Optimal Test Case Selection and Prioritization for Regression Test Suite

Dr. T.PREM JACOB

Publication Partner: IJSRP INC.

3/10/2025

Publication Partner:
International Journal of Scientific and Research Publications (ISSN: 2250-3153)

Optimal Test Case Selection and Prioritization for Regression Test Suite

Dr. T.Prem Jacob

Publishing Partner:
IJSRP Inc.
www.ijsrp.org

ISSN 2250-3153
9 772250 315302

Preface

Verification, validation, and testing have become essential in software engineering due to the growing complexity, size of software systems and the pivotal role of software in everyday life. The goal is in revealing high fault, with minimum cost which is targeted by Model Based Testing [MBT]. It is done by automating test case generation in a systematic way from abstract models of the software under test. The total testing cost includes the cost of test execution and evaluation. Automation reduces the test generation cost. Of course, there are some limitations wherein the testing approach in industrial systems cannot be moved scalable. Since all dimensions of the testing cost needs to be addressed. This calls for MBT adjustment in regard to its size of the output test suite. This research proposes different techniques to minimize the test suite size and preserve its fault detection rate which will be higher with more diverse test cases through the techniques called mutual based test case selection and prioritization. Using potential approaches, the research is invited for assigning review on search based techniques for test case generation. The research continues the choice of identifying the effective ways of the best techniques through regions empirical analysis.

To have reliability of the results and the effectiveness of the techniques, influential factors are explained through controlled experiments to analyze the problem. The existing selection techniques and the cost effectiveness of various approaches are also compared in the literature. Complementary study on estimating the best size for a test suite, based on mutation frequency comparisons among the test cases is a significant contribution. The complementary study increases the usability of the techniques proposed where testers are not needed to select an arbitrary test suite size.

To conclude, comparison of mutual based test case selection and prioritization, as the proposed technique proves to be more effective than the existing techniques. MBT is applied to adjust of the output test suite in accordance with the test budget. Hence it shows the significant impact of MBT through this research.

Publication Partner:
International Journal of Scientific and Research Publications (ISSN: 2250-3153)

Copyright and Trademarks

Authors

Dr. T. Prem Jacob received the B.E. degree in Computer Science and Engineering from C.S.I. Institute of Technology, Manonmaniam Sundaranar University, Nagercoil, India, in 2004, M.E. degree in Computer Science and Engineering from Sathyabama University, Chennai, India, in 2006, and Ph.D. degree from Sathyabama University, Chennai, India. He is a Professor of Computer Science and Engineering at Sathyabama Institute of Science and Technology, Chennai. He has participated in and presented many research papers at international and national conferences. He was awarded Top 2% Most Influential Scientists in 2023 Stanford University List: Analysis of Indian Researchers. His areas of interest include Software Engineering, Data Science, Data Analytics and Cloud Computing.

Table of Content

1. PROBLEM FORMULATION

1.1 OVERVIEW

It is the practice of retesting a program after making any modifications to the program to assure that its previously tested behavior is preserved (non-regression)[1][2]. The goal of regression testing is to find regression bugs, such as defects introduced as a side effect to the modified program. Although new test cases are required for achieving the goal, regression testing is mainly performed by rerunning some subset of the programs existing test cases[3]. Thus Corey Sandler et al (2004) describe regression testing as "Saving test cases and running them again after changes to other components of the programs" [4][5].

During the regression testing phase, testers face many challenges for rerunning test cases. Since saving and rerunning test cases from a previous version is an expensive task[11].

For the purpose of rerunning the test cases, it must be either recorded or described. As the program evolves for adapting a potentially changed specification of the software system, its corresponding test suite needs to be maintained. Maintenance of test case reveals some irrelevant test cases in regard to the new version of the product called obsolete test cases [12]. Obsolete test cases though removed, many of the test cases exercise areas of code which are irrelevant to the previous version changes. Rerunning those test cases with extra effort has little chance of finding new faults[6]. This motivates towards selection of subset of the test cases for re-execution which saves the effort with no significant risk. Limiting test cases to those related changes, the selected test remain beyond what the organization affords to execute. Here test cases needs to be chosen by the testers with high priority and execute them[7].

To describe the concepts in the context of regression testing the following notations are used. Let P be the current version of the program under test and P′ be the next version of P. Let S be the current set of the specifications for P, and S′ be the set of specifications for P′. T is the existing test suite. Individual test cases are denoted by t. P(t) stands for the execution of P using t as input.

The regression testing process involves tasks pertaining to each of the described challenges. Consider a program that has changed from P_v to P_{v+1}, that have a test suite T_v, created and run on P_v, the regression testing process which involves the foresaid tasks are interesting with research problems.

(i) Maintaining T^v to get $T_m{}^v$, identifying obsolete test cases and repairing or discarding them.

(ii) Optimizing $T_m{}^v$ to get $T^o{}_v$, selecting a subset of test cases and prioritizing the selected test cases.

(iii) Running $T^o{}_v$ on P_{v+1}, checking the results and identifying failures.

(iv) Creating a new test suite $T^n{}_{v+1}$, if necessary, to test P_{v+1}

(v) Running $T^n{}_{v+1}$ on P_{v+1} ,checking the results identifying potential failures.

(vi) Aggregating $T^o{}_v$ and $T^n{}_{v+1}$ to get T_{v+1} and saving it for future testing.

This research work shows concern on the second step i.e. creating new obsolete test case, how to repair test case, how to rerun test case efficiently, creating new effective test suites, which can be treated as the scope of this research work. The problem of regression test suite involves fine grained problems. Here the focus of this research first introduces a general form of the problem and then presents an instance of the general problem.

1.2 PROBLEM STATEMENT 1 – TEST SUITE OPTIMIZATION

Definition

Given: A program P_v, its maintained test suite $T_m{}^v$, and a new version P_{v+1}.

Problem To find a test suite $T^o{}_v$ $T_m{}^v$ such that it can most effectively, according to some criteria, find the regression bugs in P_{v+1}.

Definition 1: A test suite is a tuple of test cases and the execution sequence is indicated by the order of these test cases.

If a test case t_1 appears before t_2 in the tuple, t_2 is not run unless t_1 runs. The obsolete test cases are tracked before starting the test optimization.

According to the specification of P_{v+1}, that may differ from P_v, entire test cases in T_m^v are clearly expected to pass. Some test cases which are obsolete should be removed as part of the test case maintenance, as obsolete test cases that pertain to the software features, do not exist anymore. But those test cases must be fixed to meet the new requirements. As in definition 1, the regression fault may relate the situation where software functionality works as desired in $P_{v,}$ but it is faulty in P_{v+1}.

While criterion of effectiveness specify the goal of the test case optimization. Different goals for regression test optimization emphasize different products, testing process and organization policies. For instance, in a safety critical project any test case that can find bugs must be executed. The less sensitive products in the competitive markets, organizations may take risk by discarding few test cases for releasing the software at the earliest[8][9]. It reduces the size of test suites by encouraging the effectiveness criteria[10]. The effectiveness criteria which provides greater flexibility can be helpful, if an organization is not certain about when to stop regression testing.

According to the researchers there are two predominant problem specifications in the literature that are test case selection and test case prioritization. According to some criteria, test case selection problem optimizes the regression test suite by eliminating a subset of test cases. Since, the test cases that exists cannot reveal faults even if it is executed, wherein elimination of these can optimize the regression test suite. For instance, the test case which are not related to the changed or impacted parts of the code are not possible to reveal the faults. This elimination of test cases reduces time required for the regression testing phase. While the regression test selection does not have concern for the execution order of the resulting test suite. The test case prioritization problem focuses on the order of the test cases appearing in the final test suite (T^o_v).

1.3 PROBLEM STATEMENT 2 – TEST CASE PRIORITIZATION

The test case prioritization problem orders the test cases in the test suite so that certain property gets optimized for that ordering. The formal definition of this problem is widely accepted in the literature (Elbaum et al 2000) and this could be rephrased as,

Definition: 2

Given: A program P_v, its maintained test suite T_m^v, a new version P_{v+1}, and a function f(T) form a test suite T to a real number.

Problem: To find an ordered test suite $T^o_v \in Perm(T_m^v)$ such that $\forall\ T \in Perm(T_m^v)$, $f(T) \leq f(T^o_v)$.

In Definition 2, Perm(T) denotes the set of all permutations (orderings) of the test suite T and f(T) function is a score function reflecting the goal of prioritization. When prioritizing test cases the testers can follow different goals[13]. They may find bugs faster, cover more parts of code faster, find severe faults sooner, run easier to execute test cases earlier and so on.

The orderings that attains the goal faster are assigned higher values by the score function f(T). The prioritization goals of early fault detection which has been extensively researched through this research work. In other words, such a goal informally means a test suite is preferable if it finds bugs faster than others. Rothermel et al used a measure called Average Percentage of Faults Detected (APFD), to remind f(T) function in the above definition [15][16][17][18]. This metric Average Percentage of Faults Detected during the execution of a test suite and if the number it gives is higher, such as between 0 and 100 which indicates faster fault detection rate and the mathematical definition of this measure is found in[14].

