60	0.25	0.50	0.37	1.00	0.33
	Branch						0.36	0.66	0.28	0.58	0.46	1.00	0.42
T6	Stmt.							0.33	0.75	0.50	0.36	0.37	0.45
	MC/DC							0.33	0.66	0.33	0.37	0.33	0.33
	Branch							0.33	0.72	0.42	0.33	0.36	0.40
T7	Stmt.								0.27	0.50	0.66	0.83	0.60
	MC/DC								0.25	0.33	0.57	0.60	0.50
	Branch								0.26	0.42	0.66	0.66	0.61
T8	Stmt.									0.41	0.54	0.30	0.38
	MC/DC									0.27	0.62	0.25	0.27
	Branch									0.35	0.53	0.28	0.33
T9	Stmt.										0.50	0.55	0.90
	MC/DC										0.36	0.50	0.77
	Branch										0.41	0.63	0.78
T10	Stmt.											0.55	0.58
	MC/DC											0.37	0.50
	Branch											0.46	0.56
T11	Stmt.												0.40
	MC/DC												0.33
	Branch												0.42

Table 6.5 Level-wise integrated coverage similarity values for each test case pair

Test Cases	Levels	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
T1	C-I		0.55	0.25	0.27	0.83	0.33	1.00	0.27	0.50	0.67	0.83	0.60
	C-II		0.45	0.22	0.24	0.70	0.33	1.00	0.25	0.40	0.61	0.70	0.54
	C-III		0.45	0.23	0.24	0.69	0.33	1.00	0.25	0.41	0.63	0.69	0.56
T2	C-I			0.55	0.33	0.62	0.40	0.55	0.60	0.54	0.88	0.62	0.50
	C-II			0.52	0.29	0.59	0.38	0.45	0.60	0.51	0.81	0.59	0.42
	C-III			0.52	0.29	0.57	0.39	0.45	0.59	0.52	0.79	0.60	0.41
T3	C-I				0.70	0.27	0.50	0.25	0.70	0.63	0.50	0.27	0.58
	C-II				0.67	0.23	0.40	0.22	0.62	0.61	0.5	0.23	0.58
	C-III				0.66	0.24	0.41	0.23	0.62	0.60	0.5	0.24	0.57
T4	C-I					0.30	0.75	0.27	0.60	0.70	0.30	0.30	0.63
	C-II					0.25	0.65	0.24	0.51	0.67	0.28	0.25	0.64
	C-III					0.26	0.67	0.24	0.53	0.66	0.27	0.26	0.62
T5	C-I						0.37	0.83	0.30	0.55	0.55	1.00	0.50
	C-II						0.34	0.70	0.27	0.52	0.45	1.00	0.40
	C-III						0.35	0.69	0.27	0.54	0.45	1.00	0.41
T6	C-I							0.33	0.75	0.50	0.36	0.37	0.45
	C-II							0.33	0.70	0.40	0.36	0.34	0.38
	C-III							0.33	0.70	0.41	0.35	0.35	0.39
T7	C-I								0.27	0.50	0.66	0.83	0.60
	C-II								0.25	0.40	0.61	0.70	0.54
	C-III								0.25	0.41	0.62	0.69	0.56
T8	C-I									0.41	0.54	0.30	0.38
	C-II									0.33	0.57	0.27	0.32
	C-III									0.33	0.56	0.27	0.32
T9	C-I										0.50	0.55	0.90

	C-II										0.42	0.52	0.83
	C-III										0.41	0.55	0.81
T10	C-I											0.55	0.58
	C-II											0.45	0.53
	C-III											0.45	0.54
T11	C-I												0.40
	C-II												0.36
	C-III												0.38

Table 6.6 Ordering of test cases using coverage similarity C-I

Similarity Value (C-I)	**Group**		**Selected test pairs in order**						
	Index	**Test Case Pairs**	**Seq. No.**	**PS1**	**PS2**	**PS3**	**PS4**	**PS5**	**PS6**
1	G1	(T1, T7) (T5,T11)	1	(**T1**,**T7**)	(**T1,T3**)	(**T1, T3**)	(**T1, T7**)	(**T1,T3**)	(**T1,T7**)
0.9	G2	(T9,T12)	2	(**T5**,**T11**)	(T3, **T7**)	(T1, **T4**)	(**T9,T12**)	(T1, **T4**)	(T1, **T5**)
0.88	G3	(T2,T10)							
0.83	G4	(T1, T5) (T1,T11) (T5, T7) (T7,T11)	3	(**T9**,**T12**)	(T1, **T4**)	(**T8,T11**)	(**T2,T10**)	(T1, **T6**)	(T1,**T10**)
			4	(**T2**,**T10**)	(T1, **T8**)	(**T6, T7**)	(T1, **T5**)	(T1, **T9**)	(T7,T10)
0.75	G5	(T4, T6) (T6, T8)	5	(T1, T5)	(T3, **T5**)	(T6,**T10**)	(**T4, T6**)	(T1, **T2**)	(T1,**T12**)
0.7	G6	(T3, T4) (T3, T8) (T4, T9)	6	(T1,T11)	(T3,**T11**)	(**T5**, T6)	(**T3**, T4)	(T3,**T12**)	(**T2**, T7)
			7	(T5, T7)	(T4, T7)	(T8,**T12**)	(T1,T10)	(T1,T12)	(T1, **T9**)
0.67	G7	(T1,T10)							
0.66	G8	(T7,T10)	8	(T7,T11)	(T7, T8)	(**T2**, T6)	(T7,T10)	(T3, T9)	(T1, **T6**)
0.63	G9	(T3, T9) (T4,T12)							

0.62	G10	(T2, T5) (T2,T11)	9	(**T4**, **T6**)	(T4, T5)	(T8, **T9**)	(T4,T12)	(T1,**T10**)	(T1, **T4**)
0.60	G11	(T1,T12) (T2, T8) (T4, T8) (T7,T12)	10	(T6, **T8**)	(T4,**T10**)		(T2,**T11**)	(T3, **T8**)	(T1, **T3**)
			11	(**T3**, T4)	(T4,T11)		(T2, **T8**)	(T1, **T5**)	(T5,**T11**)
0.58	G12	(T3,T12)	12		(T5, T8)			(T1, **T7**)	(T1,T11)
0.55	G13	(T1, T2) (T2, T3) (T2, T7) (T5, T9) (T5,T10) (T9,T11) (T10,T11)	13		(T8,T11)			(T3, T7)	(T2, T5)
			14		(T1, **T6**)			(T3,**T11**)	(T5, T9)
			15		(**T2**, T4)				(T5,T12)
0.54	G14	(T2, T9) (T8, T10)	16		(T6, T7)				(T11,T12)
0.50	G15	(T1, T9) (T2, T12) (T3, T6) (T3, T10) (T5, T12) (T6, T9) (T7, T9) (T9, T10)	17		(T6,T10)				(T5, T6)
			18		(T5, T6)				(T5,**T8**)
			19		(T6,T11)				
			20		(T8,**T12**)				
0.45	G16	(T6, T12)	21		(T2, T6)				
0.41	G17	(T8, T9)							
0.4	G18	(T2, T6) (T11,T12)	22		(T11,T12)				
0.38	G19	(T8, T12)	23		(T8, **T9**)				
0.37	G20	(T5, T6) (T6, T11)							
0.36	G21	(T6, T10)							
0.33	G22	(T1, T6) (T2, T4) (T6, T7)							

0.30	G23	(T4, T5) (T4, T10) (T4, T11) (T5, T8) (T8, T11)							

Table 6.7 Ordering of test cases using coverage similarity C-II

Similarity Value (C-II)	Group		Selected Test Pairs In order						
	Index	Test Case Pairs	Seq. No.	PS1	PS2	PS3	PS4	PS5	PS6
1	G1	(T1, T7) (T5, T11)	1	(**T1**, **T7**)	(**T1, T3**)	(**T1, T3**)	(**T1, T7**)	(**T1, T3**)	(**T1, T7**)
0.83	G2	(T9, T12)	2	(**T5**,**T11**)	(T3, **T7**)	(T3, **T5**)	(**T9,T12**)	(T3, **T5**)	(T1, **T5**)
0.81	G3	(T2, T10)							
0.70	G4	(T1, T5) (T1, T11) (T5, T7) (T7,T11)	3	(**T9**,T**12**)	(T3, **T5**)	(**T4**, **T7**)	(**T2,T10**)	(T1, **T4**)	(T1,**T10**)
			4	(**T2**,**T10**)	(T3,**T11**)	(T7, **T8**)	(T1, **T5**)	(T1, **T8**)	(T1,**T12**)
0.67	G5	(T3, T4) (T4, T9)	5	(T1, T5)	(T1, **T4**)	(T8,**T11**)	(**T3, T4**)	(T3, T4)	(T1, **T2**)
0.65	G6	(T4, T6)							
0.64	G7	(T4, T12)	6	(T1,T11)	(T4, T7)	(T4,**T10**)	(T4, **T6**)	(T1, **T6**)	(T1, **T9**)
0.62	G8	(T3, T8)							
0.61	G9	(T1, T10) (T3, T9) (T7, T10)	7	(T5, T7)	(T1, **T8**)	(T3, T4)	(T4,T12)	(T1, **T9**)	(T1, **T6**)
			8	T7, T11)	(T4, T5)	(T8,**T12**)	(T3, **T8**)	(T1, **T2**)	(T1, **T8**)
0.60	G10	(T2, T8)							
0.59	G11	(T2, T2) (T2, T11)	9	(**T3**, **T4**)	(T4,T11)	(T8, **T9**)	(T1,T10)	(T3, **T10**)	(T1, **T4**)
0.58	G12	(T3, T12)							
0.57	G13	(T8, T10)	10	(T4, T9)	(T7, T8)	(T5, **T6**)	(T2, T8)	(T2, T3)	(T1, **T3**)
0.54	G14	(T1, T12) (T7, T12)	11	(T4, **T6**)	(T5, T8)	(T6,T10)	(T2,**T11**)	(T1, **T12**)	(T5,**T11**)
0.53	G15	(T10, T12)							
0.52	G16	(T2, T3)							

