Time-Aware Test Case Prioritization using Binary Integer Programming

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

of

Masters of Technology

in

Computer Science and Applications

Submitted By
Sunil
(Roll No. 601003028)

Under the supervision of:
Dr. Rajesh Kumar
Associate Professor

School of Mathematics and Computer Applications
Thapar University,
Patiala –147004.

June 2012

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Time-Aware Test Case Prioritization using Binary Integer Programming"**, in partial fulfilment of the requirements for the award of degree of Master of technology in **Computer Science and Applications** submitted in School of Mathematics and Computer Applications of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Rajesh Kumar** and refers other researcher's work which are duly listed in the reference section.

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

(Sunil)

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

(Dr. Rajesh Kumar)

Associate Professor
School of Mathematics and Computer Application

Countersigned by

(Dr. S. S. Bhatia)

Head
School of Mathematics and Computer Applications
Thapar University
Patiala

(Dr. S. K. Mohapatra)

Dean Academic Affairs
Thapar University
Patiala

Acknowledgement

I am highly grateful to the authorities of Thapar University, Patiala for providing this opportunity to carry out the thesis work. I would like to express a deep sense of gratitude and thanks profusely to my thesis guide Dr, Rajesh Kumar Associate Professor, School of Mathematics and Computer Application, Thapar University, Patiala, for his sincere and invaluable guidance, suggestion and sympathetic attitude which inspired me to submit this Thesis report in the present form. I am thankful to other faculty members of School of Mathematics and Computer Application, Thapar University, Patiala, for their intellectual support. My special thanks are due to Mr.Manoj Kumar Pachariya (Assistant Professor, *Galgotias University* Noida,) for his valuable suggestion from time to time. Special thank are due to my friend Vikas who constantly helped and encouraged me throughout the process of this thesis.

Abstract

Software testing is error prone, time consuming, expensive and core activity for quality assurance. Test cases selection, prioritization, minimization and filteration forms common thread of optimization. Quality of software testing is low due to inadequate techniques for test cases optimization. In such cases, due to time and cost constraints, the entire test suite cannot be run. Thus, it becomes essential to prioritize the test cases in order to cover maximum number faults/errors in minimum time. Binary Integer Programming (BIP) is a mathematical optimization technique that has been applied for time aware test case prioritization problem considering the faults coverage and execution time of each test case. This work presents the regression test case prioritization technique that reorders test cases in a test suite using fault coverage criteria within a time constraint environment. Simulation using MATLAB has also been done, which gives optimal solutions in optimum time.

Certificate
Acknowledgement
Abstract

Contents

Chapter1

Introduction

Software testing is one of the vital activity which is essential for quality assurance. Software testing consist of *selection* of test case, execution of test case and verifying the result. Here, selection of test case and result verification are extensive activities and require improvement by appropriate test case.

1.1 Software Testing

IEEE definition of software testing is process of executing the program with specific intent of finding errors. Software testing remains the primary concern used to gain consumer's confidence in the software. Ideally, testing of software guarantees the absence of errors in the software, but in reality it only discloses the presence of software errors, but never guarantees their absence. Even, systematic testing cannot guarantee the absence of errors, which are detected by discovering their effect.

In this chapter, we will provide introduction to various basic principle and key terms which are used in software testing.

Software testing is a process of verifying and validating that a software application or program. it meets the business and technical requirements that guided its design and development, and Works as expected.

Software testing also identifies important defects, flaws, or errors in the application code that must be fixed. The modifier "important" in the previous sentence is, well, important because defects must be categorized by severity.

During test planning, we decide what an important defect is by reviewing the requirements and design documents with an eye towards answering the question "Important to whom?" Generally speaking, an important defect is one that from the customer's perspective affects the usability or functionality of the application. Using colors for a traffic lighting scheme in a desktop dashboard may be a no-brainer during requirements definition and easily implemented during development but, in fact, may not be entirely workable if during testing, we discover that the primary business sponsor is color blind. Suddenly, it becomes an important defect.

The quality assurance aspect of software development documenting the degree to which the developers followed corporate standard processes or best practices is not addressed in this thesis because assuring quality is not a responsibility of the testing team but it also depend on testing strategy . The testing team cannot guarantee quality; they can only measure it, although it can be argued that doing things like designing tests before coding begins, will improve quality because the coders can then use that information while thinking about their designs and during coding and debugging.

Software testing has three main purposes: verification, validation, and defect finding.

- The verification process confirms that the software meets its technical specifications. A specification is a description of a function in terms of a measurable output value given a specific input value under specific preconditions.
- The validation process confirms that the software meets the business requirements. A defect is a variance between the expected and actual result.
- The defect's ultimate source may be traced to a fault introduced in the specification, design, or development (coding) phases.

1.2 Testing Principles

Following are principles of Software Testing [1]:

- All tests should be traceable to customer requirements. The objective of system testing is to uncover errors. It follows that the most severe defects are those that cause the program to fail to meet its requirements.
- Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
- Testing should begin "in the small" and progress toward testing "in the large". The first test planned and executed generally focus on individual program modules. As testing progresses, testing shifts focus in an attempt to find errors in integrated clusters of modules and ultimately in the entire system.

- Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing.

- To be most effective, testing should be conducted by an independent third party. By most effective, means testing that has the highest probability of finding errors. For this reason, the software engineer who created the system is not the best person to conduct all tests for the software.

1.3 Characteristics of a "efficient" Test case

Following are the characteristics of a efficient test [1]:

- An efficient test case has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.

- An efficient test case is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose.

- An efficient test case should be "best of breed". In a group of tests that have a similar intent, time and resource limitations may militate for the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

- An efficient test case should be neither too simple nor too complex . Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

1.4 Testing Process

The IEEE 829 standard [2] describes a framework within which the entire testing process can be managed .The framework allows easy communication between members of a testing project, organizes the testing process, and outlines the documents that should be made part of any compliant testing process.

1.4.1 Test Plan

Test plan describes the scope, approach, resources, and schedule of testing activities in a given project, and identifies the items to be tested, the features of those items to be tested, the individual testing tasks that are to be performed, and personnel responsible for those tasks, along with the risks associated with the plan.

Test plan should have the following structure:

- Test plan identifier
- Introduction
- Items to be tested
- Features to be tested
- Features not to be tested
- Testing approach
- Test item pass and fail criteria
- Test suspension criteria
- Test resumption requirements
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing and training requirements
- Schedule
- Risks and contingencies
- Approvals

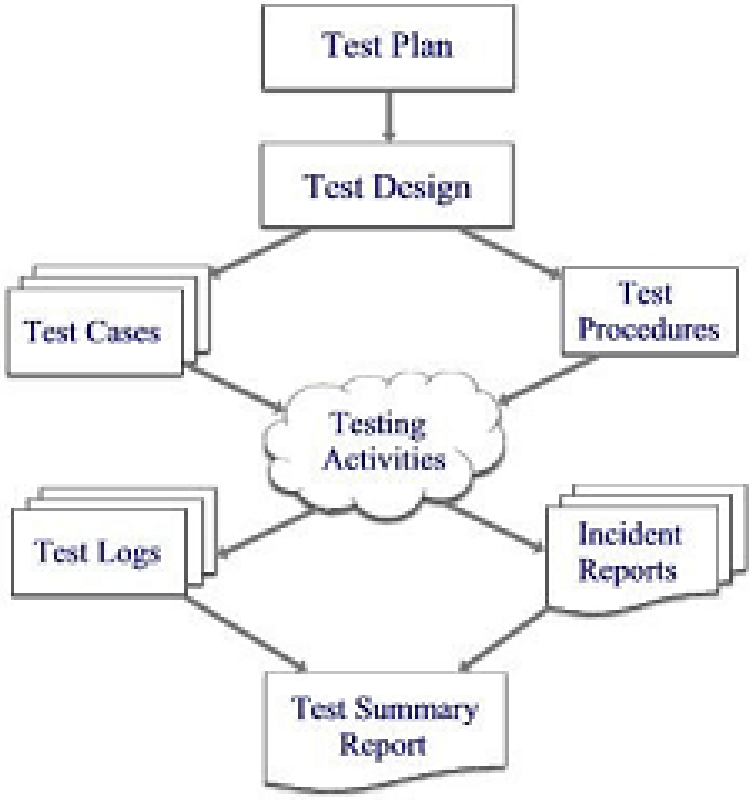

Fig. 1.1: Testing activities

1.4.2 Test Design

Test design, or test specification as it is sometimes known, further refines the testing approach, and identifies the test cases and test procedures, and test item pass and fail criteria.