There are two main advantages for early fault detection in test case prioritization. First, the cost that are associated with any bug is partly related to the time it discovered. Debugging starts earlier, if a bug is found early in the regression testing process where bugs could be fixed faster. Due to the late detection of faults which restricts the available time for fixing the bugs where bugs remain unfixed. Second, due to incomplete testing, if the testers are forced by time constraints which halts regression testing prematurely. The early fault detection ensures fewer missed faults because of incomplete testing. A cut-off is likely to occur in this situation where rerunning all the test cases is not feasible. Obviously, test case prioritization and selection are similar to each other.

Early fault detection as the goal of prioritization, the existing faults and the test cases that reveal those faults are not known when prioritizing the test suite to achieve this goal. One has to rely on the heuristics regarding the behaviors of faults and test cases to improve the fault detection rate. Covering the fault prone area of the code can accelerate the fault detection rate.

2. TEST CASE SELECTION AND PRIORITIZATION

2.1 OVERVIEW

The test case selection techniques selects the test case for testing the modified portion of the software. This test case selection problem or regression test problem resembles the test suite minimization problem and both problems are related to choosing the subset of the test cases from the test suite.

The difference between these two approaches focus on the changes in the SUT. Test suite minimization is based on metrics such as coverage is measured from a single version of the program under test. Test cases are selected in the regression test because of their relevant execution to the changes between previous and the present version of the SUT.

Test case prioritization techniques in which each test case is assigned priority according to some criteria involved in scheduling the test cases in an order so as to improve the performance of regression testing. The test case that has the faster code coverage is given the highest priority.

The advantage of this technique is that it does not remove or discard the test cases from the test suite. The other reason is the rate of detecting the faults. The main objective of this research work is to develop a test case prioritization techniques that depends on the fault detection rate.

The condition of safety under regression test selection techniques ensuring the selected subset of a test suite with same fault detection capabilities is not always safe. Hence this creates the need to schedule the test cases called test case prioritization. The test cases are prioritized towards increasing the effectiveness in order to meet the performance goal.

2.2 Test case selection and prioritization

Regression Testing is an activity to make sure that modifications performed in the specific part of the software can be tested and they do not incorporate other unexpected behavior [19]. Towards testing ambiguities in the software, all the test cases prepared in the development phase should be executed, even though this activity is time consuming and expensive.

Researchers proposed many techniques to reduce both time and cost. These proposed techniques include regression test selection[20], regression test prioritization and hybrid approaches etc. The researchers have proposed a hybrid technique [21] for test case selection using ACO and GA. In this research the above proposed techniques is implemented and compared with the regression test case selection using ACO techniques [22].

Generally the regression testing is performed in a time constraint environment only for a predetermined time. Walcott gave one similar time aware test case prioritization techniques. It schedules the test suite both in terms of execution time and potential faults detection information. Singh et al in 2010 used Ant Colony Optimization (ACO) and proposed a new time constraint prioritization techniques [23].

2.2.1 Test Case Prioritization Approaches

2.2.1.1 General

For a program P and T is the test suite. Test case in T is prioritized without having any knowledge of the modifications made to the program, so that P gets modified to the program P' [24].

2.2.1.2 Version Specific

Prioritizing the test cases are carried out in accordance to the changes made to the program P. A new technique is proposed to achieve the code coverage for the modified code at the fastest rate possible. Based on some criteria test suites are scheduled. The objective of this technique is to enhance the possibility that if the test suites are used for regression testing in the given order, they will more closely meet some objective than they would if they were executed in some other order [25].

Test case prioritization can address a wide variety of objectives, as given below:

1. Software developers or testers intend to increase the rate of fault detection.
2. Detecting the high risk faults earlier in testing life cycle.
3. To increase the possibility of regression errors related to specific code changes very early in testing process.
4. To enhance the coverage of coverable code at a faster rate.
5. To make a system more reliable.

2.2.1.3 Weight Based Methods

This is used to prioritize the GUI Test Cases (GTC). This general prioritization techniques treats all the event actions equally and depends upon the test case length in order to prioritize the ordering. The test case events of greater length are prioritized first for execution. But when critical events are found in a shorter test case, which may detect more faults during testing and then it will gain higher priority for execution.

If two test cases have the same length, the most common methods usually choose one of the two test cases randomly. These special conditions impact the fault detection rate.

The non-weighted Event Flow Graph (EFG) is extended to a Weighted Event Flow Graph (WEFG) to solve the non-weighted problem. Three different ordering the GUI Test Cases (GTC) priority are proposed by the weight based method. This includes equal weight, fault prone weight and random weight. The weight based methods are ranked from high-to-low after calculating the weight of the GTC. This test case can be used to find many faults if the weight is larger and then these are analyzed with the proposed methods and later they are compared with other GTC prioritization approaches.

2.2.1.4 Weight Based EFG Methods

Creating Weighted Event Flow Graph

Extending the construction of an EFG to WEFG, all events are classified into one of the following events before assigning a weight to each events. The five events are restricted focus events, unrestricted focus events, termination events, menu open events and system interaction events[27]. These classifications of events are used to identify components in the GUI application. Next the weight of each event type is decided [26]. The weight of each event will be assigned randomly. Totally, there are three approaches for assigning a weight to each event.

Equal Weight

Here a value of 1 is assigned to the weight for all the events representing identical importance. It means that the weight for a system interaction event or a termination event is equal.

Fault Prone Weight

In a GUI application different levels of importance should be assigned to different events. For instance, the importance of a menu open event may be lower than a button click event because the former merely calls up a menu list but the latter may trigger one or more functions to change the system state. Hence, according to the relationships between event types and errors discussed, we set different event types as different WVs. We assume that higher weighted events will lead to higher rates of fault detection than lower weighted events[28]. For this purpose, the WVs from 5 to 1 will be assigned to five event types.

Random weight

While creating the random weight graph, a random value ranging from 5 to 1 is assigned to each event. For instance, in the test case including events E1,E3,E4,E7,E9,E10,E11, where E1 is restricted event, E3,E4,E7 are system interaction events, E9 is an unrestricted focus event, E10 is a menu open event and E11 is a termination event. The WVs for all event types are E1 = 2, E3 = 3, E4 = 5, E7 = 5, E9 = 4, E10 = 1. These WVs are randomly generated.

2.3 ANT COLONY OPTIMIZATION

Ant Colony Optimization (ACO) algorithm is a probabilistic technique used to solve computational problems through graphs. Marco Dorigo proposed this technique which has its strength in the overwhelming behavior of ants, searching for a way between their colony and source of food. Ants which are blind, communicate with their colony by yielding a chemical substance, called pheromone, along the path they traverse. Other ants, which use this chemical substance pheromone trail, follow the path with maximum pheromone trail. This process is continued till sufficient food is gathered and the chemical substance pheromone, also evaporates with time. Several problems such as knapsack problem, traveling salesman problem, distributed network, telecommunication network, vehicle routing and test data generation are solved by using ACO techniques.

Prioritization of test cases can be done in terms of random, optimal statement coverage total or additional, branch coverage total or additional, failure rates or total Fault Exposing Potential (FEP) of the test cases (Jiang et al 2009).

This algorithm remains as the foundation for this research which present a tool called ACO_TCSP for the same, showing the results for the execution of the tool on the same example used by Singh et al (2010). The outcome of the execution provides near optimum results and further motivates to test the tool on various larger examples to confirm the generality of its achievements.

ACO, which is a meta-heuristic approach which has been successfully used in solving NP hard optimization problems (Dorigo et al 1996). Artificial ants have been applied successfully on different applications towards world class problems like vehicle routing, quadratic assignment, scheduling, sequential ordering, routing in internet like networks (Dorigo et al 2006). Rothermel described prioritization for large software development environments. Kim et al (2002) proposed prioritization of test cases based on historical execution of test data. Li et al (2007) made a study by using various Greedy Algorithm and time aware regression test prioritization, where testing is done within a fixed period of time, has also been proposed (Kaushik et al 2011).

Ant Colony Optimization techniques is based on search algorithm of artificial intelligence for optimal solutions with a set of instructions. Here Colorni et al (1991), Dorigo and Marco (1992) and Dorigo et al (1991) have proposed ANT System, the iconic member. Ants which are blind and small in size are capable of finding the shortest route to their food source by making use of their antennas and pheromone liquid. Inspired by the behaviors of live ants, ACO maintains an array list to synchronize with searching problems solutions for local problems and also maintain information already gathered by each ant.

Two important processes with which ACO deals are

1. Pheromone deposition
2. Trail pheromone evaporation

Pheromone deposition is the phenomenon of ants adding the pheromone on all paths they follow. Pheromone trail evaporation means decreasing the amount of pheromone deposited on every path with respect to time. Thus updating criteria is applicable to each combinational problem depending on its own local search and global search respectively[29]. The path for each ant is selected on the basis of the amount of “pheromone trail” present on the possible paths starting from the current node of the ant. Ants randomly choose their path in the case of equal or absence of pheromone on adjacent paths. The probability of the path is increased by the presence of pheromone trail on the path. Ants follow the same process for reaching the next node. The same process continues till the ants reach the starting node and this gives the solution for analyzing the shortest or the best path towards attaining optimality.