		(T5, T9) (T9, T11)	12	(T4,T12)	(T8,T11)	(**T2**, T6)		(T3, T12)	
0.51	G17	(T2, T9) (T4, T8)	13	(T3, **T8**)	(T4,**T10**)			(T1, T10)	
0.50	G18	(T3, T10)	14		(T3, T4)			(T3, T8)	
0.45	G19	(T1, T2) (T2, T7) (T5, T10)	15		(T8, **T12**)			(T3, T4)	
0.42	G20	(T2, T12) (T9, T10)	16		(T1, **T6**)			(T1, **T11**)	
0.40	G21	(T1, T9) (T3, T6) (T5, T12) (T6, T9) (T7, T9)	17 18		(T6, T7) (T8, **T9**)			(T1, **T7**)	
0.38	G22	(T2, T6) (T6, T12)	19		(T5, T6)				
0.36	G23	(T6, T10) (T11, T12)	20		(T6, T11)				
0.34	G24	(T5, T6) (T6, T11)	21		(T6, T10)				
0.33	G25	(T1, T6) (T6, T7) (T8, T9)	22		(T11,T12)				
0.32	G26	(T8, T12)	23		(**T2**, T6)				
0.29	G27	(T3, T4)							
0.28	G28	(T4, T10)							
0.27	G29	(T5, T8) (T8, T11)							
0.25	G30	(T1, T8) (T4, T5) (T4, T11) (T7, T8)							
0.24	G31	(T1, T4) (T4, T7)							
0.23	G32	(T3, T5) (T3, T11)							

0.22	G33	(T1, T3) (T3, T7)							

Table 6.8 Ordering of test cases using coverage similarity C-III

Similarity Value (C-III)	Group		Selected Test Pairs In order						
	Index	Test Case Pairs	Seq. No.	PS1	PS2	PS3	PS4	PS5	PS6
1	G1	(T1,T7) (T5,T11)	1	(**T1**, **T7**)	(**T1, T3**)	(**T1, T3**)	(**T1, T7**)	(**T1, T3**)	(**T1, T7**)
0.81	G2	(T9,T12)							
0.79	G3	(T2,T10)	2	(**T5**,**T11**)	(T3, **T7**)	(T1, **T4**)	(**T9,T12**)	(T1, **T4**)	(T1, **T5**)
0.7	G4	(T6, T8)							
0.69	G5	(T1, T5) (T1,T11) (T5, T7) (T7,T11)	3	(**T9**,T**12**)	(T1, **T4**)	(**T7, T8**)	(**T2,T10**)	(T1, **T8**)	(T7,**T10**)
			4	(**T2**,**T10**)	(T3, **T5**)	(T4, **T5**)	(**T6, T8**)	(T1, **T6**)	(T1,**T12**)
0.67	G6	(T4, T6)	5	(**T6**, **T8**)	(T3, **T11**)	(T4, **T10**)	(T1, **T5**)	(T1, **T9**)	(T1, **T2**)
0.66	G7	(T3, T4) (T4, T9)	6	(T1, T5)	(T4, T7)	(**T2**, T4)	(**T4**, T6)	(T1, **T2**)	(T1, **T9**)
0.62	G8	(T3, T8) (T4,T12 (T7,T10)	7	(T1,T11)	(T1, **T8**)	(T8,**T12**)	(**T3**, T4)	(T3,**T10**)	(**T6**, T7)
0.60	G9	(T2, T11) (T3, T9)	8	(T5, T7)	(T7, T8)	(T1, **T6**)	(T3, T8)	(T2, T3)	(T1, **T8**)
0.59	G10	(T2, T8)							
0.57	G11	(T2, T5) (T3, T12)	9	(T7,T11)	(T4, T5)	(**T11**,T12)	(T2,**T11**)	(T1,**T12**)	(T1, **T4**)
0.56	G12	(T1, T12) (T7, T12) (T8, T10)	10	(**T4**, T6)	(T4, T11)	(T2, T6)		(T3,T12)	(T1, **T3**)
0.55	G13	(T9. T11)	11	(**T3**, T4)	(T4, **T10**)	(T1, **T9**)		(T3, T9)	(T5,**T11**)
0.54	G14	(T5, T9) (T10,T12)							

0.53	G15	(T4, T8)	12		(T5, T8)			(T3, T8)	
0.52	G16	(T2, T3) (T2, T9)	13		(T8, T11)			(T3, T4)	
0.5	G17	(T3, T10)							
0.45	G18	(T1, T2) (T2, T7) (T5, T10) (T10,T11)	14 15		(**T2**, T4) (T8, **T12**)			(T1, **T5**) (T1, **T7**)	
0.41	G19	(T1, T9) (T2, T12) (T3, T6) (T5, T12) (T6, T9) (T7, T9) (T9, T10)	16 17 18		(T1, **T6**) (T6, T7) (T8, **T9**)			(T3, T7) (T3,**T11**)	
0.39	G20	(T2, T6) (T6, T12)							
0.38	G21	(T11,T12)							
0.35	G22	(T5, T6) (T6, T10) (T6, T11)							
0.33	G23	(T1, T6) (T6, T7) (T8, T9)							
0.32	G24	(T8, T12)							
0.29	G25	(T2, T4)							
0.27	G26	(T4, T10) (T5, T8) (T8, T11)							
0.26	G27	(T4, T5) (T4, T11)							
0.25	G28	(T1, T8) (T7, T8)							
0.24	G29	(T1, T4) (T3, T5) (T3, T11) (T4, T7)							
0.23	G30	(T1, T3) (T3, T7)							

Table 6.9 Analysis of Different Prioritization Techniques

Category	Name	Index	Order of Test Cases											
			T1	***T2***	***T3***	***T4***	***T5***	***T6***	***T7***	***T8***	***T9***	***T10***	***T11***	***T12***
Single Test Case Prioritization Strategy	Random	P1	2	4	1	11	8	10	5	3	7	9	12	6
	Total-St Coverage	P2	8	4	1	5	11	9	10	6	2	3	12	7
	Total-MC/DC Coverage	P3	11	5	2	4	9	8	12	7	1	6	10	3
	Total-Br Coverage	P4	9	6	2	5	11	8	10	7	1	4	12	3
Similar Test Pair Prioritization Strategy	Total C-i similarity prioritization (PS1)	C-I	1	7	12	9	3	10	2	11	5	8	4	6
	Total C-i dissimilarity prioritization (PS2)	C-I	1	10	2	4	6	9	3	5	12	8	7	11
	Iterative C-i dissimilarity prioritization (PS3)	C-I	1	11	2	3	9	6	7	4	12	8	5	10
	Iterative C-i similarity prioritization (PS4)	C-I	1	5	10	8	7	9	2	12	3	6	11	4
	Advanced iterative C-i dissimilarity prioritization (PS5)	C-I	1	6	2	3	10	4	11	9	5	8	12	7
	Advanced Iterative C-i similarity prioritization (PS6)	C-I	1	6	10	9	3	8	2	12	7	4	11	5
	Total C-i similarity prioritization	C-II	1	7	9	10	3	11	2	12	5	8	4	6

	(PS1)													
	Total C-i dissimilarity prioritization (PS2)	C-II	1	12	2	6	4	10	3	7	11	8	5	9
	Iterative C-i dissimilarity prioritization (PS3)	C-II	1	12	2	4	3	11	5	6	10	8	7	9
	Iterative C-i similarity prioritization (PS4)	C-II	1	5	8	9	7	10	2	11	3	6	12	4
	Advanced iterative C-i dissimilarity prioritization (PS5)	C-II	1	8	2	4	3	6	12	5	7	9	11	10
	Advanced Iterative C-i similarity prioritization (PS6)	C-II	1	6	11	10	3	8	2	9	7	4	12	5
	Total C-i similarity prioritization (PS1)	C-III	1	7	12	11	3	9	2	10	5	8	4	6
	Total C-i dissimilarity prioritization (PS2)	C-III	1	9	2	4	5	11	3	7	12	8	6	10
	Iterative C-i dissimilarity prioritization (PS3)	C-III	1	8	2	3	6	10	4	5	12	7	11	9
	Iterative C-i similarity prioritization (PS4)	C-III	1	5	11	10	9	7	2	8	3	6	12	4
	Advanced iterative C-i dissimilarity prioritization (PS5)	C-III	1	7	2	3	10	5	11	4	6	8	12	9

	Advanced Iterative C-i similarity prioritization (PS6)	C-III	1	6	11	10	3	8	2	9	7	4	12	5

Table 6.10 APFD metric values for each prioritization strategy

Category	**Name**	**Index**	**Average Percentage of Fault Detected (APFD) %**
Single Test Case Prioritization Strategy	Random	P1	74.40
	Total-St Coverage	P2	70.83
	Total-MC/DC Coverage	P3	68.45
	Total-Br Coverage	P4	68.45
Similar Test Pair Prioritization Strategy	Total C-i similarity prioritization (PS1)	C-I	79.16
	Total C-i dissimilarity prioritization (PS2)	C-I	77.97
	Iterative C-i dissimilarity prioritization (PS3)	C-I	74.40
	Iterative C-i similarity prioritization (PS4)	C-I	73.21
	Advanced iterative C-i dissimilarity prioritization (PS5)	C-I	72.02
	Advanced Iterative C-i similarity prioritization (PS6)	C-I	76.78
	Total C-i similarity prioritization (PS1)	C-II	75.59
	Total C-i dissimilarity prioritization (PS2)	C-II	83.92