Test design specification should have the following structure:

- Test design identifier
- Features to be tested
- Approach refinements
- Test identification
- Pass and fail criteria

1.4.3 Test Cases

Individual test cases document the values that will be used as input to individual tests, Together with the associated expected outputs. A test case identifies any constraints on the test procedure resulting from the use of the test case, and separated from the test design to allow easy reuse in other situations.

Test cases should have the following structure:

- Test case identifier
- Items to be tested
- Input specifications
- Expected output specifications
- Environmental prerequisites
- Special procedural requirements
- Inter-case dependencies

1.4.4 Test Procedure

Test procedure describes the exact steps required to operate the system and execute test cases in order to implement the test design. Test procedure is kept separate from the test design as it is followed step by step, and does not contain irrelevant detail.

Test procedure should have the following structure:

- Test procedure identifier
- Purpose
- Special requirements
- Procedure steps

1.4.5 Test Logs

Test logs are used to record what occurred during execution of a test or set of tests. Test logs may either be manually created as tests are executed, or automatically by the system as testing processes.

Test logs should have the following structure:

- Test log identifier
- Description
- Activity and event entries

Incident reports are used to provide a description of any events that occur during testing that require further investigation.

Incident reports should have the following structure:

- Incident report identifier
- Summary
- Incident description
- Impact of the incident

1.4.6 Test Summary Report

Test summary report summarizes the testing activities associated with one or more test designs.

Test summary report should have the following structure:

- Test summary report identifier
- Summary
- Variances
- Comprehensive assessment
- Summary of results
- Evaluation

- Summary of activities
- Approvals

1.5 Level of Testing

Testing is involved in every stage of software life cycle, but the testing done at each level of software development is different in nature and has different objectives.

Unit Testing: is done at the lowest level. It tests the basic unit of software, which is the smallest testable piece of software, and is often called “unit”, “module”, or “component” interchangeably.

Integration Testing: is performed when two or more tested units are combined into a larger structure. The test is often done on both the interfaces between the components and the larger structure being constructed, if its quality property cannot be assessed from its components.

System Testing: tends to affirm the end-to-end quality of the entire system. System test is often based on the functional requirement specification of the system. Non-functional quality attributes, such as reliability, security, and maintainability, are also checked.

Acceptance Testing: is done when the completed system is handed over from the developers to the customers or users. The purpose of acceptance testing is rather to give confidence that the system is working than to find errors.

Regression Testing: is any type of software testing that seeks to uncover new software bugs, or regressions, in existing functional and non-functional areas of a system after changes, such as enhancements, patches or configuration changes, have been made to them.

The intent of regression testing is to ensure that a change, such as a bug fix, did not introduce new faults. One of the main reasons for regression testing is to determine whether a change in one part of the software affects other parts of the software.

Common methods of regression testing include re-running previously run tests and checking whether program behavior has changed and whether previously fixed faults have re-emerged. Regression testing can be used to test a system efficiently by systematically selecting the appropriate minimum set of tests needed to adequately cover a particular change.

1.6 Types of Testing

Software testing methods are traditionally divided into white and black box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases**.**

1.6.1 White Box Testing

White box testing (also known as clear box testing, glass box testing, transparent box testing and structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality .In white box testing, an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in circuit testing (ICT).

While white box testing can be applied at the unit, integration and system levels of the software testing process, it is usually done at the unit level. It can test paths within a unit, paths between units during integration, and between subsystems during a system level test. Though this, method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

White box test design techniques include, control flow testing, data flow testing, branch testing, path testing, statement coverage and decision coverage.

1.6.2 Black Box Testing

Black box testing is a method of software testing that tests the functionality of an application as opposed to its internal structures or workings. This method of test can be applied to all levels of software testing: unit, integration, system and acceptance. It typically comprises most if not all testing at higher levels, but can also dominate unit testing as well.

Specific knowledge of the application's code and internal structure and programming knowledge in general is not required. The tester is only aware of what the software is supposed to do but not how i.e. when he enters a certain input, he gets a certain output without being aware of how the output was produced in the first place.

1.7 Test Case Design Techniques

There are mainly six techniques to design test case. They are as follows:

Use Case: In software and systems engineering, a use case is a list of steps, typically defining interactions between a role (known in UML as an "actor") and a system, to achieve a goal. The actor can be a human or an external system**.**

Decision Tables: Decision tables are a precise yet compact way to model complicated logic. Decision tables, like flowcharts and if-then-else and switch-case statements, associate conditions with actions to perform, but in many cases do so in a more elegant way.

All Pair Shortest Path: All-pairs testing or pair wise testing is a combinatorial software testing method that, for each pair of input parameters to a system (typically, a software algorithm), tests all possible discrete combinations of those parameters. Using carefully chosen test vectors, this can be done much faster than an exhaustive search of all combinations of all parameters, by parallelizing the tests of parameter pairs. The number of tests is typically O(nm), where n and m are the number of possibilities for each of the two parameters with the most choices.

State Transition Table: In automata theory and sequential logic, a state transition table is a table showing what state (or states in the case of a nondeterministic finite automation) a finite semi automation or finite state machine will move to, based on the current state and other inputs. A state table is essentially a truth table in which some of the inputs are the current state, and the outputs include the next state, along with other outputs. A state table is one of many ways to specify a state machine, other ways being a state diagram, and a characteristic equation.

Equivalence Partitioning: Equivalence partitioning (also called Equivalence Class Partitioning or ECP) is a software testing technique that divides the input data of a software unit into partitions of data from which test cases can be derived. In principle, test cases are designed to cover each partition at least once. This technique tries to define test cases that uncover classes of errors, thereby reducing the total number of test cases that must be developed.

Boundary Value Analysis: Boundary value analysis is a software testing technique in which tests are designed to include representatives of boundary values. Values on the minimum and

maximum edges of an equivalence partition are tested. The values could be either input or output ranges of a software component. Since these boundaries are common locations for errors that result in software faults they are frequently exercised in test cases.

1.8 Test Case Adequacy Criteria

Measurement of structural coverage of code is a means of assessing the thoroughness of testing. There are a number of metrics available for measuring structural coverage, with increasing support from software tools. Such metrics do not constitute testing techniques, but a measure of the effectiveness of testing techniques. Coverage metric is expressed in terms of a ratio of the metric items executed or evaluated at least once to the total number of metric items.

Statement Coverage

Statement Coverage = s/S

where:

s = Number of statements executed at least once.

S = Total number of executable statements.

Statement coverage is the simplest structural coverage metric. From a measurement point of view one just keeps track of which statements are executed, then compares this to a list of all executable statements. Statement coverage is therefore suitable for automation.

Decision coverage

Decision coverage = d/D

where:

d = Number of decision outcomes evaluated at least once.

D = Total number of decision outcomes.

To achieve 100% decision coverage, each condition controlling branching of the code has to evaluate to both true and false.

LCSAJ Coverage: Linear code sequence and jump.

An LCSAJ is defined as an unbroken linear sequence of statements:

(a) Which begins at either the start of the program or a point to which the control flow may jump.

(b) Which ends at either the end of the program or a point from which the control flow may jump.

(c) And the point to which a jump is made following the sequence.

LCSAJ coverage = l/L

where:

l = Number of LCSAJs exercised at least once.

L = Total number of LCSAJs.

LCSAJs depend on the topology of a module's design and not just its semantics. the do not map onto code structures such as branches and loops. LCSAJs are not easily identifiable from design documentation. They can only be identified once code has already been written. LCSAJs are not easily comprehensible.

Path Coverage

Path Coverage = p/P

where:

p = Number of paths executed at least once.