Test case prioritization techniques is implemented and evaluated by using Ant Colony Optimization within the time limit framework[30]. The fault detection and execution time information of the regression test suite are used as input by this technique. Execution time is treated as the cost of executing the test case in the proposed algorithm. Prioritization is performed in such a way to achieve total fault detection with minimum cost of execution. The technique is abbreviated as ACO_TCSP. The basic block diagram for the ACO_TCSP (Ant Colony Optimization for Test Case Selection and Prioritization) system is shown in Figure 2.1. This system includes input details of the test suite such as the faults covered in the test suites with their execution time. Generally these inputs are tabulated which need to be entered by the tester. The user of the ACO_TCSP tool needs to enter the time constraints details at the runtime. The produced output has the details of the path for each iteration, pheromone details, the best path details and the final selected and prioritized test suite.

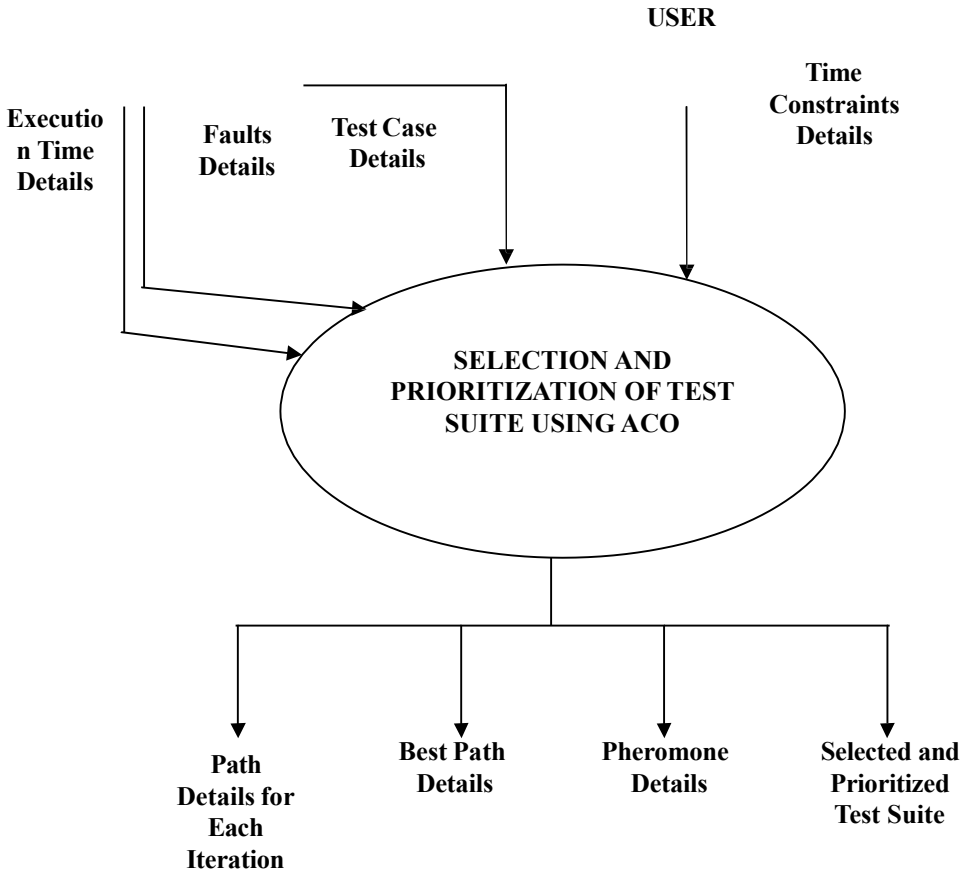

Figure 2.1 Block diagram of ACO_TCSP System

Ant colony optimization can be treated as a promising technique for solving test case selection and prioritization problem in which a tool ACO_TCSP has been applied. The results obtained are in close proximity to the optimal results although the best solution is not found in this test. The size of the test suite is reduced to 62.5% in all the four test runs, encouraging the use of developed tool by testers. Optimal test case selection process using ACO algorithm is represented with following tables, with constant time constraints, the ACO_TCSP was run four times on the example with constant time Constraints TC = 85 time units. It assumes that input to the ACO_TCSP with prior awareness of the detected faults and execution time of all the test cases. The same is tabulated in Table 2.1. The result of the simulation of Table 2.1 and TC as input for four sample runs are given in Table 2.2. This table consists of the report of all iterations related to each run, the best path and its execution time. This includes the edges with final weight on these edges and the path identified in that run. Table 2 proves ACO_TCSP found 3 out of 4 times, the optimal path. More iteration process for selecting the optimal test case sequence is used in this algorithm.

Table 2.1 Test Cases with Corresponding Faults Covered and the Execution Time

Test case/faults	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	No. of Faults Covered	Execution Time (ms)
T1		X		X			X		X		4	7
T2	X		X								2	4
T3	X				X		X	X			4	5
T4		X		X					X		3	4
T5			X			X				X	3	4
T6	X						X				4	5
T7			X			X		X			3	4
T8		X								X	2	2

Table 2.2 Optimal path using ACO_TCSP Algorithm

RUN	Iteration	ANT	Best Path	Execution Time	Result
1	1	A1	1,5,8,3	18	Weight on edges after all the Iterations 1, 5 : 0.531441 3, 4 : 6.12459 3, 8 : 1.121931 4, 5 : 6.12459 5, 8 : 0.531441 Rest all edges have 0 weight Execution Time for all Iterations is 85 Best Path is found to be : 3, 4, 5 OPTIMUM PATH FOUND IN THIS RUN
	2	A8	8,3,4,5	15	
	3	A5	3,4,5	13	
	4	A3	3,4,5	13	
		A5	5,4,3	13	
	5	A3	3,4,5	13	
		A5	5,4,3	13	
	6	A3	3,4,5	13	
		A5	5,4,3	13	
2	1	A5	5,1,3	16	Weight on edges after all the Iterations 1, 3 : 7.902621 1, 5 : 7.902621 Execution time for all Iterations 96 Best path is found to be 3, 1, 5
	2	A3	3,1,5	16	
		A5	5,1,3	16	
	3	A3	3,1,5	16	
		A5	5,1,3	16	
	4	A3	3,1,5	16	
		A5	5,1,3	16	
	5	A3	3,1,5	16	
		A5	5,1,3	16	
	6	A3	3,1,5	16	
		A5	5,1,3	16	

Table 2.2 (Continued)

RUN	Iteration	ANT	Best Path	Execution Time	Result
3	1	A8	8,1,3,5	18	Weight on edges after all the Iterations 1, 3 : 1.712421 1, 8 : 0.531441 3, 5 : 7.246521 4, 5 : 5.5341 Execution Time for all Iterations is 86 Best Path is found to be : 3, 4, 5 OPTIMUM PATH FOUND IN THIS RUN
	2	A1	1,3,5	16	
		A5	5,3,1	16	
	3	A4	4,5,3	13	
	4	A3	3,4,5	13	
		A4	4,5,3	13	
	5	A3	3,5,4	13	
		A4	4,5,3	13	
	6	A3	3,5,4	13	
		A4	4,5,3	13	
4	1	A1	1,3,5	16	Weight on edges after all the Iterations 1, 3 : 3.024621 3, 5 : 7.173621 4, 5 : 4.149 4, 8 : 0.729 Execution Time for all Iterations is 89
	2	A1	1,3,5	16	
		A5	5,3,1	16	
	3	A1	1,3,5	16	
		A5	5,3,1	16	
	4	A8	8,4,5,3	15	
	5	A3	3,5,4	13	
		A4	4,5,3	13	
	6	A3	3,5,4	13	
		A4	4,5,3	13	

2.4 GENETIC ALGORITHM

Genetic algorithm is stochastic search technique which is based on selection of the fittest chromosome. In this different codes such as binary, real numbers, permutation represents the population of chromosome. In order to find the fittest chromosome, genetic operations like selection, crossover and mutation are applied on the chromosome and a suitable objective function defines the fitness of the chromosome[31]. As a class of stochastic methods genetic algorithm is different from a random search. While genetic algorithm provides a multidimensional search by maintaining population of potential user[32]. Random methods consisting of a combination of iterative search methods. Simple random search methods can find a solution for a given problem. One of the genetic method's most attractive features is to explore the search space by considering the entire population of the chromosome.

The steps of genetic algorithm are:

1. Generate population (chromosome).
2. Evaluate the fitness of generated population.
3. Apply selection for individual.
4. Apply crossover and mutation.
5. Evaluate and reproduce the chromosome.

Generate Population

The initial population is selected randomly and encoded. Possible solution of the problem is represented by each chromosome. The chromosome is the sequence of the test cases and the aim is to attain the optimized sequence.