	Iterative C-i dissimilarity prioritization (PS3)	C-II	85.11
	Iterative C-i similarity prioritization (PS4)	C-II	69.64
	Advanced iterative C-i dissimilarity prioritization (PS5)	C-II	77.97
	Advanced Iterative C-i similarity prioritization (PS6)	C-II	68.45
	Total C-i similarity prioritization (PS1)	C-III	79.16
	Total C-i dissimilarity prioritization (PS2)	C-III	74.40
	Iterative C-i dissimilarity prioritization (PS3)	C-III	76.78
	Iterative C-i similarity prioritization (PS4)	C-III	68.45
	Advanced iterative C-i dissimilarity prioritization (PS5)	C-III	72.02
	Advanced Iterative C-i similarity prioritization (PS6)	C-III	64.88

To validate the results, we measure the APFD value of the prioritized test suite by each technique using Eq. 5. For example, the resulting APFD's for the three test case orders (Random, Total-St Coverage, and Advanced Iterative CS-i Dissimilarity Prioritization) are 62.56%, 38.62%, and 72.0%, respectively. The test order by PS-5 is, in fact, an optimal order of the test suite in comparison to the test order by P-1 (random) and P-2, ensuring the earliest discovery of the maximum faults. The APFD percentage illustrates that the similarity based prioritization techniques gives a better result as compared to the other conventional techniques. The analysis of different prioritization techniques is illustrated in Table 4.

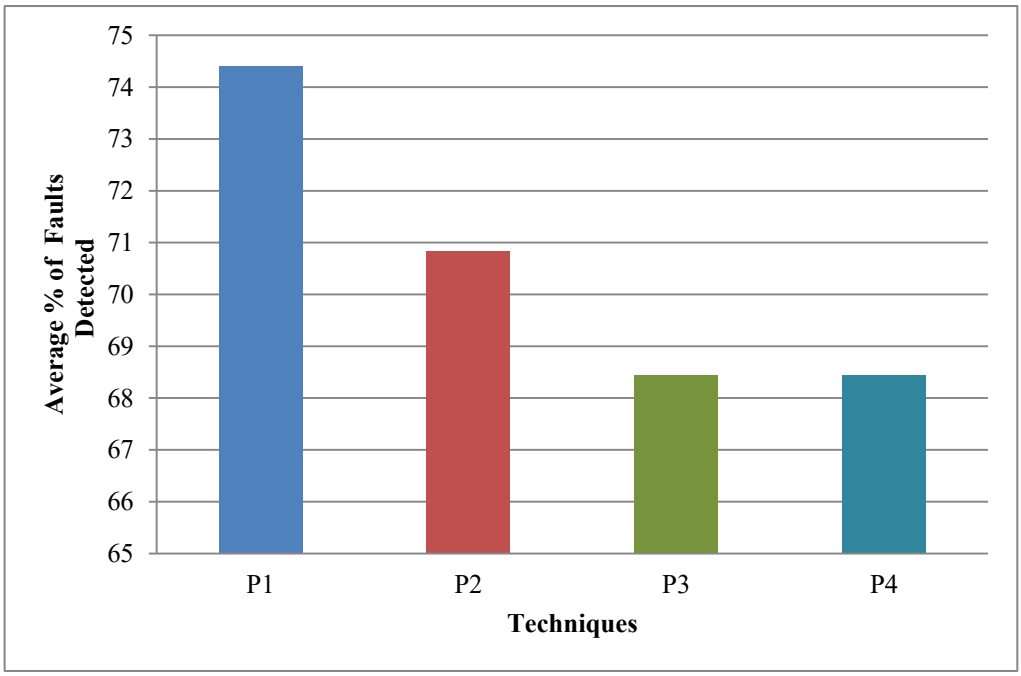


Figure 6.4 APFD% based on single test case prioritization strategies

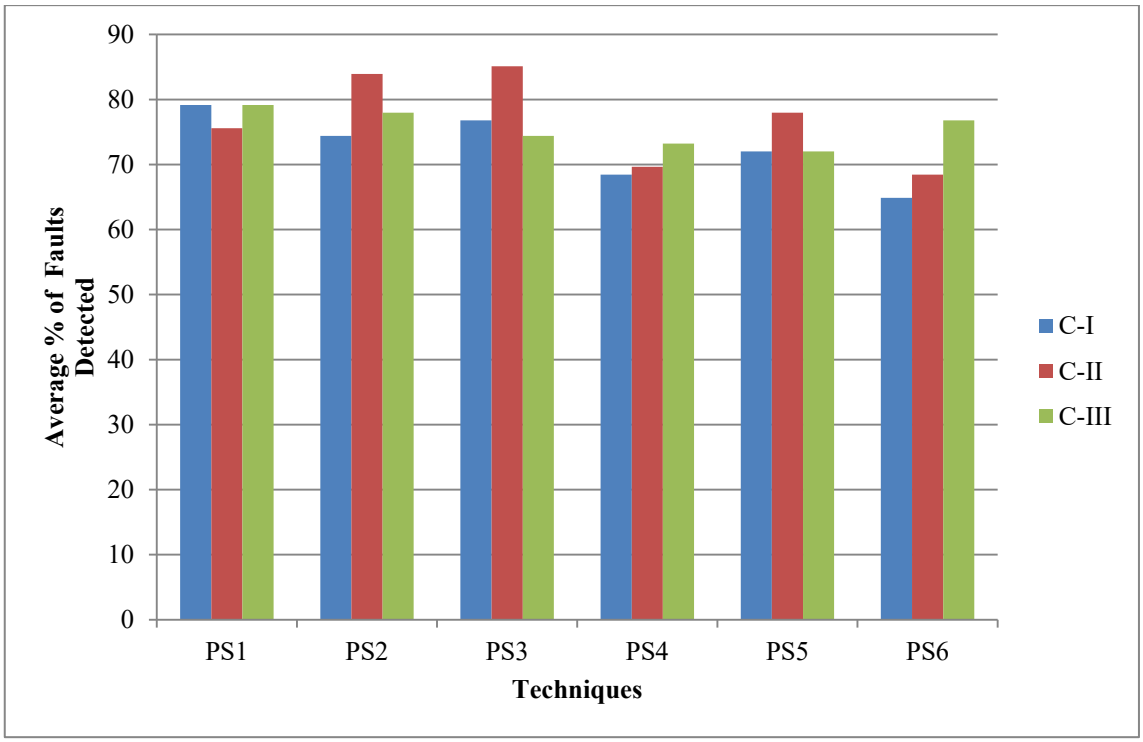


Figure 6.5 APFD% based on similar test pair prioritization strategies

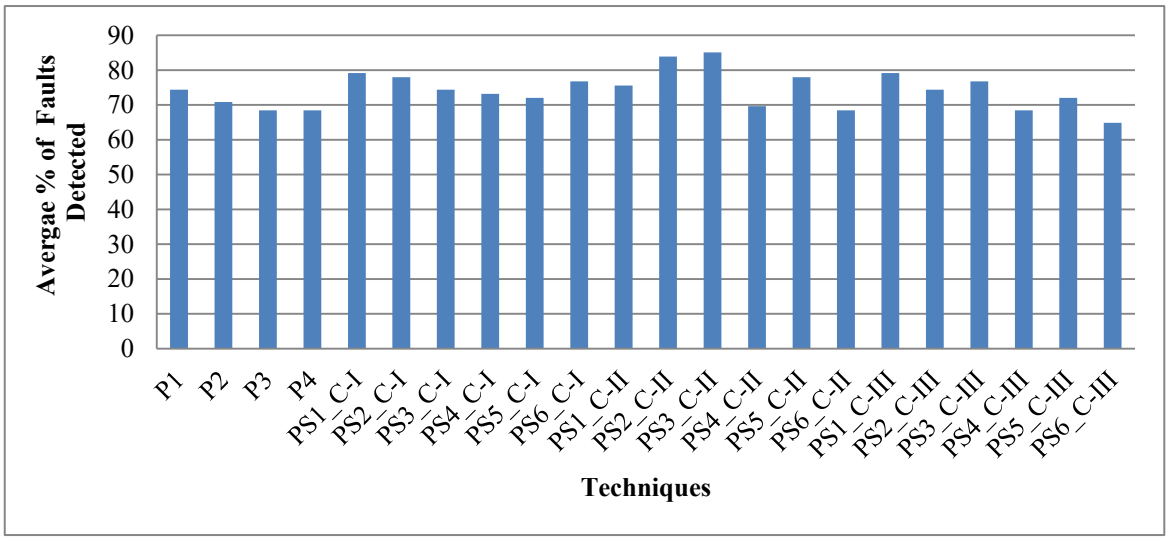


Figure 6.6 APFD% based on various prioritization schemes for pushdown program

The graphical representation of APFD% for pushdown procedure using single test case prioritization strategies is shown in Figure 6.4 and for similar test pair prioritization strategies the observed values is shown in Figure 6.5. The same methodology discussed so far in this chapter for applying proposed prioritization schemes in the experimental case study is applied to other remaining programs. For the remaining subject programs, measuring the similarity values of test case pair and grouping process is carried out systematically as discussed in this chapter. The results generated by the proposed prioritization schemes are summarized in Table 6.11. The graphical representation of the comparative analysis based on APFD for each subject program is given in Figure 6.6 to Figure 6.10. The y-axis in the graph represents the APFD percentage against the techniques considered in the x-axis. From Figure 6.6 to Figure 6.10 it can be concluded that the proposed similar test pair prioritization strategy improved the performance as compared to the random and single test case prioritization strategy. The result of comparative analysis also reveals that when the proposed prioritization approach is used the APFD for the subject programs ranges from 74% to 75%. The experimental results indicate that the APFD obtained by applying our proposed approach is on an average greater than that one obtained by applying the random or single test case prioritization strategy. Out of the proposed prioritization techniques, Total C-i dissimilarity

prioritization (PS2) and Iterative C-i dissimilarity prioritization (PS3) performs better than other proposed techniques.