P = Total number of paths.

Path coverage looks at complete paths through a program. For example, if a module contains a loop, then there are separate paths through the module for one iteration of the loop, two iterations of the loop, and to n iterations of the loop. The thoroughness of test data designed to achieve 100% path coverage is higher than that for decision coverage.

Condition Operand Coverage

Condition Operand Coverage = c/C

where:

c = Number of condition operand values evaluated at least once.

C = Total number of condition operand values.

Condition operand coverage gives a measure of coverage of the conditions which could cause a branch to be executed. Condition operands can be readily identified from both design and code, with condition operand coverage directly related to the operands. This facilitates automation and makes condition operand coverage both comprehensible and maintainable.

Condition Operator Coverage

Condition Operator Coverage = o/O

where:

o = Number of condition combinations evaluated at least once.

O = Total number of condition operator input combinations.

Condition operator coverage looks at the various combinations of Boolean operands within a condition. Each Boolean operator (and, or, xor) within a condition has to be evaluated four times, with the operands taking each possible pair of combinations of true and false.

Boolean Operand Effectiveness Coverage

Boolean Operand Effectiveness Coverage = b/B

where:

b = Number of boolean operands shown to independently influence the outcome of boolean expressions.

B = Total number of boolean operands.

To achieve boolean operand effectiveness coverage, each boolean operand must be able to independently influence the outcome of the overall boolean expression. The straight forward relationship between test data and the criteria of boolean operand effectiveness coverage makes the metric comprehensible and associated test data maintainable.

1.9 Test Case Prioritization

In this technique test cases are assigned a priority. Priority is set according to some criterion and test cases with highest priority are scheduled first. For example, criterion may be that the test case which has faster code coverage gets the highest priority. Advantage to previous techniques is that it doesn't discard or permanently remove the test cases from test suite. Another criterion may be rate at which fault is detected.

Test case prioritization schedule test cases in order to increase their ability to meet some performance goal:

- Rate of fault detection
- Rate of code coverage
- Rate of increase of confidence in reliability

1.10 Organization of thesis

This thesis is organized as follows:

Chapter 2: This chapter describes in detail the literature survey on related works in the field of test case prioritization**.**

Chapter 3: In this chapter gap and objectives statement of the problem is given which has derived from the literature survey.

Chapter 4: This chapter gives detail of our proposed solution "Test Prioritization Using Binary Integer Programming".

Chapter 5: In this chapter results have been presented which is derived from implementation of proposed solution.

Chapter 6: This chapter describes the conclusion and future research wor

Chapter 2

Literature survey

In this chapter we have surveyed related works in different types Test case prioritization techniques of and mapped them to our taxonomy to guide future design and development efforts.

Test case prioritization techniques provide a way to schedule and run test cases, which have the highest priority in order to provide earlier detect faults. This study presents numerous techniques developed, that can improve a test suite's rate of fault detection.

2.1 Test Case Prioritization Techniques

In this section various types of Test case prioritization techniques along with related works has been discussed.

2.1.1 Customer Requirement-Based Prioritization Techniques

Customer requirement-based techniques are methods to prioritize test cases based on requirement documents. Many researchers have researched this area and introduced many weight factors which used in these techniques, including custom-priority, requirement complexity and requirement volatility.

Hema et al. [3] presented the requirements-based test case prioritization approach to prioritize a set of test cases. Amitabh [4] built upon current test case prioritization techniques and proposed to use several factors to weight (or rank) the test cases. Those factors are the customer-assigned priority (CP), requirements complexity (RC) and requirements volatility (RV). Additionally, they assigned value (1 to 10) to each factor for the measurement. They stated that higher factor values indicate a need for prioritization of test case related to that requirement.

Weight prioritization (WP) is measured as.

$WP = \Sigma(PF_{value} * PF_{weight})$; PF=1 to n

where:

- WP denotes weight prioritization that measures the importance of testing a requirement.
- PF_{value} is the value of each factor, like CP, RC and RV.
- PF_{weight} is the weight of each factor, like CP, RC and RV.

Test cases are then ordered such that the test cases with high WP are executed before others.

Manish et al. [5] proposed an approach for test case generation for web based applications. They presented a simple approach for test case prioritization through the requirement traceability matrix. The matrix can be produced by mapping from use cases in the use case diagram to functional requirements from users. They also proposed to use weight values assigned to each requirement by developers. Each requirement is assigned a priority weight from 1 to 10, 10 being highest.

2.1.2 Coverage-Based Prioritization Techniques

Coverage-based techniques are methods to prioritize test cases based on coverage criteria, such as requirement coverage, total requirement coverage, additional requirement coverage and statement coverage.

Test coverage analysis is a measure used in software testing known as code coverage analysis for practitioners. It describes the quantity of source code of a program that has been exercised during testing. It is a form of testing that inspects the code directly and is therefore a form of white box testing.

The following lists a process of coverage-based techniques:

(a) Finding areas of a program not exercised by a set of test cases

(b) Creating additional test cases to increase coverage

(c) Determining a quantitative measure of code coverage, which is an indirect measure of quality.

(d) Identifying redundant test cases that do not increase coverage.

The coverage-based technique is a structural or white-box testing technique. Structural testing compares test program behavior against the apparent intention of the source code. This

contrasts with functional or black-box testing, which compares test program behavior against a requirements specification. It examines how the program works, taking into account possible pitfalls in the structure and logic. Functional testing examines what the program accomplishes, without regard to how it works internally. The coverage-based techniques are methods to prioritize test cases based on coverage criteria, such as requirement coverage, total requirement coverage, additional requirement coverage and statement coverage.

Leon and Podgurski [6] presented an empirical comparison of four different techniques for filtering large test suites: 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 that maximize code coverage as quickly as possible, while the latter two are based on analyzing the distribution of the test's execution profiles.

Rothermel et al [7] have researched and surveyed test case prioritization. They considered nine approaches for prioritizing a set of test cases and reported results measuring the effectiveness of those approaches to improve the capability to reveal faults.

Following are the techniques proposed by them

(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.

(h) Additional fault-exposing-potential prioritization.

Bryce [8] described an algorithm for re-generating prioritized test suites. The generated test suites are a special kind of a covering array called a biased covering array. They began by defining a set of interaction weights for each value of each factor. For each factor the weight of combining it with each other factor is computed as a total interaction benefit. The factors are sorted in decreasing order of interaction benefit and then filled as follows. First, the

individual interaction weights for each of the factor's values are computed. This selects the value of the factor that has the greatest value interaction benefit. After all factors have been fixed, a single test is added, and the benefits for factors are recomputed and the process starts again. The algorithm is complete when all pairs have been covered.

Leon and Andy [9] believed that test case filtering is closely related to the field of test case prioritization. The goal of test case filtering is to select a relatively small subset of a test suite which finds a large portion of the defects that would be found if the whole test suite were to be used. In this paper, they presented an empirical comparison of four different techniques for filtering large test suites: test suite minimization, prioritization by additional coverage, cluster filtering with one per-cluster sampling and failure pursuit sampling. Their results indicate that their techniques can be as efficient as or more efficient at revealing defects than coverage-based techniques, but that the two kinds of techniques are also complementary in the sense that they find different defects. Accordingly, some simple combinations of these techniques were evaluated for use in test case prioritization. The results indicate that applying this combination of techniques can produce results more efficiently than applying prioritization by additional coverage alone.

Xiao et al. [10] has examined the prioritization methods of Combination Integration Testing (CIT) test suites and developed several ways to control the prioritization through weightages. They used methods that utilize code coverage data from prior releases, as well as one that is specification based. Further, they observed that Bryce and Colbourn's prioritization technique [11] is a combination of the generation and prioritization techniques, rather than a pure prioritization method. This is because it regenerates tests each time rather than simply reordering them.

Hyunsook Do and Gregg Rothermel [12] considered seven different test case prioritization techniques, which they classified into three groups:

(a) The first group is the control group, containing three *"orderings"* that serve as experimental controls. The untreated ordering is the ordering in which test cases are originally provided with the object. The optimal ordering represents an upper bound on prioritization technique performance and is obtained by greedily selecting each test case in terms of its exposure of faults not yet exposed by test cases already ordered. The process is repeated until all test cases are ordered. Ties are broken randomly. The random ordering places test cases in a random order.