Evaluate the Fitness

The objective function exposes the fitness of the chromosome. A real number is generated by the objective function from the input chromosome. Using this number two or more chromosomes are compared.

Apply Selection

The selection is made depending on the fitness value of the chromosome. Based on the problem definition the chromosome with higher or lower value is selected.

Apply Crossover and Mutation

Parents that are chosen are randomly combined. This technique used for generating random chromosome is called crossover.

There exist two types of crossover,

(i) Single point crossover.

(ii) Multiple point crossover.

Test Case Optimization Using GA

Let's say a program has test suite T, now if one can make modification in the program P, suppose modified program is P', so in order to test program P' one can generate a prioritize sequence of test cases from test case suite T , on the basis of the line of code modified. Here the following genetic parameter will be used.

By using genetic algorithm this research provides techniques for test case prioritization. Suppose a program P, has a test suite T, with modification in the program P', to test program P', sequence of test cases can be generated from test case suite T, based on lines of code modified. Here the following genetic parameter will be used.

Fitness Function

The following objective function (fitness function) will be used.

Fitness value (F) = Σ {order * (number of modified lines covered by test cases)}

Crossover

Here one can use one point cross over with crossover probability $P_c = 0.33$.

Crossover Probability = Fitness Function of Chromosomes / ΣFitness Function

Mutation

Here we will use mutation probability $P_m = 0.2$. It means that 20% of the genes will be muted within a chromosome. For instance, the test cases with execution history. The practical implementation of genetic algorithm has been explained briefly. Table 2.3 explains which test case covers which line of code being tested. The test case with test case ID one covers the statements eight, nine, ten, eleven, twelve and thirteen like case. This helps to compare the number of modified lines with the above information and sort out which test case covers the most modified lines of code as in Table 2.4.

Each test case needs to be associated with its implicit properties such as code functions they parse through, within the developed code and the complexity of the tested code. Assume that the lines 5, 8, 10, 15, 20, 23, 28, 35 are modified and the modified lines of code covered by each test case are shown in the Table 2.5.

It shows the test cases that do not cover all the modified lines of code we use genetic algorithm. Genetic Algorithm is considered as the best search problem which overcome the problem in hill climbing which is supposed to be more efficient.

Table 2.3 Test Case Execution History of Genetic Algorithm

Test Case ID	A	B	C	Expected Output	Execution History
T1	30	20	40	Obtuse angle triangle	8,9,0,11,12,13
T2	30	20	40	Obtuse angle triangle	8,9,10,11,12,13,14,15,16,17
T3	30	20	40	Obtuse angle triangle	10,11,12,13
T4	30	20	40	Obtuse angle triangle	10,11,12,13,14,15,16,20,21,22

T5	30	20	40	Obtuse angle triangle	12,13,14,15,16,20,21,22

Table 2.3 (Continued)

Test Case ID	A	B	C	Expected Output	Execution History
T6	30	20	40		22,23,24,25,28
T7	30	20	40	Obtuse angle triangle	5,6,7,8,9,10,11,12,13,14,15,16,20,21,15,16,20,21,35
T8	-	-	-		
T9	30	20	40		5,6,7,8,9,10,11,12,13,14,15,16,20,21,15,16,20,21,35
T10	30	20	40		18,19,20,21,35
T11	30	20	40	Obtuse angle triangle	24,25
T12	30	20	40	Obtuse angle triangle	15,16,20,21

Table 2.4 Test Case Code Coverage

Statement	Test case 1	Test case 2	Test case 1	Test case 4	Test case 5	Test case 6	Test case 7	Test case 8	Test case 9	Test case 10	Test case 11	Test case 12
5							X		X			
6							X		X			
7							X		X			
8	X	X					X		X			
9	X	X					X		X			
10	X	X	X	X			X		X			
11	X	X	X	X			X		X			

Table 2.4 (Continued)

Statement	Test case 1	Test case 2	Test case 1	Test case 4	Test case 5	Test case 6	Test case 7	Test case 8	Test case 9	Test case 10	Test case 11	Test case 12
12	X	X	X	X	X		X		X			
13	X	X	X	X	X		X		X			
14		X		X	X		X		X			
15		X		X	X		X		X			X
16		X		X	X		X		X			X
17		X										
18										X		
19										X		
20				X	X		X		X	X		X
21				X	X				X	X		X
22				X	X	X						
23						X						
24						X					X	
25						X					X	
26												
27												
28						X						
29												
30												
31												
32												
33												
34												
35							X		X	X		

Table 2.5 Number of modified lines covered by the Test Case

Test cases	Number of modified lines
T1	2
T2	4
T3	1
T4	3
T5	2
T6	2
T7	5
T8	2
T9	4
T10	1
T11	0
T12	2

Now we apply Genetic Algorithm, on this data and one can represents this information in a matrix, the first column would be order, second column would be number of lines modified by each test cases, and then one can generate random number without repetition and put it in the following column, these pattern of random number would represent chromosomes and we would have chromosomes, e1, e2, …. and so on in the following column of the matrix and then we find the fitness of each chromosomes, find probability, perform selection and recommend which chromosomes to be taken into the population.

Now Genetic Algorithm is applied on the data where one can represent this information in a matrix. The first column would be in an order, second column would be number of lines modified by each test case. Then random number can be generated without repetition and put in the following column. These patterns of random number would represent the chromosomes and would have chromosomes, e1, e2,….and so on. In the following column of the matrix and then we find the fitness of each chromosomes, find probability, perform selection and recommend which chromosomes to be taken into the population.

On the basis of the random recommends the chromosome 1 which is represented as numbers.

(T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12)

Because the selected random number lies between 0-0.342. Second random number recommends the chromosome 2 which is represented as

(T2→T4→T6→T8→T10→T12→T1→T3→T5→T7→T9→T11)

Because the random number lies between 0.342-0.671.The third random number recommends the chromosome 1 which is represented as

(T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12)

Because the selected random number lies between 0-0.342. So now we have the following member in our mating pool:

T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12→T2→
T4→T6→T8→T10→T2→T1→T3→T5→T7→T9→T11
T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12

Now we will apply the one point cross over on these chromosome and will generate the new offspring.

T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12

T2→T4→T6→T8→T10→T12→T1→T3→T5→T7→T9→T11

T1→T2→T3→T4→T5→T6→T7→T8→T9→T10→T11→T12

When we apply one point crossover to the selected population then we get these offspring.

T1→T2→T3→T4→T5→T6→T7→T9→T11→T8→T10→T12

T2→T4→T6→T8→T10→T12→T1→T9→T11→T3→T5→T7

T1→T2→T3→T4→T5→T6→T7→T9→T11→T8→T10→T12

Suppose the crossover probability is 0.3, then 2 chromosomes are selected from the offspring one from the parents based on the fitness function value. As in Table 2.6, the near optimum solution is obtained on repeating the process certain fixed number of iterations and on repeating this procedure multiple times.

Table 2.6 Using Genetic Algorithm on the same data

Chromosomes	Fitness Value	Normalized Value	Cumulative Probability	Selection of Random Numbers	Recommandation
T1->T2->T3-> T4->T5-> T6->T7-> T8->T9-> T10->T11 ->T11->T12	196	196/573 =0.342	0.342	0.3	Chromosomes e1
T2->T4->T6 ->T8->T10-> T12- >T1->T3 ->T5->T7->T9 ->T11	189	189/573 =0.329	0.671	0.4	Chromosomes e2
T5->T6->T8 ->T9->T12 ->T1->T7 ->T11->T2 ->T4->T10	188	188/573 =0.328	1	0.2	Chromosomes e1

2.5 RTST TECHNIQUES

The retest-all approach which consumes large amount of time and resources will increase the cost of performing the regression testing[33][34]. The developer has to face the problem for the selection of the appropriate subset i.e. T' of the test T for rerunning on the program P' and this process is called as an RTST (Regression Test Selection Techniques). Here only the subset of the test case is selected by the selection techniques from the entire test suite.

RTST has to rerun each of the test cases from the test suite T on the program P' to select the test case T' that is equal to the test suite T. RTST techniques will use information from program P , modified versions P', test suite T, for selecting a subset of T for testing P'. This technique helps to reduce the testing effort compared to the retest-all approach.

Each test case selected by the RTST techniques from the test suite behave differently in the modified and original software versions [35]. Precision and efficiency of RTST techniques are related to the granularity level where these techniques operate from the safety point of view for the RTST techniques (Rothermel et al 2002).

It guarantees that the subset T' contains all test cases that reveal the regression faults occurring in P'. The technique works in a higher abstraction level. RTST must be more cost effective than the cost for performing, rerunning the subset of the selected test case should be comparatively less than the cost for rerunning the entire test suite.

In the cost models for the regression testing cost terms depend on some specific scenario. The study prove that the techniques which are existing now will be more cost effective and studies are performed using some limited size of subjects [36].

Test suite requiring human interaction, savings should account human effort to be saved. It is too expensive for using the precise technique on larger systems. The existing safe techniques are not so cost effective when it is applied to the large software and the efficient techniques will be more incomplete and achieve little savings in the testing effort.