Table 6.11 Summary of APFD % from different prioritization schemes

Subject Programs	Techniques	Average Percentage of Fault Detected (APFD) %
Pushdown	P1	74.40
	P2	70.83
	P3	68.45
	P4	68.45
	PS1_C-I	79.16
	PS2_C-I	77.97
	PS3_C-I	74.40
	PS4_C-I	73.21
	PS5_C-I	72.02
	PS6_C-I	76.78
	PS1_C-II	75.59
	PS2_C-II	83.92
	PS3_C-II	85.11
	PS4_C-II	69.64
	PS5_C-II	77.97
	PS6_C-II	68.45
	PS1_C-III	79.16
	PS2_C-III	74.40
	PS3_C-III	76.78
	PS4_C-III	68.45

	PS5_C-III	72.02
	PS6_C-III	64.88
Triangle	P1	74.40
	P2	70.83
	P3	68.45
	P4	68.45
	PS1_C-I	78.52
	PS2_C-I	76.90
	PS3_C-I	74.40
	PS4_C-I	70.21
	PS5_C-I	72.22
	PS6_C-I	74.52
	PS1_C-II	74.45
	PS2_C-II	84.33
	PS3_C-II	85.11
	PS4_C-II	69.64
	PS5_C-II	76.40
	PS6_C-II	78.50
	PS1_C-III	83.16
	PS2_C-III	73.64
	PS3_C-III	76.98
	PS4_C-III	69.40
	PS5_C-III	72.22
	PS6_C-III	73.80
Prime Number	P1	64.40
	P2	69.80

	P3	58.50
	P4	70.45
	PS1_C-I	78.16
	PS2_C-I	78.22
	PS3_C-I	75.45
	PS4_C-I	72.21
	PS5_C-I	74.22
	PS6_C-I	70.98
	PS1_C-II	74.97
	PS2_C-II	84.02
	PS3_C-II	83.21
	PS4_C-II	70.64
	PS5_C-II	78.16
	PS6_C-II	71.42
	PS1_C-III	80.16
	PS2_C-III	72.44
	PS3_C-III	78.52
	PS4_C-III	70.45
	PS5_C-III	71.09
	PS6_C-III	65.12
Leap Year	P1	68.40
	P2	62.50
	P3	69.40
	P4	70.12
	PS1_C-I	80.25
	PS2_C-I	77.78

	PS3_C-I	75.40
	PS4_C-I	72.31
	PS5_C-I	75.22
	PS6_C-I	77.09
	PS1_C-II	75.70
	PS2_C-II	84.77
	PS3_C-II	85.60
	PS4_C-II	69.40
	PS5_C-II	75.19
	PS6_C-II	66.40
	PS1_C-III	70.16
	PS2_C-III	76.98
	PS3_C-III	72.22
	PS4_C-III	65.70
	PS5_C-III	78.25
	PS6_C-III	65.87
Greatest Number	P1	62.02
	P2	70.03
	P3	65.13
	P4	72.50
	PS1_C-I	83.61
	PS2_C-I	80.80
	PS3_C-I	78.50
	PS4_C-I	73.30
	PS5_C-I	72.98
	PS6_C-I	72.94

	PS1_C-II	79.22
	PS2_C-II	83.92
	PS3_C-II	80.10
	PS4_C-II	74.16
	PS5_C-II	75.97
	PS6_C-II	58.09
	PS1_C-III	83.88
	PS2_C-III	78.52
	PS3_C-III	73.50
	PS4_C-III	62.45
	PS5_C-III	72.22
	PS6_C-III	70.42

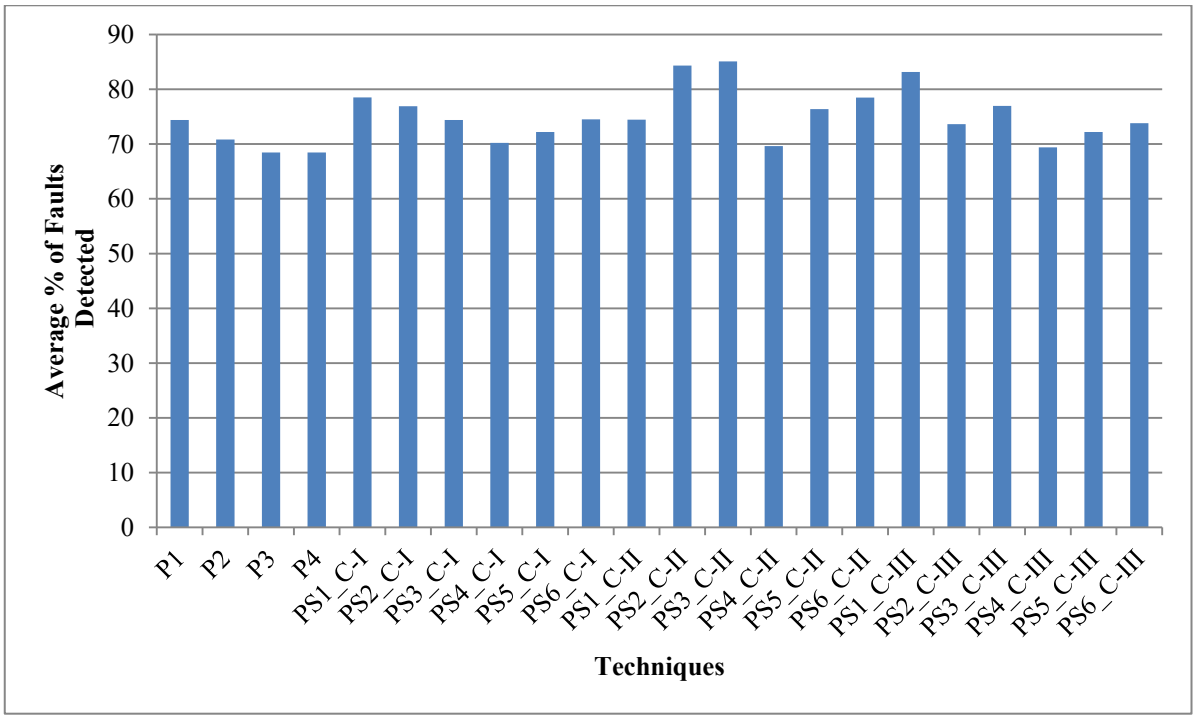


Figure 6.7 APFD values based on various prioritization schemes for Triangle Program

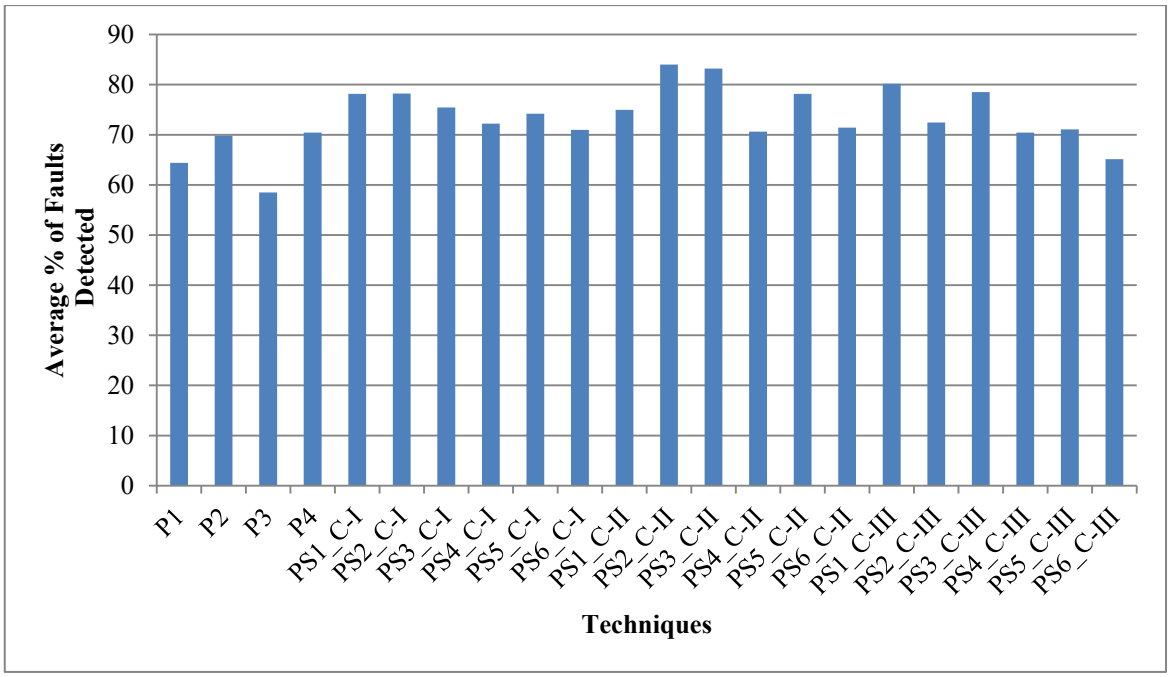


Figure 6.8 APFD values based on various prioritization schemes for Prime Number program