(b) The second group is the block level group, containing two techniques: block total and block addtl. By instrumenting a program, they can determine the numbers of basic blocks in that program that are exercised by that test case. The block-total technique prioritizes test cases according to the total number of blocks they cover simply by sorting them by that number. The block addtl technique prioritizes test cases in terms of the numbers of additional blocks test cases cover by greedily selecting the test cases that cover the most as-yet-uncovered blocks until all blocks are covered, then repeating this process until all test cases have been placed in order and (c) the third group is the method level group, containing two techniques: method total and method-addtl. These techniques are exactly the same as the corresponding block level techniques just described, except that they rely on coverage measured in terms of numbers of methods rather than numbers of blocks covered.

The test case prioritization techniques studied by B. Korel and J. Laski in [13] are primarily based on variations of the total requirement coverage and the additional requirement coverage of various structural elements in a program. For instance, total statement coverage prioritization orders test cases in decreasing order of the number of statements they exercise.

Jeffrey and Neelam Gupta [14] presented a new approach for prioritizing test cases that is based not only on total statement coverage (also known in that paper as branch coverage), but that also takes into account the number of statements executed that influence or have potential to influence the output produced by the test case. The set of such statements corresponds to the relevant slice, which is computed on the output of the program when executed by the test case given by. The approach is based on the following observation: if a modification in the program has to affect the output of a test case in the regression test suite, it must affect some computation in the relevant slice of the output for that test case. Therefore, their heuristic for prioritizing test cases assigns higher weight to a test case with larger number of statements in its relevant slice of the output. They used the following factors in their approach to prioritize test cases: (a) the number of statements in the relevant slice of output for the test case, because any modification should necessarily affect some computation in the relevant slice to be able to change the output for this test case and (b) the number of statements that are executed by the test case but are not in the relevant slice of the output.

Jones and Mary Jean Harrold [15] presented new algorithms for test-suite reduction and prioritization that can be tailored effectively for use with modified coverage (MC) and decision coverage (DC). Most existing techniques from researchers, who have been

investigating test suite reduction (also referred to as test suite minimization) and prioritization techniques consider a set of test-case coverage criteria such as, statements, decisions, definition user associations and specification items. In their paper, they focused on MC and DC criteria as for test case reduction and prioritization, building on Rothermel's test case prioritization technique et al. [16]. Their approach uses total requirement coverage and the additional requirement coverage to weight and schedule test cases accordingly.

2.1.3 Cost Effective-Based Prioritization Techniques

Cost effective-based techniques are methods of prioritizing test cases based on costs, such as cost of analysis and cost of prioritization. Many researchers have researched this area. The following paragraphs present existing cost effective-based test case prioritization techniques.

Leung and White [17] presented a cost model for regression test selection in. The proposed model incorporates various costs of regression testing, including the costs of executing and validating test cases and the cost of performing analyses to support test selection, and provides a way to compare tests for relative effectiveness. This model can be appropriately applied to an effective regression test selection techniques, which necessarily select all test cases in the existing test suite that may reveal faults. However, Leung's model does not consider the costs of overlooking faults due to discarded tests.

Alexey Malishevsky et al [18] presented cost models for prioritization that take these costs into account. They defined the following variables to prioritize test cases: cost of analysis, Ca(T) and cost of the prioritization algorithm, Cp(T). Then weight prioritization value (WP) is calculated as:

$$WP = Ca(T) + Cp(T)$$

where:

- Ca(T) includes the cost of source code analysis, analysis of changes between old and new versions, and collection of execution traces.
- Cp(T) is the actual cost of running a prioritization tool, and, depending on the prioritization algorithm used that can be performed during either the preliminary or critical phase.

Furthermore, Malishevsky [18] divided the regression testing process into two phases: preliminary phase and critical phase. Preliminary phase activities may be assigned different costs than critical phase activities, since the latter may have greater ramifications for things like release time.

The cost of a test case is related to the resources required to execute and validate it. Additionally, cost-cognizant prioritization requires an estimate of the severity of each fault that can be revealed by a test case. Fault severity may be used to order tests by the same two criteria listed previously. Meanwhile, Malishevsky et al. [19] focus on four practical code-coverage-based heuristic techniques. Those four techniques are: total function coverage prioritization (fn-total), additional function coverage prioritization (fn-addtl), total function difference-based prioritization (fn-diff-total) and additional function difference-based prioritization (fn-diff-addtl).

2.1.4 Chronographic History-Based Prioritization Techniques

Chronographic history-based techniques are methods to prioritize test cases based on test execution history. The following paragraphs present an overview of existing chronographic history-based test case prioritization techniques.

Jung-Min and Adam [20] proposed to use information about each test case's prior performance to increase or decrease the likelihood that it will be used in the current testing session. Their approach is based on ideas taken from statistical quality control (exponential weighted moving average) and statistical forecasting (exponential smoothing).

Kim [20] defined the selection probabilities of each test case, TC, at time, t, to be $P_{tc,\ t}$ (H_{tc}, α), where H_{tc} is a set of t, time-ordered observations {h_1, h_2, …h_n} drawn from runs of TC and α is a smoothing constant used to weight individual historical observations. The higher values emphasize recent observations, while lower values emphasize older ones. These values are then normalized and used as probabilities. The general form of:

P is $P_0 = h_1$ and $P_k = \alpha_{hk} + (1- \alpha)P_k$-1, 0 α<=1, k>=1.

When testing in a black box environment, source code related information is not available. In such situations, practitioners only have output of test cases and other run-time information available, such as the running time of test cases.

Bo Qu [21] proposed a prioritization technique based on this limited information. One general method of prioritization for black box testing is to initialize a test suite using test history, and then adjust the order of the rest of the test cases based on run-time information. To guide the adjusting strategy, a matrix R is used. They defined the matrix, R, to predict the fault detection relationship of test cases, so once a test case revealed regression faults, related test cases can be adjusted to higher priority to achieve a better rate of fault detection. Let T be a test suite, let T′ be a subset of T, and let R be a matrix which describes the fault detection relationship of test cases. Their general process of test case prioritization for black box testing can be described shortly as follows:

(a) Select T′ from T and prioritize T′ using available test history

(b) Build a test case relation matrix R based on available information

(c) Draw a test case from T′, and run it

(d) Reorder rest test cases using run-time information and test case relation matrix R and

(e) Repeat from step c until testing resource is exhausted.

A number of research works have been done on the Test Case Prioritization in order to test system effectively as discussed in this chapter but none of researcher has mapped their research with time as dead line for test case to identify errors in system. On the bases of literature survey, formulation of gap is done in chapter

Chapter 3

Gap and objectives

3.1 Gaps in Present Study

On the basis of literature review, following gaps has been identified.

Target of software engineering is to develop high quality software within time and budget. Since software testing is time consuming, error prone, full of uncertainty and expensive process. Industry does not have enough time or resources to run the entire test suite. Test case prioritization and selection is vital activity of software testing. The need of hour is to provide cost effective strategy for software test case optimization. Therefore, there is a strong need to decide which test cases to run first to maximize the fault detection. Quality of software testing is low due to inadequate test case prioritization and selection technique. By applying appropriate test case prioritization and selection technique, cost, efforts, uncertainty of software testing can be reduced considerably. Industry requires cost effective adequate technique for test case selection and prioritization. Fault detection is prime factor for evaluating adequacy and ranking the test cases. Test cases should prioritize on the basis of fault detection capability. Test cases should be designed, selected and prioritized in such a manner that it will detect the maximum number of faults in software under test within deadline of time and budget.

Genetic algorithm to reorder test cases in a test suite using execution time as a constraint had shown that prioritization technique is appropriate for regression testing environment and explains how the baseline approach can be extended to operate in additional time constrained testing circumstances.