A new algorithm for RTST handles the features of object oriented language that is more precise and safe for the larger systems. There are two phases in the algorithm. They are partitioning and selection[37]. High level of the graph representation for the program P, P' is created in partitioning phase. The analysis goal identifies based on the information in the changed classes to interface the program parts P, P' to do further analysis. Selection phase provides a denoted graph in the algorithm selection phase towards identifying program parts P, P'. The analysis goal is for identifying, based on the information in the changed classes which interface the program parts P, P' for further analyzing.

A detailed graph is built in the algorithm selection phase for the identified program parts P, P'. The graphs are analyzed for identifying the differences between programs which selects for the rerun of the test cases that are in T, which traverse these changes. Although, these techniques are defined in the Java language and it can also be adapted for object oriented language.

This technique can handle the object oriented features and it does not require analyzing the entire system and requires only the identified partition at the initial phase. The advantage of our partitioning can compute the simple dependences and it postpones that expensive analysis to the second phase.

2.5.1 Types of Phases in RTST Techniques

In this technique we combine the RTST effectiveness which is precise, it may not be efficient for the larger systems with the efficiency of the techniques which works on a higher level and may be imprecise. This can be done using the two phase approach which performs an initial analysis at a high level which identifies the system parts that has to be analyzed further, in-depth analysis of the parts identified that selects those test cases which are in T are rerun. In partitioning phase, towards identifying aggregation, technique is used to analyze those program and the technique uses the relationship between the interface and the cases (Walcott et al 2006). These interfaces have changed syntactically, towards identifying the program parts that may get affected by those changes between the programs P, P'.

The output of this phase will be the subset of those interfaces and the classes in those programs. In selection phase this technique which the input as the partition that are identified in the first phase. The test selection at the edge level selects the test case just by analyzing the changes and the coverage information at a level of flow of the control between the statements (Bible et al 2001). Due to the partitioning that is performed at the phase one, expensive analysis, low level is performed to the small fraction in the whole program (Walcott et al 2006). Since a partial portion of the programs is analyzed, it is performed under safe assumptions because, the partitioning will identify all the interfaces and the classes whose behavior changes due to modifications to the program P, edge level technique which is used in selection phase will be safe.

2.5.1.1 The Partitioning Phase

This performs the high-level analysis for program P and the modified program P' and the relationships between the interfaces and classes from the program.

2.5.1.2 The Syntactic Change Accounting

Program changes are classified as two groups. They are changes in statement level and changes in declaration level. The former consists of addition, modification, deletion of executable statements. This change is handled easily by the RTST, each of the test case which traverse the nearly modified part of the code has to be executed measures their coverage of the test suite T when the program P is tested. For the program entities, coverage is computed like edges and statements. For each of the test case t which is in T, the information is recorded on which entities to the program P that are executed by the test case t coverage information, using the coverage tools, is collected automatically and is represented as the coverage matrix, like one row for each entity, one column for each test case (Jeffrey et al 2007). This technique is applied to the medium and large systems.

A tool DejaVu is presented which implements this technique on the set of those subjects. The study investigates the percentage of programs under test and its selection by the partition technique and how the overall RTST cost gets affected (Li et al 2010, Kim et al 2002). It has been investigated the gain in precision during the operation of this technique at high abstraction level and the overall savings achieved in the process of regression testing. In declaration level, it consists of modification in a declaration. This can be variable type with modification, removal or addition of any method, the inheritance relationship modifications, the change of the type in the catch clause, the modifiers list change. These changes may be problematic for the RTST than the system level changes because the program behavior can be affected in directly declaration level changes have complex effects when compared to the changes in the statement level. The improper handing leads, RTST technique unsafe, imprecise or both (Yoo et al 2007).

2.5.1.3 The Partitioning Algorithm

The input for the algorithm is set of the syntactic changed types in C, two Interclass Relation Graph's, original version, modified versions of the program.

Step 1: Add the partition part which involves the changed types. Step 2: Add each

type to a temporary set.

Step 3: Add the types in the temporary set of partition. Step 4: Return

partition.

2.5.1.4 The Selection Phase

The changed information is computed by analyzing those types that are identified in the first phase, and then test selection is performed by matching the changed information computed with the coverage information.

2.5.2 Performing Test Case Selection

The DejaVu architecture consists of three components, InsECT, DEI, and Selector as in Figure 2.2. Java is used in developing the InsECT which is an extensible, modular, coverage analyzer. By using this information on edge, coverage of the programs is gathered when it is executed against the test suite. The two phase techniques are implemented by using Dangerous Edge Identifier (DEI) and selector.

Three medium to large sized programs like JABA, DAIKON, JBoss are used as subjects from which five consecutive versions are extracted stimulating the way the regression test occurs. The DejaVu is modified in the experimental design so that identifying the partition at phase I makes partition to skip at phase II by which all the test cases are selected by using the partition (Prem Jacob and Ravi 2013).

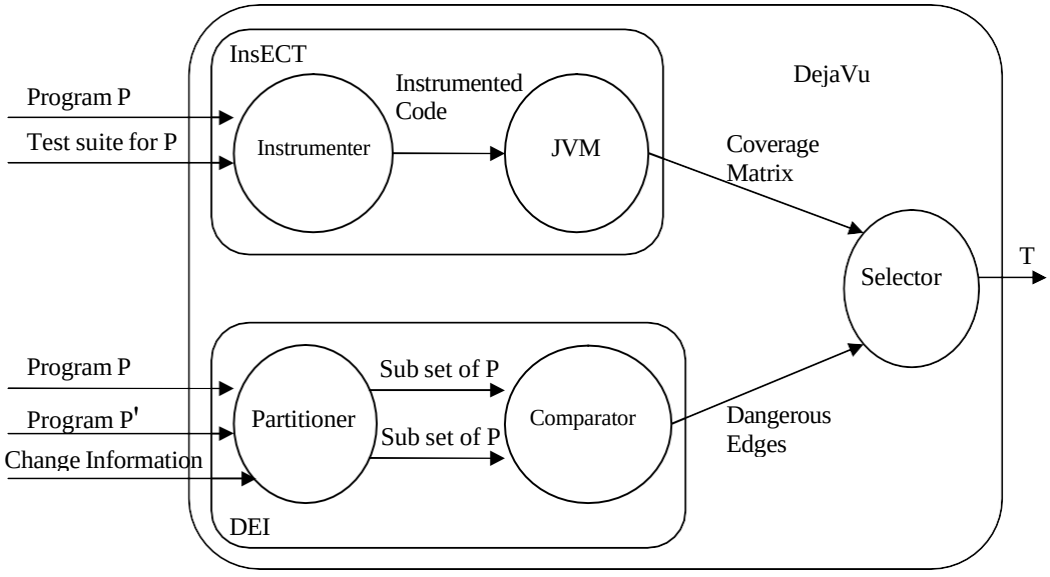

Figure 2.2 DejaVu regression test case selection system

2.6 TABU SEARCH

The Tabu search technique is a meta-heuristic technique. This is established successfully in real world applications such as traveling salesman problem. It is obtained appropriate for test case generation issues in software testing. In that only few outcomes have been distributed with comparatively few samples. It is established with many samples and all input domain data types. A brief explanation about the Tabu Search Algorithm, initially initialize the current solution (CUS) and store in CFG (Jacob et al 2013). Then add the current solution to Tabu list. Select the node to be cover up and calculate the zone candidates for every candidate. If the candidate value in node n is less than the CFG in node n then store the candidate in CFG. If sub goal node is not covered, then add current solution to Tabu list. Otherwise destroy the Tabu list. If current solution is used up then add the current solution to Tabu list LT and apply backtracking process (Jacob et al 2013).