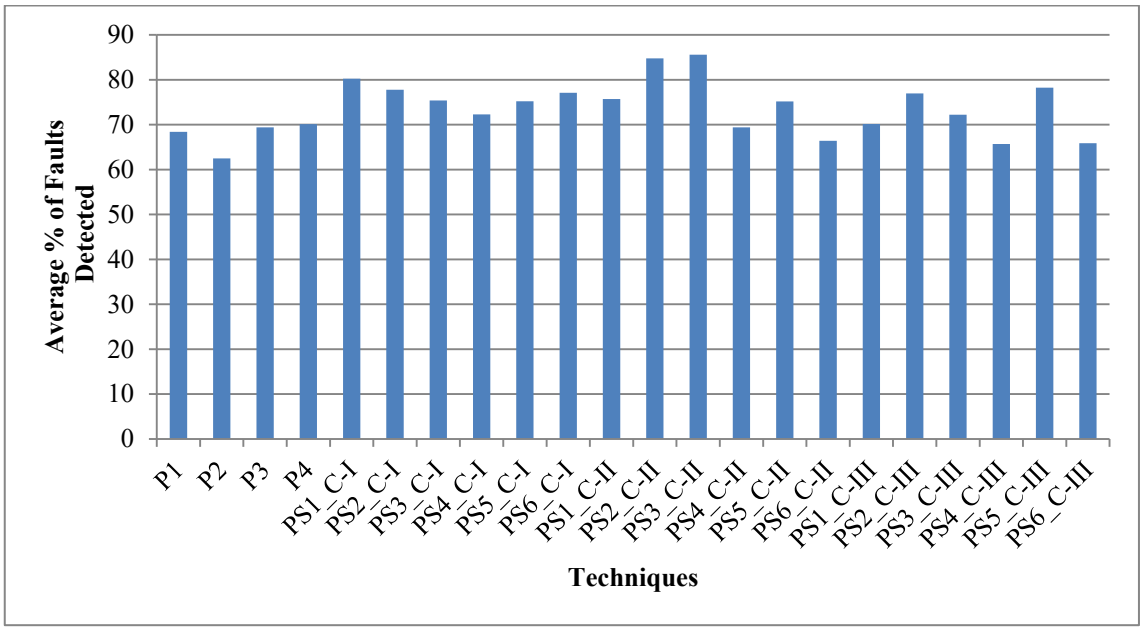


Figure 6.9 APFD values based on various prioritization schemes for Leap Year program

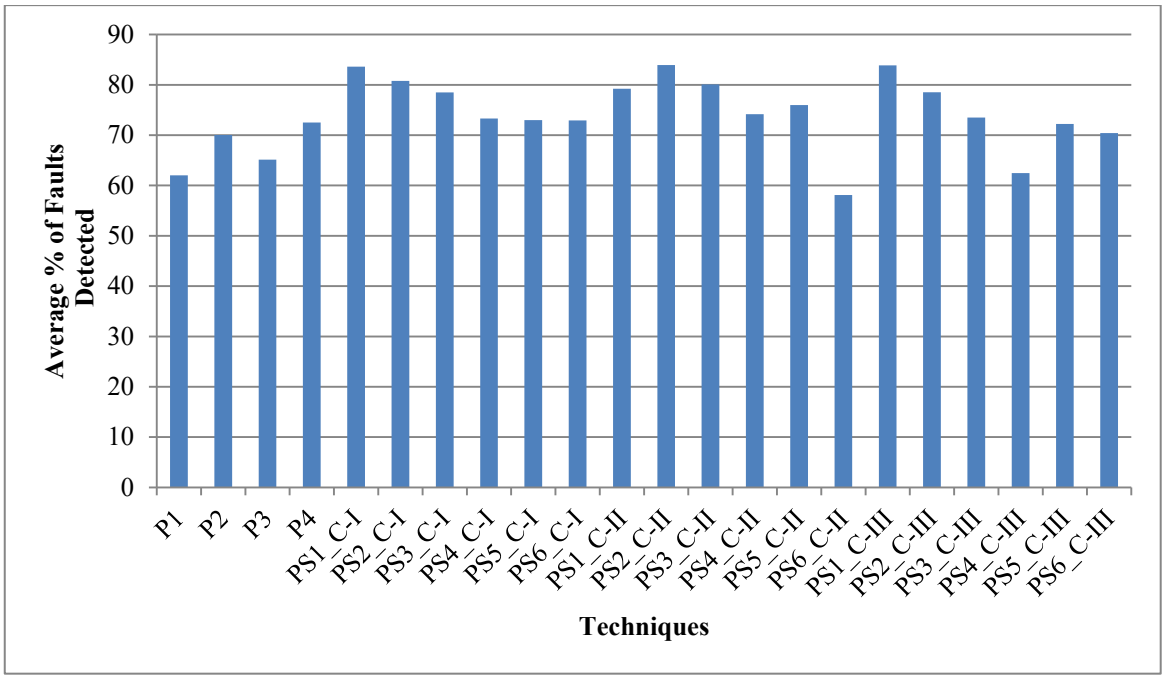


Figure 6.10 APFD values based on various prioritization schemes for Greatest Number program

6.5 Statistical Validation

Statistical Hypothesis testing is conducted to assess whether or not the Average Percentage of Fault Detected (APFD) is improved in the prioritized test suite as compared to the test suite generated by random or single test case prioritization strategy. Since the sample size is small, the student T-test is applied to find out the level of significance and reject the null hypothesis.

The student's T-test is one of the most commonly used techniques to test a hypothesis based on the difference between sample means. Since the rejection or acceptance of a null hypothesis is based on either (0.05) alpha (α) or (0.01) alpha (α) level of significance for one tailed or two tailed test, (0.01) alpha (α) level of significance for a two-tailed test is taken for rejection of the null hypothesis. A null hypothesis shows that there is no significant relationship between two or more parameters, while the alternative hypothesis proves relationships. The rejection of a null hypothesis provides a strong basis for accepting the relationship or accepting the alternative hypothesis.

The formulated hypothesis is mentioned below:

Null Hypothesis $(\boldsymbol{H_{01}})$**:** Average Percentage of Faults Detected (APFD %) cannot be improved using the proposed approach as compared to the random approach.

Alternative Hypothesis $(\boldsymbol{H_{11}})$**:** Average Percentage of Faults Detected (APFD %) can be improved using the proposed approach as compared to the random approach.

Null Hypothesis $(\boldsymbol{H_{02}})$**:** Average Percentage of Faults Detected (APFD %) cannot be improved using the proposed approach as compared to Single Test Case Prioritization Strategy.

Alternative Hypothesis $(\boldsymbol{H_{12}})$**:** Percentage of Suite Size Reduction (SSR %) can be improved using the proposed approach as compared to Single Test Case Prioritization Strategy.

To find out the significance of the difference between the Random and Single Test Case Prioritization Strategy against the Similar Test Pair Prioritization Strategy in terms of APFD, the means of both old and new APFD are calculated as shown in Table 6.12 and Table 6.13. Pearson coefficient of correlation shows that the old APFD values before treatment and new values of APFD after treatment are highly correlated. The degree of freedom for both APFD values is 4.

The t value comes out to be 2.572 and 12.911 respectively while comparing with random and Single Test Case Prioritization Strategy. As the values exceed the t critical value i.e. 2.376 and 2.776 for a two-tailed test at the 0.05 level for 4 degrees of freedom, the null hypothesis H_{01} and H_{02} are strongly rejected and the alternate hypothesis H_{11} and H_{12} are accepted.

Hence it is validated that APFD value is improved by applying Similar Test Pair Prioritization Strategy with multiple criteria as compared to random and Single Test Case Prioritization Strategy. Overall our approach is highly effective for test case prioritization.

Table 6.12 T-Test: Paired Two Sample for Means in terms of APFD (Random vs. Similar Test Pair Prioritization Strategy)

Statistical Observation	Random	Similar Test Pair Prioritization Strategy
Mean	68.724	75.108
Variance	32.04488	0.19117
Observations	5	5
Pearson Correlation	0.291914	
Hypothesized Mean Difference	0	
df	4	
t Stat	-2.57256	
P(T<=t) one-tail	0.030906	
t Critical one-tail	2.131847	
P(T<=t) two-tail	0.021811	
t Critical two-tail	2.376445	

Table 6.13 T-Test: Paired Two Sample for Means in terms of APFD (Single Test Case Prioritization Strategy vs. Similar Test Pair Prioritization Strategy)

Statistical Observation	Single Test Case Prioritization Strategy	Similar Test Pair Prioritization Strategy
Mean	68.258	75.108
Variance	1.93222	0.19117
Observations	5	5
Pearson Correlation	0.589155	
Hypothesized Mean Difference	0	

df	4	
t Stat	-12.9119	
P(T<=t) one-tail	0.000104	
t Critical one-tail	2.131847	
P(T<=t) two-tail	0.000208	
t Critical two-tail	2.776445	

6.6 Summary

Pair wise selection of test case is a fundamental strategy to compare and prioritize them by measuring their associations between them. Using the idea of similarity approach with multiple criteria is quite attractive to arrange the test cases in some specific order. In this chapter, we have proposed test case prioritization techniques based on this strategy for the regression testing. The main target of this proposed prioritization method is to order the test cases based on the similarity between test cases to conduct regression testing more effectively to achieve more than one performance goals such as statement coverage, branch coverage, and MC/DC coverage. To attain this goal the test cases are prioritized based on level-wise calculated similarity value for each pair of test cases. Based on such quantification, we have developed a family of new prioritization techniques to rearrange test cases. We have empirically verified that our techniques are achievable, and some of them are more effective than existing techniques or random ordering in terms of APFD.