Literature review is an evidence that several classical and search based techniques have been applied to find out the solutions for test case prioritization problem. Some of these works apply Ant Colony Optimization, Genetic Algorithms, Greedy Algorithms, Linear Programming, Case Base Reasoning, Fuzzy logic and so on. Ant Colony Optimization, Genetic Algorithms, Greedy Algorithms provide near optimal and local solutions to test case prioritization problem. Also, these approaches have not considered the precedence of test cases. Classical computing techniques such as Linear Programming provide single global

solution to test case prioritization and selection problem. Several classical and soft computing techniques have been explored for test cases selection and prioritization using code, requirement coverage, fault coverage, and cost. Many interesting results have been received but the test case prioritization and selection based on fault detection with time constraint using Binary Integer Programming technique has not been explored. So, there is still space for the researchers to explore and experiment the Binary Integer Programming based approach to find out the order of test cases on the basis of faults detections with time constraint

3.2 Objectives of Thesis

On the basis of gaps identified, following are the objectives of this thesis:

1.To study the various existing techniques for test case prioritization.

2.To investigate binary integer programming approach for software test case prioritization.

3.Propose framework for time aware test case prioritization based on fault coverage using binary integer program

Chapter 4

Fault Detection Based Test Case Prioritization Using Linear Programming Approach

4.1 Linear Programming

Linear Programming (LP), or Linear Optimization is a mathematical optimization technique, used for determining a way to achieve the best outcome (such as maximum profit or lowest cost).More formally, linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints. It is a specific case of mathematical optimization for real NP-hard problem. In LP, feasible region is a convex polyhedron, which is a set defined as the intersection of finitely many half spaces, each of which is defined by a linear inequality. Objective function is a real-valued affine function defined on this polyhedron. A linear programming algorithm finds a point in the polyhedron, where this function has the smallest (or largest) value if such point exists.

Linear programming problems can be expressed in canonical form:

$$Maximization\ Z = c^T x \qquad \ldots(1)$$

$$subject\ to\ constraint\ Ax \leq b$$

and $x \geq \mathbf{0}$

Where x represents the vector of variables (to be determined), c and b are vectors of (known) coefficients, *A* is a (known) matrix of coefficients, and C^T is the transpose matrix of C. The expression Z to be maximized or minimized, is called the objective function and inequalities $Ax \leq b$ are the constraints, which specify a convex polytypic. The function Z is to be optimized over this convex polytypic. In this context, two vectors are comparable when they have the same dimensions. If every entry in the first is less-than or equal-to the corresponding entry in the second then we can say the first vector is less-than or equal-to the second vector.

4.1.1 Integer Programming (IP) An integer programming problem is a mathematical optimization or feasibility program in which some or all of the variables are restricted to be integers. In many settings the term refers to integer linear programming,

which is also known as mixed integer programming when some but not all the variables are restricted to be integers. NP-hard problems can be solved by using integer programming. NP-Complete problems can be solved using Binary Integer Programming (BIP), a special case, 0-1 linear integer programming, in which unknowns variables are binary. However, integer programming problems with a constant number of variables (LP-type problem) may be solved in linear time.

All integer programming problems have linear equalities and inequalities and some or all of decision the variables are required to be integer. Types of integer programming are listed below:

- *Pure Integer Programming (PIP)*: Here all variables must have integer solutions.
- *Mixed Integer Programming (MIP)*: In this, some but not all variables have integer solutions.
- *Binary Integer Programming(BIP)*: Here all variables have values of 0 or 1
- *Mixed Binary Integer Programming (MBIP):* In this type, some decision variables are binary, and other decision variables are either general integer or continuous valued.

There are several methods to solve the integer programming problems. Some of them are as follows:

- *Cutting Plane Method:* A means of adding one or more constraints to linear programming problems to help in producing an optimum integer solution.
- *Branch and Bound Method:* An algorithm for solving all-integer and mixed-integer linear programming and assignment problems. It divides the set of feasible solutions into subsets that are examined systematically. Integer programming solver uses this method. It is more realistic and flexible method.

4.1.2 Binary Integer Programming (BIP)

In binary integer programming problems, each decision variable can only take the 0 or 1. It represents the selection or rejection of an option, the turning on or off of switches, a yes/no answer. This can also be used for test case selection. i.e. test case is selected or not. The specified test case will belong to test suite or not. It requires that the problem be put into a standard form:

$$Minimization\ z = \sum_{j=1}^{n} c_j * x_j \qquad \ldots(2)$$

Subject to constraint $\sum_{j=1}^{n} a_{ij} \geq b_i\ for\ i = 1,2,\ldots\ldots\ldots m$,

Where all decision variables x_j are binary variables (can only have a value of 0 or 1) and $1 \leq j \leq m$. All the coefficients of objective function are non negative integers. The variables are ordered according to their objective function coefficients, so that $0 \leq c_1 \leq c_2 \leq \cdots \leq c_n$.

This seems to be a restriction conditions, but many problems can easily converted into this form. Decision variable for negative objective function are handled by $(1-x_j)$ in place of decision variables x_j. It is also easy to reorder the variables. Constraint right hand sides can be negative, so ≤ constraints are easily converted to ≥ form by multiplying throughout by -1.

Capital budgeting problems, facilities location problems, airline crew scheduling problems, knapsack problems, NP-Complete problems are being solved using binary integer programming approach with high accuracy. BIP can also be explored for computer science and engineering problems like software testing, planning, routing, scheduling, assignment, and design problems.

4.1.3 Binary Variables

Binary digits (0 or 1) may the only possible values for a decision variable. It can be used for the decision variable dealing with yes or no. It is also called “auxiliary binary variable”. Decision variable (x_i) is defined as follows:

$$x_i = \begin{cases} 1 & \textbf{if an event takes place} \\ 0 & \textit{if an event not takes place} \end{cases} \qquad \ldots(3)$$

4.2 Time Aware Test Case Prioritization based on Fault Coverage using Binary Integer Programming

Software testing is the key technology for evaluating the fault detecting capability quantitatively and is a vital exercise in quality assurance. Software testing is time consuming, ambiguous, error prone and expensive process. The need of the hour is to provide a cost effective strategy for software testing. Common thread of test cases optimization is formed by

test cases classification, minimization, selection and prioritization. Test cases optimization is NP-Complete problem. However, by applying appropriate test case optimization techniques, these efforts and cost can be reduced considerably. Quality of the software testing is directly proportional to the optimization of test cases. Quality of software testing is low due to inadequate strategy for test cases optimization. Binary integer programming for software test cases selection and prioritization with test data adequacy criteria and automation of testing process will help in improving the overall quality of the software. Industry requires cost effective adequate technique for test case selection and prioritization. Fault detection or coverage is prime factor for evaluating adequacy and ranking the test cases. Test cases should prioritize on the basis of fault detection capability. Test cases should be designed, selected and prioritized in such a manner that it will detect the maximum number of faults in software under test within deadline of time and budget. Binary integer programming provides the best solutions to the NP-Complete problems, since test case optimization is NP-Complete problem and can be solved efficiently, accurately using binary integer programming approach.

Test prioritization schemes typically create a single re-ordering of test cases of the test suite that can be executed after many subsequent changes to the program under test. Test case prioritization techniques reorder the execution of test cases in test suite is an attempt to ensure that defects are revealed earlier in the test execution phase. If testing must be terminated early, a reordered test suite can also be more effective at finding faults than one that was not prioritized.

Using the preceding BIP approach, we can select a subset of test cases that can satisfy the time budget and maximize the number of faults covered by the selected test cases. However; there may be some unselected test cases that, if further selected, cannot exceed the time budget. Further selecting such test cases cannot increase the number of faults covered, but these test cases may still be helpful for criteria like statement coverage. To further select these test cases, BIP approach is used in which, we adopt a strategy similar to our total strategy incoming sections.

The proposed test case prioritization technique using BIP within a time restricted framework is implemented and evaluated. The technique uses the fault/error coverage and execution time information of the regression test suite as an input. In the proposed algorithm faults coverage acts as a cost of execution. We abbreviate the technique as BIP_TEST.