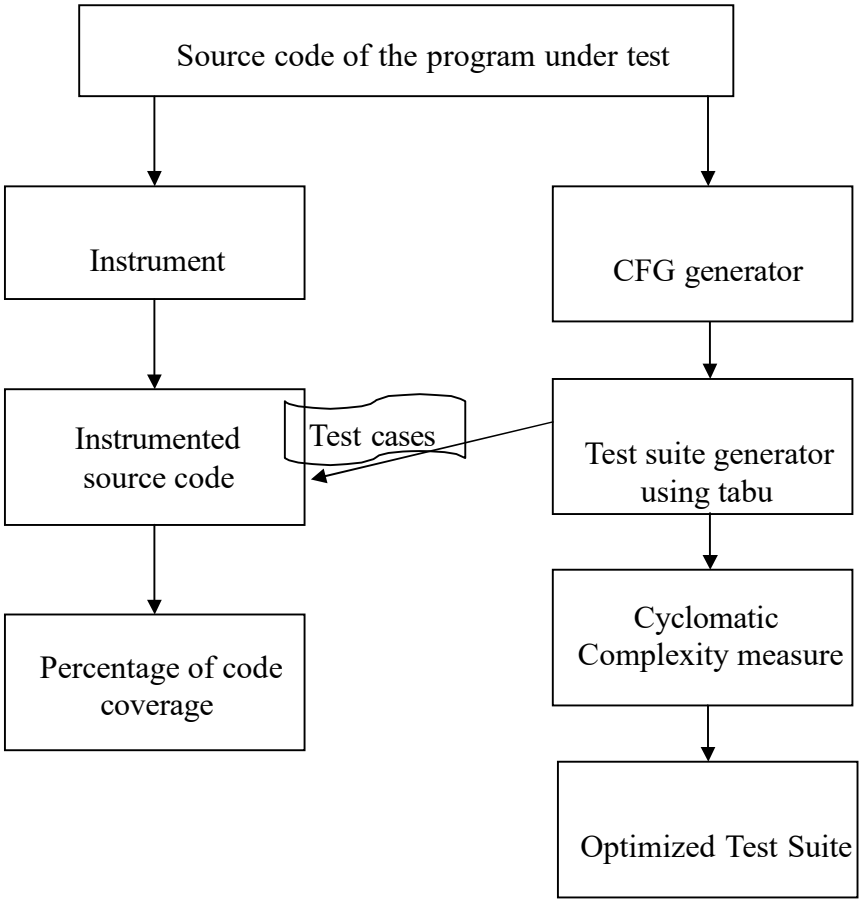

Figure 2.3 Architecture of Code Coverage using Tabu Search

Cyclomatic complexity, a software structural metric is used to measure the program complexity, but usually it is measured by using Control Flow Graph. The cyclomatic complexity of a planned program is shown as:

$$C = nE\text{-}nN+2D$$

Here C is the cyclomatic complexity, nE be the number of edges of the graph, nN be the number of nodes of the graph and D be the number of disconnected components. It gives the essential number of lower bound on the numerical of test cases towards achieving the branch coverage. The quantity of test effort is calculated by the cyclomatic complexity, with which there are measurable smaller quantity test cases. The coverage can be attained with fewer number of test cases. Various steps are followed to create the test cases to obtain the code coverage, to find its best set of optimal test cases for regression testing. The first step in the source code of the program is taken as input to the control flow graph generator.

The different steps in the automated test cases are:

1. Control Flow Graph generator which is used to generate the Control Flow Graph by taking the source code of the particular application as input.
2. Amount of test effort is calculated in the second step by using cyclomatic complexity measure.
3. Control Flow Graph is analyzed and the branching condition of the source code information is extracted.
4. With the help of Tabu search technique test cases are generated for every condition in the source code from the input field of the variables occupied in the branch condition.
5. With the use of cyclomatic complexity measure has to find fulfillment of number of test cases.
6. In order to check the branch coverage the generated test cases are practical to the instrumented source program.
7. Finally by finding the suitable test cases from a test suite for the given application source code under test.

Tabu Search Algorithm

```
CUS = value;   C =
CUS;
TLS.add(CUS);
S = A, B, C,
do
FIND ADC
for(int i = 0; i<= ADC.length; i++)
{ if(CAN<Ca)
{ C=Ca;
}
}
if(sgn!=S)
{
TLs.add(CUS)
}
else
{
delete (TLS);
}
Boolean b = true; if(CUS=b)
{ T1T.add(CUS);
T1T.bt();
}
While (sign !=S && iteration. Length<MAXIT) break
```

Parameters:

CUS – current solution; C-
CFG;

TLS – Tabu List Software testing; ADC –
Adjacent Candidates;

CAN – Candidate's value in node n; Ca – CFG
in node n;

A, B, C – covered sub goal node; Sgn –
Subgoal node;

T1T- tabu list LT ;

Pseudo code of Tabu Search Algorithm

Start

Step1: Current Solution should be initialized.

Step2: Save the Current Solution in Control Flow Graph Generator. Step3: In Tabu list add current value.

Step4: Choose sub goal nodes which have to be covered. Step5: Has to do calculation with the neighborhood candidates. Step6: Each and every candidate repeat the step 5.

Step7: if (candidate value in node n <CFG in node n) then Step8:
Again Save the candidate in CFG

Step9: end if

Step10: end for

Step11: if the sub goal node is not covered, then have to do following operations.

Step12: Again Add Current Solution to Tabu list LT. Step13: else Delete Tabu list LT.

Step14: end if

Step15: Choose a sub goal node which has to be covered and Current solution.

Step16: if Current Solution is deleted, then

Step17: Now, Add Current Solution to Tabu list LT after completing this process.

Step18: Now, Apply a backtracking process.

Step19: The new Current Solution and may be new sub goal node. Step20: end if

Step21: while (NOT all nodes covered AND number of iterations <MAXIT)

Step22: end

2.7 CLUSTERING BASED TECHNIQUES

Dataset is divided into mutually exclusive group where the members of each group become "close" to one another and different groups are as "far" as possible from one another where all other variables are used to measure the distance. Clustering is an unsupervised learning process with no predefined classes. Cluster analysis means similarities are found between data, based on it, characteristics and grouping similar data object into clusters. High quality clusters denote the goodness of the clustering method where it will be with high interclass similarity measure used by the method. Its implementation determines to explore the quality of a clustering result. There is also separate "quality" function which measures the "goodness" of a cluster.

In the test case prioritization, the total number of comparison required for pairwise comparison is O(n2) comparisons (Elbaum et al 2004). In case while redundancy makes pairwise comparison very robust the applications to test case prioritization gets discouraged by the high cost incurred. Approximately 100, the maximum number of comparison can be made by a human (Li et al 2007). The effectiveness is reduced due to growing inconsistency. For instance, suppose there are 1000 test cases to prioritize then 4,99,500 pair wise comparisons would be required and it is unrealistic to expect reliable responses from a human tester for large number of comparisons. Techniques are used to prioritize the clusters of test cases, such as clustering based prioritization.

2.7.1 K-Means Clustering Criteria

There are two methods of clustering such as data can be arranged as a group of individuals or as a hierarchy of groups. It can thereafter be established that whether the data group belong to some preconceived ideas or suggest new ones (Boris Beizer 1990). Cluster analysis groups data objects into clusters such that objects belonging to the same cluster are similar, while those belonging to different ones are dissimilar. Clustering techniques could be categorized into modes Partitional or Hierarchical.

2.7.1.1 Partitional Clustering

Partitions of the data are constructed by a partitional clustering algorithm with a given data base. Here each cluster optimizes a clustering criterion such as minimization of the sum of squared distance from the mean within each cluster (Roongruangsuwan and Siripong 2005). Partitional clustering enumerates all possible groupings to find the global optimum. That is why the complexity of partitional clustering is large.

2.7.1.2 Hierarchical Clustering

Hierarchical algorithm create a hierarchical decomposition of the objects. They are either agglomerative (bottom-up) or divisive (top- down).

a) Agglomerative algorithm begin with each object which is a separate cluster itself, this merge into groups in accordance with the distance measure (Tsai et al 2005). When all objects are in a single group, the clustering may stop. These methods generally follow a greedy like bottom-up merging.

b) Opposite strategy is followed by the divisive algorithm where (Angelo Susi et al 2009) all objects are started with one group and the group is divided into smaller groups until each object falls in one cluster [13]. The data objects are divided into disjoint groups at every step through divisive approaches and the same pattern is followed till all objects fall into a separate cluster. Divide and conquer follow the same approach as followed by the division algorithm.

2.7.2 The K-Means Clustering Method

$$E = \sum_{i=1}^{k} \sum p \in C_i |p - m_i|^2 \quad (4.1)$$

Where C_i will be the clusters, P will be the point of the cluster C_i, m_i will be the mean of the cluster C_i. The mean of the cluster is represented as a vector for each of the attribute, mean values for the data in the cluster, the input parameter will be the total number of the clusters k. As the output of the algorithm will return the means for each cluster C_i. The distance is usually measured in Euclidean distance as Equation (4.1). Proximity index and the optimization criteria have no restrictions which can be represented according to user preference or the application.

Algorithm

Step 1: The initial centers are selected as k objects. Step 2:

Assign the data objects in the center.

Step 3: The center of each cluster are recalculated.

Step4: Repeat the steps 2-3 unless the data object distribution in the clusters is not changed.

It has to be analyzed whether the clustering technique can make the test case prioritization happen for the test suites whether the prioritization of the test case makes the rate of fault better which is detected from these test suites.

2.7.2.1 Prioritization Measure Based on Clustering and Efficiency

The coefficient are read and the roots are decided immediately once the prioritization technique on clustering to the problem of quadratic equation is applied.