It is necessary to statistically validate the approach to make it acceptable in the society. This chapter validates the proposed work using Student T-test to test the hypothesis. The observed values after T-test reveal that the proposed prioritization techniques consistently outperformed the Random and Single Test Case Prioritization Strategy in terms of APFD.

Chapter 7 Concluding Remarks

This chapter summarizes the main contribution of the research and also presents some suggestions for the future in the area of test suite optimization. First, the conclusions are drawn in Section 7.1 and the possible future works are presented in Section 7.2.

7.1 Conclusions

The main objective of our work is to develop some cost-efficient algorithms to optimize the test suite for regression testing. The two important contribution of our thesis towards regression testing is in the field of Test Suite Minimization and Test Case Prioritization. The main objective of this doctorate research is to improve the process of test suite minimization and prioritization by proposing a similarity based optimization strategy aiming to improve the fault detection effectiveness (FDE) of the minimized test suite. The Chapter 1 and Chapter 2 discuss the fundamental concepts of software testing, regression testing and introduce test suite minimization and test case prioritization as an important optimization technique. Chapter 3 discovers the relevant literature on minimization and prioritization techniques. The literature survey on these techniques using different approaches has been done and in particular code coverage and similarity-based approaches are analyzed in detail. In this work, similarity-based test suite minimization and prioritization techniques are implemented, evaluated and results are presented respectively in Chapters 4, Chapter 5 and Chapter 6. This chapter presents the overall conclusions and future scope of work.

A similarity-based greedy approach is proposed to get an optimal test suite. The goal is to apply a similarity-based strategy with multiple criteria to identify the difference between a pair of test cases and compare them to similarity level. The work concentrates on the combination of regression testing techniques i.e. minimization and

prioritization with different coverage criterion to optimize the test suite size. The main idea is to analyze the test cases first to know the difference or similarity value of the test case pairs and further apply the greedy and clustering approach to optimize the test cases accordingly. We have proposed the SBGA algorithm for optimizing the test suite. The proposed approach can be very helpful when the fault detection effectiveness is more important as compared to code coverage. The experimental study is conducted on different subject programs to evaluate the performance of the proposed approach. To evaluate the effectiveness of the proposed work two performance metrics were used i.e. SSR and FDL. And the experimental results show that the fault detection ability is highly improved by the proposed technique as compared to an existing technique.

We also proposed a similarity-based test suite optimization (SB-TSO) approach based on the comparison of similarity between test cases in a given test suite. An important consideration in reducing test suits is to remove the most common test cases according to the calculated similarity or diversity values between test case pair. So that there are fewer similar test cases in the representative test suite, which aims to have better coverage in terms of testing requirements and faults. A single coverage criterion is not sufficient to select the diverse test cases, thus the key idea behind this approach is to use more than one coverage criteria i.e. statement, MC/DC and branch coverage to measure the similarity degree for each pair of test cases. So that the minimized test suites have the maximum diversity as well as better requirement and fault coverage. The paper also utilizes a clustering approach to speed up the minimization process. On the other hand, the test cases are also ranked based on their weight to maximize the fault detection effectiveness. The experimental results indicate that the minimized test suite sizes obtained by applying our proposed approach are in an average greater than that one obtained by applying the state-of-the-art algorithms. Considering the fault detection loss, it can be observed that in an average of the cases our strategy presents a similar behavior as compared to other heuristics.

We have proposed different test case prioritization techniques based on similarity-based strategy for effective regression testing. A similarity-based approach is used to identify the difference level between a pair of test cases which quantitatively illustrates that how much the test case pair are similar or diverse to each other. Our techniques consider three coverage criteria i.e. statement, MC/DC, and branch coverage associated with

each pair of test cases to calculate the similarity value between them. The proposed techniques reorganize the execution order of test cases based on the similarity value of test case pairs computed in three levels. Each level represents the integration of selected coverage criteria. Based on such quantification, we have developed a family of new prioritization techniques to rearrange the test cases. We designed and conducted empirical studies on different sample program to validate the effectiveness of our proposed techniques. The result shows that by incorporating a similarity-based approach with multiple coverage criteria; prioritization can be more effective than any other conventional coverage based approaches in terms of APFD (Average Percentage of Faults Detected).

Different proposed optimization approaches are experimentally evaluated by conducting various case studies and their performances are also compared with traditional approaches with the help of some standard measures. Therefore, using the proposed similarity-based test suite optimization approach, the software tester/user obtains optimal or near-optimal solutions for optimization.

7.2 Future Works

The following suggestions may be taken up for further research and as an extension to the present work.

- An automation of the regression testing process by developing a tool is under active consideration of us. The whole process for generating coverage information should be automated so that public at large can use it.
- This research work concentrates on optimization based on some selected coverage criteria such as statement, branch, def-use, control-flow and MC/DC coverage of the test cases. In the future research work, this can be done based on other possible combination of coverage criteria.
- We have used agglomerative clustering approach for making group of similar test cases. As a future work, the other types of clustering methods can be applied to the optimization process and the performance of them can be compared in achieving optimal test suite.

- In the future, we aim at exploring different distance measures to calculate a similarity degree between test case pair and will analyze the results accordingly.
- These proposed methods should be compared with another existing method for strengthening its uses.
- These algorithms have been tested for subject programs and our aim is to test them using real-life projects with more LOC.
- We have used similarity-based approach with multi-coverage criteria for TCM and TCP problem but it is not used in TCS, so we aim at exploring the scope of using that approach for TCS of regression testing.
- We will also investigate the possibility of automated test case generation.

References

[1] Aggarwal, K. K., & Singh, Y. (2005). Software engineering programs documentation, operating procedures. *New Age International Publishers, Revised Second Edition–2005*.

[2] Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2002). Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, *28*(2), 159-182.

[3] Bentley, J. E., & Bank, W. (2005, April). Software testing fundamentals-concepts, roles, and terminology. In *Proceedings of SAS Conference* (pp. 1-12).

[4] Beizer, B. (1990). Software Testing Techniques, Van Nostrand Reinhold. *Inc, New York NY, 2nd edition. ISBN 0-442-20672-0*.

[5] Do, H., Mirarab, S., Tahvildari, L., & Rothermel, G. (2008, November). An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (pp. 71-82). ACM.

[6] Edelstein, D. V. (1993). Report on the IEEE STD 1219–1993—standard for software maintenance. *ACM SIGSOFT Software Engineering Notes*, *18*(4), 94-95.

[7] Ellims, M., Bridges, J., & Ince, D. C. (2006). The economics of unit testing. *Empirical Software Engineering*, *11*(1), 5-31.

[8] Korel, B. (1990). Automated software test data generation. *IEEE Transactions on software engineering*, *16*(8), 870-879.

[9] Koochakzadeh, N. (2009). A measurement, detection, and visualization framework for software test redundancy.

[10] Malik, Q. A. (2010). Combining model-based testing and stepwise formal development.

[11] Mala, D. J., & Mohan, V. (2010). Quality improvement and optimization of test cases: a hybrid genetic algorithm based approach. *ACM SIGSOFT Software Engineering Notes*, *35*(3), 1-14.

[12] Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave Macmillan.

[13] Grottke, M., & Trivedi, K. S. (2007). Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, *40*(2).

[14] Tassey, G. (2002). *RTI: The economic impacts of inadequate infrastructure for software testing*. Technical Report 02-3, National Institute of Standards and Technology, Gaithersburg, MD, USA (May 2002).

[15] Radatz, J., Geraci, A., & Katki, F. (1990). IEEE standard glossary of software engineering terminology. *IEEE Std*, *610121990*(121990), 3.

[16] Rapps, S., & Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE transactions on software engineering*, (4), 367-375.

[17] Saif-ur-Rehman Khan, A. N., & Awais, A. (2006). TestFilter: a statement-coverage based test case reduction technique. In *IEEE International Multitopic Conference* (pp. 275-280).

[18] DeMilli, R. A., & Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, *17*(9), 900-910.

[19] Chilenski, J. J., & Miller, S. P. (1994). Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, *9*(5), 193-200.

[20] Binkley, D. (1997). Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, *23*(8), 498-516.

[21] Zhong, H., Zhang, L., & Mei, H. (2008). An experimental study of four typical test suite reduction techniques. *Information and Software Technology*, *50*(6), 534-546.

[22] Myers, G. J. (1979). IBM Systems Research Institute, Lecturer in Computer Science, Polytechnic Institute of New York, The Art of Software Testing, by John Wiley & Sons.

[23] Mathur, A. P. (2013). *Foundations of software testing, 2/e*. Pearson Education India.

[24] Xiao, M., El-Attar, M., Reformat, M., & Miller, J. (2007). Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering*, *12*(2), 183-239.

[25] Binder, R. V. (2000). *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional.

[26] Khan, S. U. R., Lee, S. P., Ahmad, R. W., Akhunzada, A., & Chang, V. (2016). A survey on Test Suite Reduction frameworks and tools. *International Journal of Information Management*, *36*(6), 963-975.

[27] Inozemtseva, L., & Holmes, R. (2014, May). Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 435-445). ACM.

[28] Hemmati, H., Arcuri, A., & Briand, L. (2010, November). Reducing the cost of model-based testing through test case diversity. In *IFIP International Conference on Testing Software and Systems* (pp. 63-78). Springer, Berlin, Heidelberg.

[29] Marick, B. (1995). *The craft of software testing: subsystem testing including object-based and object-oriented testing*. PTR Prentice Hall.

[30] Wang, R., Jiang, S., & Chen, D. (2015). Similarity-based regression test case prioritization. In *SEKE* (pp. 358-363).

[31] Cartaxo, E. G., Machado, P. D., & Neto, F. G. O. (2011). On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, *21*(2), 75-100.