The basic block diagram for the Binary Integer Programming for Test Case Selection and Prioritization system (BIP_TEST) is shown in fig.1. The inputs to the system include details of the test suite i.e the test cases along with the faults covered by them and their execution time. These inputs are generally tabulated and are entered by the tester. The produced output has an order of test cases of test suite, finally selected and prioritized.

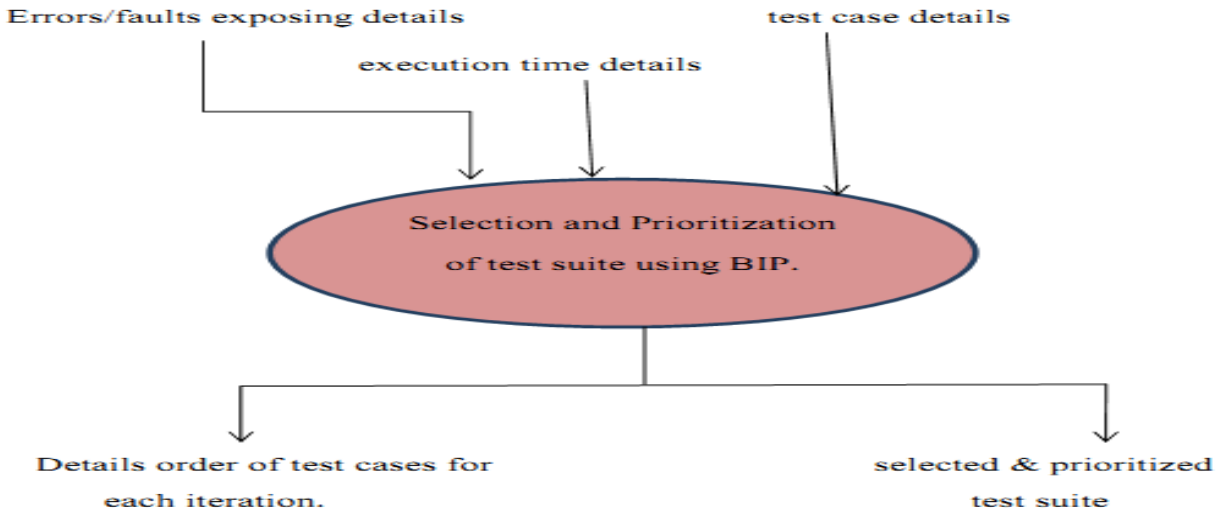

Fig: 4.1: Time Aware Test Case Prioritization based on Fault Coverage using BIP

4.2.1 Statement of the Problem

Let T be a test suite containing n elements and T= $\{t_1,t_2,t_3,.....t_n\}$ be a sequence on T. let F be a function from test suite elements to some domain on which the relation $\geq$ imposes a total order. T' is a test case prioritization of T with respect to F if and only if for all I, $\leq i \leq n$ -1 such that $F(t_i) \geq F(t_{i+1})$, that is F is monotonic over T.

4.2.2 Problem Formulation

A selective regression testing technique chooses a subset of test suite that was used to test the software before modifications were made and then uses this subset to test the modified software. Prioritization is the process of scheduling test cases in a order to meet some performance goal.

We define a test suite T as a n-tuple of test cases t_i, where i = 1to n and T= { t_1, t_2, t_3,……t_n}. The prioritization of T is denoted as T'. In the problem formulation, the maximum time within which a prioritize test suite must execute is the maximum capacity of test suite,

execution time of each test case is its weight and its value is number of faults covered. The output of algorithm is prioritized list that fits the required time limit.

4.2.3 Our Strategy

In time-aware total fault detection prioritization, select a subset (denoted as T') of the original test suite (denoted as T) such that T satisfies both the time budget and the following condition. For each test case t_i in T, the number of errors covered byt_i as E (t). Formalize test-case selection in time-aware test case prioritization as an BIP model consisting of decision variables, an objective function, and a constraint system described below followed by an illustration with the example.

Decision Variable

For each test case, a boolean decision variable is used to represent the test case selection. Thus, for test suite T= {t1, t2 ...tn}, use n boolean decision variables (denoted as x_i, where $1 \leq i \leq n$). Formally, x_i is defined as follows

$$x_i = \begin{cases} 1 & \text{if an event takes place} \\ 0 & \text{if an event not takes place} \end{cases} \quad \ldots(4)$$

Objective Function

The objective function is defined as follows:

$$\text{Max} \sum_{i=1}^{n} Nfc(t_i) * x_i \quad \ldots(5)$$

Constraint System

To ensure that the selected test cases satisfy the time budget, we define the constraint system of this model as the following inequality (i.e equation 6), which indicates that the sum of time required to find out the faults by all the test cases is no more than the time budget.

$$\sum_{i=1}^{n} time(t_i) * x_i \leq time_{\max} \quad \ldots(6)$$

In the constraint system $\sum_{i=1}^{n} time(t_i) * x_i$ represents the sum of time required to find the errors of selected test cases, because if test case t_i is not selected, x_i is 0 and its time is not counted in the sum in formula given by equation [(t] ,1).

Following are the assumptions made in the problem stated above

Original test suite is taken as T={ t_1 , t_2 , t_3 , ……. , t_n}

Set of all faults is defined as F ={f_1 , f_2 , f_3……. ,f_x}

Each test case t_i, where i=1to n in the original test suite covers some or all the statements.

4.3 Illustration with the Example

We have considered that there are six test cases in a regression test suite (denoted as T = {t_1 ,t_2, t_3, t_4, t_5, t_6 }) and there are six errors(denoted as F = {f_1, f_2 , f_3, f_4, f_5, f_6 }). The test cases and the time required to find out the fault/error are presented in Table 4.1. Let us suppose that the time budget is 19 seconds, we need to find out an ordering (denoted as T') of a subset of T under the condition that the execution time of all selected test cases (sum(time(t_i))) is no more than 19($time_{max}$)seconds. The ordering of test cases can also be effective in errors/faults coverage as early as possible.

Table 4.1: Fault/Error Identification and Execution Time of Test Cases

Test case/Error	f_1	f_2	f_3	f_4	f_5	f_6	Time (seconds)
t_1	1	1		1	1		9
t_2	1						2
t_3		1	1	1			6
t_4	1				1		4
t_5						1	5
t_6		1		1			6

The total fault/Error-coverage strategy yields the following BIP approach. The objective function and the constraint system are depicted in Formula 7 and Formula 8, respectively,

$$\max(4x_1 + x_2 + 3x_3 + 2x_4 + x_5 + [2x]_6) \quad (7)$$

$$9x_1 + 2x_2 + 6x_3 + 4x_4 + 5x_5 + 6x_6 \leq 19 \quad (8)$$

Thus, the total error-coverage strategy in this approach selects t_1 (covering four errors in nine seconds), t_3 (covering three errors in six seconds), and t_4 (covering two errors in four seconds) and the selected test cases in total covers the nine errors in 19 seconds. The ordering of test cases is, t_1 t_3, and t_4. This order of test cases detects maximum number of faults within the time constraint. Order of test cases is done by tester. The implementation of above work is done in MATLab 7.2 using binary integer programming tool box, and implementation and results are given in chapter

Table 4.2: Final ordering of Test C

Order of Test Cases	t_1	t_3	t_4	Adequacy
Faults Indentified	f_1, f_2, f_4, f_5	f_2, f_3, f_4	f_1, f_5	f_1, f_2, f_3, f_4, f_5
Number of Faults / Errors Covered	4	3	2	5 out of 6=83%
Execution Time	9	6	4	19 Seconds

4.3.1 Solution Representation

The problem can be represented in the form of a undirected graph G (V, E) where V is the set of vertices and E is the set of edges in the graph. i.e, test cases are represented by vertices in the graph. Each edge in the graph represents the pheromone trail associated with the edge $e_i \in E$, which reflects the amount of statement coverage s_i on the chosen path within time constraint, TC.