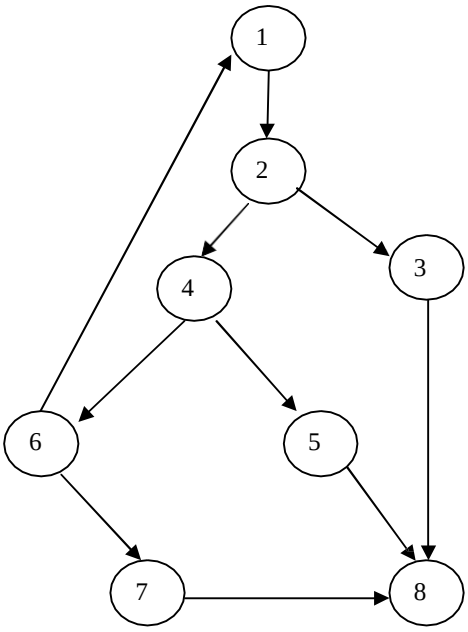

Figure 2.4 The flow graph for quadratic equation problem

Procedure

1. Initialize the coefficient
2. Calculate the roots
3. if root less than zero
4. Imaginary roots
5. if root equals zero
6. Calculate root1

   Assign root 2 equals root1
7. if root greater than zero
8. Calculate root1, root2
9. end

2.8 AVERAGE PERCENTAGE OF FAULTS DETECTED

The effectiveness of the new test case orderings is determined by the Average Percentage of Faults detected (APFD) metric (Malishevsky et al 2006, Elbaum et al 2001, Rothermel et al 1997). APFD metric which was developed by Elbaum et al (2006) is used to quantify the goal of increasing a subset of the test suite, rate of fault detection and it measures the rate of fault detection per percentage of test suite execution on an average. The weight average of the number of faults detected during the run of the test suite is considered to calculate APFD. Depending on the size of the test suite and the time that each case takes to run, the existing technique may take a long time. The test cases can be recorded by the testers through the use of an effective prioritization techniques to obtain an increased rate of fault detection (Elbaum et al 2001). A new regression test suite prioritization algorithm is implemented by the technique, where the test cases are prioritized with the objective of maximizing the number of faults that are likely to occur during the constrained execution.

The number of faults that each test case detect and the time taken to detect the faults can be traced as it is assumed that the desired execution time to run the test cases is known in advance.

The value obtained for prioritized case is more than the previous method and more effective which is shown by the comparison between prioritized and non-prioritized test case. The regression testing is improved by using the test case prioritization. APFD metric is used to analyze the prioritized and non-prioritized test case.

Let T be the test suite under evaluation

M -> Number of faults contained in the program under test P n -> Total number of test cases

TFi -> Position of the first test in T that exposes fault i.

APFD = 1 – [(TF1 + TF2+ ..... + TFm)/nm] + 1/ 2n

APFD Algorithm

Input: Test suite T, number of fault detected by a test case f, and cost to run each test case $T_{cost.}$

Output: Prioritized Test suite T'.

1: **begin**

2: set T' empty

3: **for each** test case t ε T **do**

4: calculate average fault found per minute as f/T_{cost} 5: **end**

for

6: sort T in descending order based on the value of each test case 7: let T' be T

8: **end**

APFD can be calculated only when prior knowledge of faults is available as shown by the formula for APFD.

3. RESEARCH METHODOLOGY MUTUAL METHOD

3.1 PROPOSED WORK

How to reuse the existing test suite for the modified program is an important issue in regression testing. Selective retest (Ngah and Amir 2012) and retest-all are the two main regression testing strategies. The entire existing test suite on the modified program is run again by the retest-all approach. In theory, all modifications parts in the modified program are exercised by the retest-all approach and so it is considered as safe approach. Due to the time and resources needed, it is impractical to use it for large software systems. Selecting a subset of the existing test suite and retesting only the relevant part of the modified program reduce the time required to retest a modified program and this is possible in selective retest techniques.

Two issues in selective retest techniques have been identified by Rothermel and Harrold (1999).

i. Issue of how to select test cases from existing test suite.

ii. The issue of identifying where additional test cases may be required.

The proposed model presented in this research work tackles both the issues. Prevention of inefficient code coverage, reduction of testing time and reduction of testing cost is done by using proposed novel method which is mainly known as mutual method. For testing a modified program, the test sequence cannot be selected randomly by the proposed method. Not only the primary metric (i.e. fault detection rate, time and cost) will be selected by the selected optimal test sequences, but also novel metric (i.e. suitable child generation at mutation level) will be considered. The test case sequence is prioritized by using the new metric which is based on their occurrence of suitable child test case generation. The mutual method used which increases the efficiency of the ranking algorithm at the initial stage.

3.1.1 Proposed Architecture

The proposed architecture is explained in this chapter in detail. Based on the requirements of the client, the initial version program is modified with some functions, with the test cases which is already generated, the modified program version in now retested. As a result, the modified program version is now retested with already generated test cases. Due to some additional functions, some additional test cases are needed for the modified program version. At the time of testing, the new test case could not be generated manually. So the testing of the modified program

is done with already generated test cases. The optimal test cases are selected by the new mutual method for testing the modified program. The proposed algorithm are discussed below.

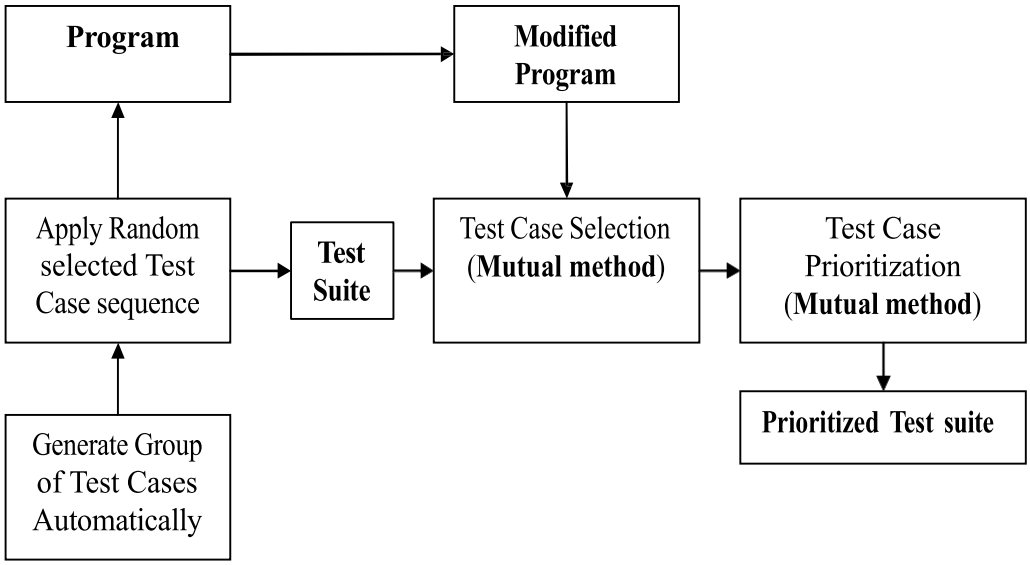

Figure 3.1 Proposed Architecture

3.2 TEST CASE SELECTION USING MUTUAL ALGORITHM

The working of the proposed test case selection algorithm is based on Modified Genetic Algorithm. The core operation of Genetic Algorithm is utilized. The Genetic Algorithm differs from the new child test cases generation. Two offspring of two parent test cases are used to generate new child test cases. Proposed algorithm is used to select the parent test case. The input parameter frequency i.e., the frequency of every test case input parameter obtained through their mutation

occurrence at the time of new child test case generation, use normal Genetic Algorithm for initial program version.

3.2.1 Pseudocode for Mutual Based Selection Algorithm

Input : Test case sequence t_i …..t_n **Output :** Optimal sequence S_i…..S_n Procedure mutual ( )

begin

for each program version P

begin

for each test suite T consists of N test cases $t_i \in T$

$T_i \in I_j$ and function F_k , where I_j is an input parameter

repeat begin

Initialize test suite and population size, where each population represents an optimal solution, i.e.population = sequence (test sequence, population size) Devise the fitness function

F_n= min (cost, time) and max (code coverage, $F(I_i)$) , where $F(I_i)$ is frequency of input I_i

Devise the selection criteria update the frequency of I_1 and I_2.

Perform genetic operation child=

mutation(child) crossover ( )

Update the population

population= population U child

end

end repeat

end

3.2.1.1 Procedure for Crossover Operation

Procedure crossover ( ) begin

for (the number of test case times) if (until

the point of cross over)

new matrix first column = elements of selected chromosomes end if

else

do

until

initialize the n;

if (n crosses the number of test cases) then set n to

zero

set check true;

for (j from 0 to current index)

if (current chromosomes element is already in the new matrix) set check to false

break;

end if
end for

end while if check is not true

new matrix first column=element of selected chromosomes end else

end for

```
//Second Child
for(the number of test case times) if( until the
point of cross over)
new matrix second column=elements of selected chromosomes end if
else
do
until
initialize the n;
if(n crosses the number of test cases) then set n to
zero
set check true;
for (j from 0 to current index)
if (current chromosomes element is already in the new matrix) set check to false
break; end
if end for
end while if check is not true
new matrix second column = element of selected chromosomes end else
end for
end
```

3.2.1.2 Procedure for Mutation Operation

Procedure mutation ( ) begin

srand (time(NULL)); generate first

random number do

until

set check true

generate second random number;

if (the two random numbers are same) set check to

false

break;

end while if check is not true

//swap

Swap the execution order of selected random number for first child

Swap the execution order of selected random number for the second child update the population

end

Basically, Genetic Algorithm is based on the survival of the fittest. Optimal solution is found by using GA. Exhaustive search procedure is involved in GA. Best results are obtained when GA is used to solve the optimization problems. The domain of input values and those which satisfy the desired goal of testing is done by using GA. Further searching of the desired goal is done by using the inputs which generate new inputs when combined together.