[32] Gupta, R., & Soffa, M. L. (1993). Employing static information in the generation of test cases. *Software Testing, Verification and Reliability*, *3*(1), 29-48.

[33] DeMilli, R. A., & Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, *17*(9), 900-910.

[34] Harrold, M. J., & Orso, A. (2008, September). Retesting software during development and maintenance. In *Frontiers of Software Maintenance, 2008. FoSM 2008.* (pp. 99-108). IEEE.

[35] Zhong, H., Zhang, L., & Mei, H. (2008). An experimental study of four typical test suite reduction techniques. *Information and Software Technology*, *50*(6), 534-546.

[36] Ilkhani, A., & Abaee, G. (2010, October). Extraction test cases by using data mining; reducing the cost of testing. In *CISIM* (pp. 620-625).

[37] Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, *22*(2), 67-120.

[38] Leung, H. K., & White, L. (1989, October). Insights into regression testing (software testing). In *Software Maintenance, 1989., Proceedings., Conference on* (pp. 60-69). IEEE.

[39] Mathur, A. P. (2013). *Foundations of software testing, 2/e*. Pearson Education India.

[40] IEEE standard for software maintenance. IEEE Std 1219-1998, Oct 1998.

[41] Harrold, M. J., Gupta, R., & Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *2*(3), 270-285.

[42] Chen, T. Y., & Lau, M. F. (1996). Dividing strategies for the optimization of a test suite. *Information Processing Letters*, *60*(3), 135-141.

[43] Wong, W. E., Horgan, J. R., London, S., & Mathur, A. P. (1998). Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, *28*(4), 347-369.

[44] Marré, M., & Bertolino, A. (2003). Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, *29*(11), 974-984.

[45] Tallam, S., & Gupta, N. (2006). A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes*, *31*(1), 35-42.

[46] Jeffrey, D., & Gupta, N. (2005, September). Test suite reduction with selective redundancy. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on* (pp. 549-558). IEEE.

[47] Jeffrey, D., & Gupta, N. (2007). Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on software Engineering*, *33*(2).

[48] Black, J., Melachrinoudis, E., & Kaeli, D. (2004, May). Bi-criteria models for all-uses test suite reduction. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference On* (pp. 106-115). IEEE.

[49] Hsu, H. Y., & Orso, A. (2009, May). MINTS: A general framework and tool for supporting test-suite minimization. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 419-429). IEEE Computer Society.

[50] Heimdahl, M. P., & George, D. (2004, September). Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE international conference on Automated software engineering* (pp. 176-185). IEEE Computer Society.

[51] Rothermel, G., Harrold, M. J., Ostrin, J., & Hong, C. (1998, March). An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *icsm* (p. 34). IEEE.

[52] Zhang, R., Jiang, J., Yin, J., Jin, A., Lou, J., & Wu, Y. (2008, November). A new method for test suite reduction. In *The 9th International Conference for Young Computer Scientists* (pp. 1211-1216). IEEE.

[53] Parsa, S., & Khalilian, A. (2010). On the optimization approach towards test suite minimization. *International Journal of Software Engineering and its applications*, *4*(1), 15-28.

[54] Selvakumar, S., Ramaraj, N., Naaghumeenal, R. M., & Nandini, R. (2011, March). Computing with Enhanced Test Suite Reduction Heuristic. In *International Conference on Advances in Communication, Network, and Computing* (pp. 504-507). Springer, Berlin, Heidelberg.

[55] Xu, S., Miao, H., & Gao, H. (2012, August). Test suite reduction using weighted set covering techniques. In *Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on* (pp. 307-312). IEEE.

[56] Ould, M. A. (1991). Testing-a challenge to method and tool developers. *Software Engineering Journal*, *6*(2), 59-64.

[57] Luo, L. (2001). Software testing techniques: technology maturation and research strategy. *Class report for*.

[58] Badhera, U., Purohit, G. N., & Biswas, D. (2012). Test case prioritization algorithm based upon modified code coverage in regression testing. *International Journal of Software Engineering & Applications*, *3*(6), 29.

[59] Jiang, B., Zhang, Z., Chan, W. K., & Tse, T. H. (2009, November). Adaptive random test case prioritization. In *Proceedings of the 2009 IEEE/ACM*

International Conference on Automated Software Engineering (pp. 233-244). IEEE Computer Society.

[60] Kaur, A., & Goyal, S. (2011). A genetic algorithm for fault based regression test case prioritization. *International Journal of Computer Applications*, *32*(8), 975-8887.

[61] Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., & Rothermel, G. (2012). A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, *38*(6), 1258-1275.

[62] Wong, W. E., Horgan, J. R., Mathur, A. P., & Pasquini, A. (1997, August). Test set size minimization and fault detection effectiveness: A case study in a space application. In *Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International* (pp. 522-528). IEEE.

[63] Elbaum, S., Kallakuri, P., Malishevsky, A., Rothermel, G., & Kanduri, S. (2003). Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software testing, verification and reliability*, *13*(2), 65-83.

[64] Di Nardo, D., Alshahwan, N., Briand, L., & Labiche, Y. (2015). Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability*, *25*(4), 371-396.

[65] Andrews, J. H., Briand, L. C., Labiche, Y., & Namin, A. S. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, (8), 608-624.

[66] Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994, May). Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering* (pp. 191-200). IEEE Computer Society Press.

[67] Simao, A. D. S., De Mello, R. F., & Senger, L. J. (2006, September). A technique to reduce the test case suites for regression testing based on a self-organizing neural network architecture. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International* (Vol. 2, pp. 93-96). IEEE.

[68] Kovács, G., Németh, G. Á., Subramaniam, M., & Pap, Z. (2009, September). Optimal string edit distance based test suite reduction for SDL specifications. In *International SDL Forum* (pp. 82-97). Springer, Berlin, Heidelberg.

[69] Chen, T. Y., Kuo, F. C., Merkel, R. G., & Tse, T. H. (2010). Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, *83*(1), 60-66.

[70] Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2000). *Prioritizing test cases for regression testing* (Vol. 25, No. 5, pp. 102-112). ACM.

[71] Coutinho, A. E. V. B., Cartaxo, E. G., Machado, P. D., & SPLab-UFCG, C. G. (2013). Test suite reduction based on similarity of test cases. In *7st Brazilian workshop on systematic and automated software testing—CBSoft*.

[72] Hemmati, H., Arcuri, A., & Briand, L. (2013). Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *22*(1), 6.

[73] Sharma, C., & Singh, S. (2014, May). Mechanism for identification of duplicate test cases. In *Recent Advances and Innovations in Engineering (ICRAIE), 2014* (pp. 1-5). IEEE.

[74] Singh, S., Sharma, C., & Singh, U. (2013). A simple technique to find diverse test cases. In *IJCA Proceedings on 9th International ICST Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness (QShine-2013)* (pp. 1-4).

[75] Wong, W. E., Horgan, J. R., Mathur, A. P., & Pasquini, A. (1997, August). Test set size minimization and fault detection effectiveness: A case study in a space application. In *Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International* (pp. 522-528). IEEE.

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

[77] Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, *27*(10), 929-948.

[78] Srikanth, H., Williams, L., & Osborne, J. (2005, November). System test case prioritization of new and regression test cases. In *Empirical Software Engineering, 2005. 2005 International Symposium on* (pp. 10-pp). IEEE.

[79] Hou, S. S., Zhang, L., Xie, T., & Sun, J. S. (2008, September). Quota-constrained test-case prioritization for regression testing of service-centric systems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on* (pp. 257-266). IEEE.

[80] Do, H., & Rothermel, G. (2006). On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, *32*(9), 733-752.

[81] Do, H., Rothermel, G., & Kinneer, A. (2004, November). Empirical studies of test case prioritization in a JUnit testing environment. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on* (pp. 113-124). IEEE.

[82] Do, H., Rothermel, G., & Kinneer, A. (2006). Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering*, *11*(1), 33-70.

[83] Jones, J. A., & Harrold, M. J. (2003). Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering*, *29*(3), 195-209.

[84] Leon, D., & Podgurski, A. (2003, November). A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *null* (p. 442). IEEE.

[85] Aggrawal, K. K., Singh, Y., & Kaur, A. (2004). Code coverage based technique for prioritizing test cases for regression testing. *ACM SIGSOFT Software Engineering Notes*, *29*(5), 1-4.

[86] Bryce, R. C., & Colbourn, C. J. (2005, May). Test prioritization for pairwise interaction coverage. In *ACM SIGSOFT Software Engineering Notes* (Vol. 30, No. 4, pp. 1-7). ACM.

[87] Srikanth, H., Cohen, M. B., & Qu, X. (2009, November). Reducing field failures in system configurable software: Cost-based prioritization. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on* (pp. 61-70). IEEE.

[88] Jeffrey, D., & Gupta, N. (2006, September). Test case prioritization using relevant slices. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International* (Vol. 1, pp. 411-420). IEEE.

[89] Sampath, S., Bryce, R. C., Viswanath, G., Kandimalla, V., & Koru, A. G. (2008, April). Prioritizing user-session-based test cases for web applications testing. In *Software Testing, Verification, and Validation, 2008 1st International Conference on* (pp. 141-150). IEEE.

[90] Carlson, R., Do, H., & Denton, A. (2011, September). A clustering approach to improving test case prioritization: An industrial case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on* (pp. 382-391). IEEE.

[91] Coutinho, A. E. V. B., Cartaxo, E. G., & de Lima Machado, P. D. (2016). Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal*, *24*(2), 407-445.

[92] Yoo, S., Harman, M., Tonella, P., & Susi, A. (2009, July). Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (pp. 201-212). ACM.