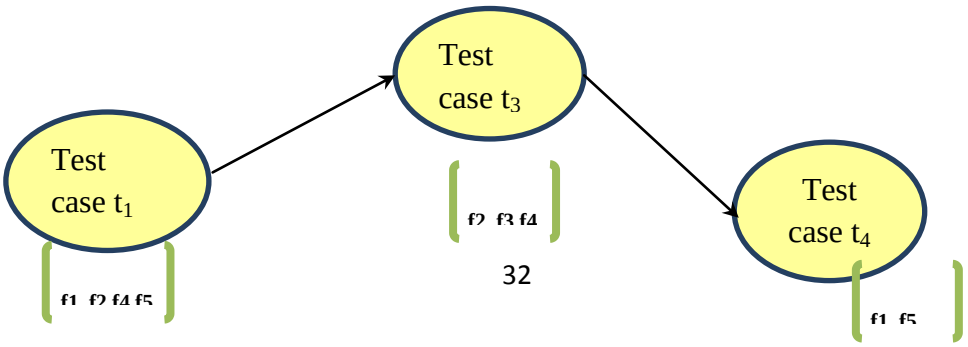

Fig. 4.2: Solution Representation

Chapter 5

Experimental Study

The implementation of proposed work is done in MatLab 7.2 using binary integer programming tool box. We used printokens2 program, a lexical analyser, written in c. It is

benchmark dataset freely available at http://www.sir.unl.edu. Details of experimentation are as follows:

5.1 Inputs for Experimental Study

For the experiment, we used printokens2 program, a lexical analyzer, written in c. It is benchmark dataset freely available at website http://www.sir.unl.edu of Software Infrastructure Repository (SIR).Out of the complete test pool, we have randomly chosen a test suite of 500(five hundred) test cases. Here, we find fault coverage and execution time of each test case. Fault Coveragebility and execution time can be computed using Perl script and time utility of Ubantu 11.2 and details are as follows:

a) **Perl Script:** First of all we compile the program printtokens2 using gcc gnu compiler. Subsequently, script is run for finding errors /fault coverage for specified test cases. Format of script command is as follows:

Source-file-name <Test-case-file-name>Output-file-name

Some sample commands of test script for finding the faults coveragebility for test cases (t_1) are given below:

echo script type: R

echo ">>>>>>>>>running test 1"

../source/print_tokens2.exe < ../inputs/newtst148.tst > ../outputs/t1

b) **Time** : time command is used to identify the execution time. Format of this command is as:

Time [options]command[arguments….]

The time command runs the specified program command with the given arguments. When command finishes ,time writes a message to standard output giving timing statistics about the program run. These statistics consist of :

- The elapsed real time between invocation and termination.
- The user CPU time.
- The system CPU time.

5.2 Program under Test

Printtokens2 version 2 is used for experimentation purpose. It has been taken from Software-Artifact Infrastructure Repository (SIR) and is a lexical analyser. It is a benchmark project and freely available at www.sir.unl.edu. This program has been implemented in C and consists of 19 procedures, 570 Line of Code (LOC). It has been originally created by Tom Ostrand and colleagues at Siemens Coporate Research and its version is SIRV: 2.0. It consists of 4115 test cases, out of which 500 test cases and 78 faults are used for implementation part.

5.3 Implementation

The algorithm has been named as BIP_TEST and is implemented in MATLAB programming according to the proposed algorithm. Binary Integer Programming tool (BINPROG) is used to select and prioritize the test cases using fault detecting capability with time constraint. We used xlsread() command to read the values of number of faults revealed and execution time of each test cases from Microsoft Excel file named 'fault.xlsx' located in same folder, and store number of faults revealed and execution time of each test cases into one dimensional array f and A respectively. Test cases selection and prioritization is done to achieve the maximum fault detection with time constraint of 100 second using BINPROG tool. Under the time constraint environment, program has been compiled and run with the required inputs (number of faults covered and execution time for each test case). MatLab commands for BIP_TEST are given below:

```
'Program for Software test cases selection & prioritization
based on faults 'coverage criteria using Binary Integer
Programming.
f=xlsread('fault.xlsx','Sheet1','CA0002:CA501')
faults=f';
faults=faults .* (-1);
faults
[A]=xlsread('fault.xlsx','Sheet1','CF0002:CF501')
A=A';
b = [100];
X = bintprog(faults,A,b);
disp('Test prioritization & Selection are as follows');
X
```

Implementation of code in MATLAB has led to the following results:

Table 5.1: Order of Test Cases in Test suite

S.No.	Test Case ID	**Number of errors identified**	**Execution time**
1	t244	**4**	**1**
2	t285	**4**	**1**
3	t24	**4**	**2**
4	t69	**4**	**2**
5	t156	**4**	**2**
6	t204	**4**	**2**
7	t312	**4**	**2**
8	t345	**4**	**2**
9	t504	**4**	**2**
10	t512	**4**	**2**
11	t134	**3**	**1**
12	t266	**3**	**1**
13	t282	**3**	**1**
14	t352	**3**	**1**
15	t444	**3**	**1**
16	t477	**3**	**1**
17	t511	**3**	**1**
18	t530	**3**	**1**
19	t533	**3**	**1**
20	t546	**3**	**1**
21	t563	**3**	**1**
22	t594	**3**	**1**
23	t5	**2**	**1**

24	t16	**2**	**1**
25	t18	**2**	**1**
26	t27	**2**	**1**
27	t28	**2**	**1**
28	t29	**2**	**1**
29	t37	**2**	**1**
30	t62	**2**	**1**
31	t86	**2**	**1**
32	t89	**2**	**1**
33	t102	**2**	**1**
34	t104	**2**	**1**
35	t111	**2**	**1**
36	t124	**2**	**1**
37	t127	**2**	**1**
38	t129	**2**	**1**
39	t130	**2**	**1**
40	t141	**2**	**1**
41	t142	**2**	**1**
42	t144	**2**	**1**
43	t148	**2**	**1**
44	t150	**2**	**1**
45	t152	**2**	**1**
46	t205	**2**	**1**
47	t210	**2**	**1**
48	t216	**2**	**1**

49	t219	**2**	**1**
50	t223	**2**	**1**
51	t224	**2**	**1**
52	t229	**2**	**1**
53	t230	**2**	**1**
54	t231	**2**	**1**
55	t233	**2**	**1**
56	t235	**2**	**1**
57	t236	**2**	**1**
58	t247	**2**	**1**
59	t255	**2**	**1**
60	t269	**2**	**1**
61	t270	**2**	**1**
62	t278	**2**	**1**
63	t283	**2**	**1**
64	t286	**2**	**1**
65	t287	**2**	**1**
66	t288	**2**	**1**
67	t289	**2**	**1**
68	t309	**2**	**1**
69	t310	**2**	**1**
70	t323	**2**	**1**
71	t324	**2**	**1**
72	t333	**2**	**1**
73	t335	**2**	**1**

74	t339	**2**	**1**
75	t349	**2**	**1**
76	t353	**2**	**1**
77	t416	**2**	**1**
78	t417	**2**	**1**
79	t418	**2**	**1**
80	t419	**2**	**1**
81	t420	**2**	**1**
82	t422	**2**	**1**
83	t427	**2**	**1**
83	t428	**2**	**1**
85	t429	**2**	**1**
86	t434	**2**	**1**
87	t442	**2**	**1**
88	t443	**2**	**1**
89	t448	**2**	**1**
90	t452	**2**	**1**
91	t456	**2**	**1**
92	t468	**2**	**1**

As per the above table, pictorial representation for the final iteration is shown in Fig 5.1

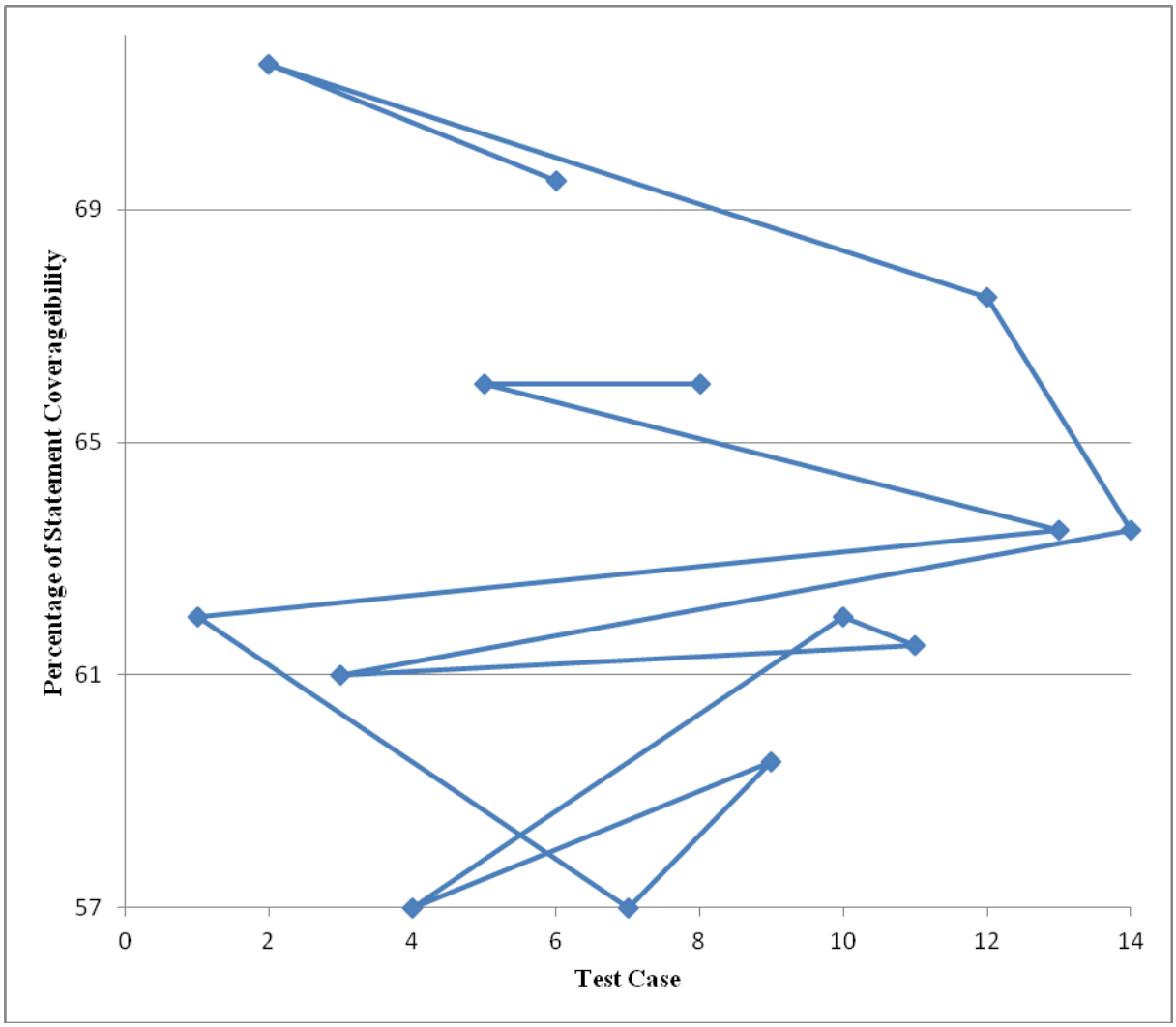

Fig. 5.1: Test Case Prioritization for Last Iteration.

Chapter 6

Conclusion

In this thesis, have proposed the test case selection and prioritization approach based on BIP to improve fault coverage and reduce the total execution time for regression testing, and obtained the optimal solution in optimum time. Proposed work gave cheering results. BIP is strong and robust as it involves positive feedback and parallel computations and hence, it can lead to better solutions in optimum time. The proposed algorithm is validated by analyzing an industrial project. Result indicates that the proposed technique lead to improved rate of fault detection in comparison to non-prioritized test cases. Multi-objective test case prioritization and selection using Binary Integer programming will be explored in future. We have also plan to further improve our approach using additional coverage rather than resorting to total coverage, when the full coverage is reached, and considering the time and cost required to find the fault when prioritizing the selected test cases. In addition to above, we plan to investigate soft computing techniques for test case selection for time, cost aware prioritization

References

[1] R. S. Pressman, “Software Engineering: A Practitioner’s Approach”, McGraw Hill, New York, 1992.

[2] D. Perry and G. Kaiser, “Adequate testing and object-oriented programming”, JOOP, pp: 13-19, 1990.

[3] Hema Srikanth and Laurie Williams,“Requirements-Based Test Case Prioritization”,North Carolina State University, ACM SIGSOFT Software Engineering, pages 1-3, 2005.

[4] Amitabh Srivastava and Jay Thiagarajan,“Effectively Prioritizing Tests in Development Environment”, In Proceedings of the International Symposium on Software Testing and Analysis, pages97-106, 2002.

[5] K. Onoma, W.-T. Tsai, M. Poonawala, and H.Suganuma, “Regression testing in an industrial environment”, Comm. Of the ACM, 41(5):81–86,1988.

[6] David Leon and Andy Podgurski, “A Comparison ofCoverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases”, Proc. Int’l Symp. Software Reliability Eng., pp. 442-453, 2003.

[7] Gregg Rothermel, R. H. Untch, C. Chu, and M. J.Harrold, “Test case prioritization: An empiricalstudy”, In Proceedings of the IEEE InternationalConference on Software Maintenance, pages 179-188, Oxford, England, UK, 1999.

[8] R. Bryce and C. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Journal of Information and Software Technology, 48(10):960–970, 2006.

[9] David Leon and Andy Podgurski, “A Comparison ofCoverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases”, Proc. Int’lSymp. Software Reliability Eng., pp. 442-453, 2003.

[10] Xiaofang Zhang, Changhai Nie, Baowen Xu and BoQu, “Test Case Prioritization based on VaryingTesting Requirement Priorities and Test CaseCosts”, Proceedings of Seventh InternationalConference on Quality Software (QSIC’07), 2007

[11] R. Bryce and C. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Journal of Information and Software Technology, 48(10):960–970, 2006.

[12] Hyunsook Do and Gregg Rothermel, “A ControlledExperiment Assessing Test Case Prioritization Techniques via Mutation Faults”, Proceedings of the IEEE International Conference on Software Maintenance, pages 411-420, 2005.

[13] B. Korel and J. Laski, “Algorithmic software fault localization”, Annual Hawaii International Conference on System Sciences, pages 246–252,1991.

[14] Dennis Jeffrey and Neelam Gupta, “Test Case Prioritization Using Relevant Slices”, InProceedings of the 30th Annual International Computer Software and Applications Conference, Volume 01, 2006, pages 411-420, 2006.

[15] James A. Jones and Mary Jean Harrold, “Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage”, In Proceedings of the International Conference on Software Maintenance, 2001.

[16] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrinand Christie Hong, “An Empirical Study of the Effects of Minimization on the Fault DetectionCapabilities of Test Suites”, In Proceedings of IEEE International Test Conference on SoftwareMaintenance (ITCSM'98), Washington D.C., pp. 34-43, 1998.

[17] H. K. N. Leung and L. White, “A cost model tocompare regression test strategies”, In Proc. Conf. Softw. Maint., pages 201–208, 1991.

[18] Alexey G. Malishevsky, Gregg Rothermel, Sebastian Elbaum, “Modeling the Cost Benefits Tradeoffs for Regression Testing Techniques”,International Conference on Software Maintenance (ICSM’02), 2002.

[19] Gregg Rothermel, Sebastian Elbaum, AlexeyMalishevsky, Praveen Kallakuri and Brian Davia,“The Impact of Test Suite Granularity on the Cost-Effectiveness of Regression Testing”, 2001.

[20] Jung-Min Kim and Adam Porter, “A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments”, In Proceedings of the International Conference on Software Engineering (ICSE), pages 119–129. ACM Press, 2002.

[21] Xiaofang Zhang, Changhai Nie, Baowen Xu and Bo Qu, “Test Case Prioritization based on Varying Testing Requirement Priorities and Test CaseCosts”, Proceedings of Seventh InternationalConference on Quality Software (QSIC’07), 2007.