[93] Jiang, B., Zhang, Z., Chan, W. K., & Tse, T. H. (2009, November). Adaptive random test case prioritization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (pp. 233-244). IEEE Computer Society.

[94] Ledru, Y., Petrenko, A., Boroday, S., & Mandran, N. (2012). Prioritizing test cases with string distances. *Automated Software Engineering*, *19*(1), 65-95.

[95] Fang, C., Chen, Z., Wu, K., & Zhao, Z. (2014). Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, *22*(2), 335-361.

[96] Wang, R., Jiang, S., Chen, D., & Zhang, Y. (2016). Empirical study of the effects of different similarity measures on test case prioritization. *Mathematical Problems in Engineering*, *2016*.

[97] Wang, R., Jiang, S., Chen, D., & Zhang, Y. (2016). Empirical study of the effects of different similarity measures on test case prioritization. *Mathematical Problems in Engineering*, *2016*.

[98] Hemmati, H. (2015, August). How effective are code coverage criteria?. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on* (pp. 151-156). IEEE.

[99] Ramler, R., Biffl, S., & Grünbacher, P. (2006). Value-based management of software testing. In *Value-based software engineering* (pp. 225-244). Springer, Berlin, Heidelberg.

[100] Boehm, B., & Huang, L. G. (2003). Value-based software engineering: A case study. *Computer*, *36*(3), 33-41.

[101] Zhang, L., Hou, S. S., Guo, C., Xie, T., & Mei, H. (2009, July). Time-aware test-case prioritization using integer linear programming. In *Proceedings of the eighteenth international symposium on Software testing and analysis* (pp. 213-224). ACM.

[102] Ashraf, E., Rauf, A., & Mahmood, K. (2012). Value based regression test case prioritization. In *Proceedings of the world congress on engineering and computer science* (Vol. 1, pp. 24-26).

[103] Reeves, C. R. (1995). *Modern heuristic techniques for combinatorial problems. Advanced topics in computer science*. Mc Graw-Hill.

[104] Yoo, S., & Harman, M. (2007, July). Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis* (pp. 140-150). ACM.

[105] Huang, Y. C., Huang, C. Y., Chang, J. R., & Chen, T. Y. (2010, July). Design and analysis of cost-cognizant test case prioritization using genetic algorithm with test history. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual* (pp. 413-418). IEEE.

[106] Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2004, July). Where the bugs are. In *ACM SIGSOFT Software Engineering Notes* (Vol. 29, No. 4, pp. 86-96). ACM.

[107] Ahmed, A. (2009). *Software testing as a service*. Auerbach Publications.

[108] Srivastava, P. R. (2008). TEST CASE PRIORITIZATION. *Journal of Theoretical & Applied Information Technology*, *4*(3).

[109] Raamesh, L., & Uma, G. V. (2010). Reliable mining of automatically generated test cases from software requirements specification (SRS). *arXiv preprint arXiv:1002.1199*.

[110] Rothermel, G., & Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(2), 173-210.

[111] Chvatal, V. (1979). A greedy heuristic for the set-covering problem. *Mathematics of operations research*, *4*(3), 233-235.

[112] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.

[113] Chen, T. Y., & Lau, M. F. (1998). A simulation study on some heuristics for test suite reduction. *Information and Software Technology*, *40*(13), 777-787.

[114] Chen, T. Y., & Lau, M. F. (1998). A new heuristic for test suite reduction. *Information and Software Technology*, *40*(5-6), 347-354.

[115] de Mello, R. F., & Senger, L. J. (2006, June). Model for simulation of heterogeneous high-performance computing environments. In *International Conference on High Performance Computing for Computational Science* (pp. 107-119). Springer, Berlin, Heidelberg.

[116] Chae, H. S., Woo, G., Kim, T. Y., Bae, J. H., & Kim, W. Y. (2011). An automated approach to reducing test suites for testing retargeted C compilers for embedded systems. *Journal of Systems and Software*, *84*(12), 2053-2064.

[117] Santelices, R., Jones, J. A., Yu, Y., & Harrold, M. J. (2009, May). Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 56-66). IEEE Computer Society.

[118] Hemmati, H., & Briand, L. (2010, November). An industrial investigation of similarity measures for model-based test case selection. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on* (pp. 141-150). IEEE.

[119] Singh, S., & Shree, R. (2016). A combined approach to optimize the test suite size in regression testing. *CSI transactions on ICT*, *4*(2-4), 73-78.

[120] Singh, S., & Shree, R. (2017). A Multi Criteria based Test Suite Optimization Framework. International Journal of Software Engineering and Its Applications (SERSC), 11 (1), 77-86.

Appendix A: Abbreviations

- **AHP-** Analytic Hierarchy Process
- **APFD** - Average Percentage of Faults Detected
- **ART**- Adaptive Random Testing
- **CS** – Coverage Similarity
- **FDE** – Fault Detection Effectiveness
- **FDL** – Fault Detection Loss
- **GRE** – Greedy Redundant Essential
- **HGS -** Harrold-Gupta-Soffa
- **ICP-** Interleaved Clusters Prioritisation
- **ILP** - Integer Linear Programming
- **LOC –** Lines of Code
- **MBT**- Model-based Testing
- **MC/DC –** Modified Condition/Decision Condition
- **SB_TSO** – Similarity Based Test Suite Optimization
- **SBGA** – Similarity based Greedy Algorithm
- **SSR** – Suite Size Reduction
- **SUT** – Software Under Test
- **TCP** – Test Case Prioritization

- **TCS** – Test Case Selection
- **TSM** – Test Suite Minimization
- **TSO -** Test Suite Optimization

Appendix B: Plagiarism Report

URKUND

Urkund Analysis Result

Analysed Document:	my thesis_Final.pdf (D40560525)
Submitted:	7/12/2018 7:01:00 AM
Submitted By:	gbl.bbau@gmail.com
Significance:	3 %

Sources included in the report:

https://aaltodoc.aalto.fi/handle/123456789/28522
http://www.jatit.org/volumes/research-papers/Vol18No2/5Vol18No2.pdf
https://www.ripublication.com/ijaer17/ijaerv12n9_22.pdf
https://www.ijedr.org/papers/IJEDR1603064.pdf
http://cse.unl.edu/~grother/papers/icsm99.pdf
https://pdfs.semanticscholar.org/e489/7d612d0da21af26355be5ff05f12d617a494.pdf
http://www.ijates.com/images/short_pdf/1447076778_169D.pdf
http://www.apjcri.org/papers/v1n2/2.pdf
https://www.slideshare.net/kanoahinc/test-ccase-prioritization-techniques
https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/PASTE05.pdf
https://link.springer.com/chapter/10.1007%252F978-3-642-10509-8_24
https://www.ukessays.com/essays/computer-science/test-case-prioritization-and-regression-test-selection.php
https://jserd.springeropen.com/articles/10.1186/s40411-017-0045-x
http://cse.unl.edu/~grother/papers/tr03-60-04.pdf
https://www.researchgate.net/publication/289884518_A_Tool_for_Generation_and_Minimization_of_Test_Suite_by_Mutant_Gene_Algorithm
https://www.researchgate.net/publication/261267276_Reducing_the_Cost_of_Regression_Testing_by_Identifying_Irreplaceable_Test_Cases
https://www.cc.gatech.edu/~orso/papers/hsu.orso.mints-tr.pdf
https://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1017&context=csearticles

Instances where selected sources appear:

34

Appendix C: List of Publications

Papers Published:

1. Shilpi Singh and Raj Shree, "**A New Similarity-based Greedy Approach for Generating Effective Test Suite**", International Journal of Intelligent Engineering and Systems (IJIES), **Scopus, UGC Indexed. (Accepted) (In-Press)**
2. Shilpi Singh and Raj Shree "**A combined approach to optimize the test suite size in regression testing**", CSI transactions on ICT, 2016, 1; 4 (2-4):73-8. **Springer, UGC Indexed.**
3. Shilpi Singh and Raj Shree, "Different Similarity Measures for Test Suite Optimization", Advanced Science, Engineering and Medicine, Vol. 10, 1–4, 2018, **Scopus, UGC Indexed. (Accepted) (In-Press)**
4. Shilpi Singh and Raj Shree "**A Multi Criteria based Test Suite Optimization Framework**", International Journal of Software Engineering and Its Applications (SERSC). 2017 , 11 (1), pp. 77-86
5. Singh, Shilpi, and Raj Shree. "**An Analysis of Test Suite Minimization Techniques**" International Journal of Engineering Sciences and Research Technology 5 (2016): 252-260, **UGC Indexed**
6. Shilpi Singh and Raj Shree, "**Hyper-Heuristic Test Suite Optimization**", In Proceedings of the IEEE International Conference on Advanced Computing and Software Engineering (ICACSE-16). (**International IEEE Conference)**
7. Shilpi Singh and Raj Shree, "**Pair-Wise Selection Approach for Test Case Prioritization in Regression Testing**", In Proceedings of the 3rd International Conference on Computers and Management (ICCM 2017) , Vol. 1, 29–35, 2018. (**International Conference)**

8. Shilpi Singh and Raj Shree, "**A Systematic Literature Review of Test Case Prioritization Techniques**", In Proceedings of the National Conference on Information Security Challenges (NCISC-2016), BBAU, Lucknow, 24th Feb 2016, **(National Conference)**

Patent Published:

TITLE OF THE INVENTION: SYSTEM AND METHOD FOR TEST SUITE OPTIMIZATION IN REGRESSION TESTING

APPLICATION NO: 201711035077 A

PUBLICATION DATE: 24/11/2017

NAME OF INVENTOR: RAJ SHREE, SHILPI SINGH, RAVI PRAKASH PANDEY