Crossover and mutation are the two primary operation based on which evolution is done. GA is not a simple random search despite the randomized nature of GA. To generate new solutions with improved performance, it utilizes old knowledge which is held in parent population. Sub optimal solutions are discarded and the best solutions are retained. Set of individuals which are generated randomly are included in the population. One of the main tasks which affect the performance of the next generation significantly is the selection of the initial generation.

The individual is represented by each test case input parameter binary stream is represented as a chromosome. Crossover and mutation operators are used to modify the chromosome which are composed of genes (input parameter). To compare the performance of each individual, fitness of each individual is calculated. The sequence which have higher fitness value for the individual represents sub optimal solution. New members are produced by using the process of mutation and crossover.

The most commonly used operators are crossover and mutation. At the individual level, crossover is operated in binary form. Two individuals are selected in crossover and a point along the bit string is selected and swapped. Random bits are changed by mutation in the binary string. The state of the gene is changed from 0 to 1 or vice versa. New population are generated once crossover and mutation are over. For that population fitness value is calculated, which determines the survival of parents and offspring.

The best solution of a problem is calculated by using the fitness function. Higher fitness value is obtained by an individual which is near optimal solution than the other individual. Every fitness value is used to represent a search space. Two individuals are selected from a generation by a selection operator to become parents for the recombination process. Based on the fitness value, this selection is done. The population size selects the GA in a random manner sometimes for testing the modified program, suitable test cases are not selected by the random function. The invalid child test cases may also be generated by the selected test case frequently. The test case input parameter frequency metric is utilized to avoid this issue. While testing a modified program, continuous monitoring of each test cases input parameter is done to obtain the mutation frequency. Using this frequency, the parent test cases are selected but not in a random order. The fitness function of GA involves the application of this frequency metric.

3.2.2 Operations on Mutual Method

3.2.2.1 Selection

Based on the fitness of an individual, how they are chosen for mating is determined by applying a selection scheme. Fitness can be defined as a capability of an individual to survive and reproduce in an environment. For starting a new generation, selection generates the new populations from the old one. To determine the fitness value, each chromosome is evaluated in present generation. For the next generation, better chromosomes are selected from the population by using the fitness value.

3.2.2.2 Crossover or Recombination

Application of the crossover operation is done to the selected chromosomes after selection. Swapping of genes or sequence of bits in the string between two individuals is involved. Till the next generation has enough individuals, this process is repeated. Application of mutation operator is done to a selected subset of the population in sequence order after the crossover.

3.2.2.3 Mutation

To introduce new good traits, the chromosomes are altered in small ways by the mutation. Diversity is brought into the population by applying mutation.

3.3 TEST CASE PRIORITIZATION USING MUTUAL ALGORITHM

Fault detection algorithm, average fault detection algorithm, coverage based techniques and model based techniques are the existing test case prioritization techniques that are briefly discussed in literature survey. Due to the processing of specific primary parameter, those techniques are still lacking to achieve the full code coverage within the minimum time.

3.3.1 Pseudocode for Mutual Based Prioritization Algorithm

Input: Optimal test case from mutual method.

Output: Ordering of test cases. **Procedure**

prioritization ( ) begin

let T be the test suite having N test cases, for each test case represented by t_i , $t_i \varepsilon N$

mutual( ) // execute mutual method

let M be the number of optimal test case selected from mutual method for every optimal test case ε M, having an input parameter I_i and a function F_k

repeat

begin

assign frequency value n_i to each I_i mutation(I_i)

if check (I_i) covers the function

update the frequency of n_i of input parameter order the test

cases

until all the test case are ordered

end end

Initially, a number of test cases is contained in a test suite. A number of input parameter and testing functions is contained in every test case. The test cases are applied into modified program versions. The frequency of test case input parameter is increased, when tested new branch condition or function. Then, the test cases are ordered by this frequency as a higher frequency.

4. RESULTS AND DISCUSSION

4.1 OVERVIEW

To evaluate the effectiveness of the test suite reduction and test suite prioritization algorithm, the various applications are analyzed in this chapter. The prototypes of the reduction and prioritization algorithm are implemented to investigate the effectiveness of the algorithm. Oops concept is used to write both the prototypes. As both prototypes use the same data structures, much of the code is used in both prototypes.

Table 4.1 List of experimental parameters

Parameter	Value
Program Version	5.0 (Modified version)
Test Suite	100
Group of test case	13,585 (approximately for single test suite)
Population Selection	1. Unmodified Program: Random 2. Modified Program: Sequence
Number of iterations	100
Population Size	10
Crossover probability	0.5
Mutation Frequency	Depend upon the input parameter mutation at the software testing time

For the analytical comparison of the selection techniques and test case prioritization techniques in regression testing, a framework is presented from the theoretical issues that are relevant to the test selection of the regression testing. The existing techniques are compared by using the proposed framework.

Application of test suite reduction prototype for modified program to each of 1000 test suites is done to evaluate the test suite reduction algorithm. To acquire a test case requirement coverage report, space was run for each test in the union of all test suites (13,585 test cases). In each of the test suites, execution of algorithm with the coverage information for all test cases was done. The population size is varied at initial version and modified version. At the initial version, the input parameter frequency is zero and due to this the initial version uses random function to select the test cases. As the test cases are ordered as mutual high frequency value, the sequence order is followed to select the test cases. Selection and prioritization algorithm parameters are the basic properties of the proposed test case that are shown in Table 4.1.

4.2 EXPERIMENTAL SETUP

4.2.1 Test Case Selection and Prioritization

The details of a fragment of single test suite are shown in the Table 4.2. Five test cases are included in the test suite and number of input parameter and testing functions are included in each and every test case. The input parameter frequency of five test cases after testing the source code, is shown in Table 4.3.

Table 4.2 Fragmentation of a single test suite

Test case ID	Number of input parameter and functions
T1	I=5, F=10
T2	I=6, F=15
T2	I=8, F=25
T4	I=6, F=10
T5	I=4, F=12

Table 4.3 Fragmentation of frequency level of input parameter

Test case ID	Frequency of input parameter
T1(I1,I2,I3,I4,I5)	T1(5,6,7,8,9)
T2(I2,I2,I3,I4,I5,I6)	T2(8,16,12,13,21,11)
T3(I1,I2,I3,I4,I5,I6,I7,I8)	T3(2,17,14 ,25,28,11,34)
T4(I1,I2,I3,I4,I5,I6)	T4(25,10,11,12,18,23)
T5(I1,I2,I3,I4)	T5(15,16,17,18)

The test cases mutation for new child generation gives rise to the input parameter frequency. Due to high cost and more time, this frequency is used to select the proposed algorithm fitness function from the test cases. As the test cases such as input parameter are mutated and tested by modified program, the test case 3 and test case 4 is suitable for testing the modified programs. The modified program utilizes the generated new test cases input parameter frequently. From the given test cases, i.e. every test cases input parameter generate new input parameter then it cover new branch condition or untested source code from modified program, the test case T3 and T4 is the optimal test case sequence. To measure the code coverage of modified program, the frequency value is utilized. Selected optimal test case sequence is shown in Table 4.4.

Table 4.4 Selected optimal test case sequence

Test case ID	Input Parameter
T3(I1,I2,I3,I4,I5,I6,I7,I8)	T3(2,17,14 ,25,28,11,34)
T4(I1,I2,I3,I4,I5,I6)	T4(25,10,11,12,18,23)

Table 4.5 Fragmentation of proposed and existing algorithms code coverage

Test suite ID	Test suite size	ACO code coverage (%)	Genetic algorithm code coverage (%)	Tabu search code coverage (%)	RTST code coverage (%)	Mutual Method code coverage (%)
TS1	800	40	65	70	75	95
TS2	1300	55	75	85	80	98
TS3	1400	57	78	88	89	99
TS4	1330	50	56	72	80	98
TS5	1226	43	45	60	70	96

The code coverage of proposed and existing algorithms are shown in Table 4.5. Minimum percentage of source code is used to test the existing algorithms. All modified source code is covered by the proposed algorithm at the initial stage.

Table 4.6 Prioritized test cases using Mutual Method

Test case ID	Input Parameter
T3	T3(2,17,14 ,25,28,11,34)
T4	T4(25,10,11,12,18,23)
T2	T2(8,16,12,13,21,11)
T5	T5(15,16,17,18)
T1	T1(5,6,7,8,9)

The Table 4.6 shows the prioritized test cases based on their parameter of frequency of input parameter where the test cases are ordered with following functions,

Prioritized Test Suite = Max (Fre(I))

Where Fre(I) represents the frequency level of test case input parameter.

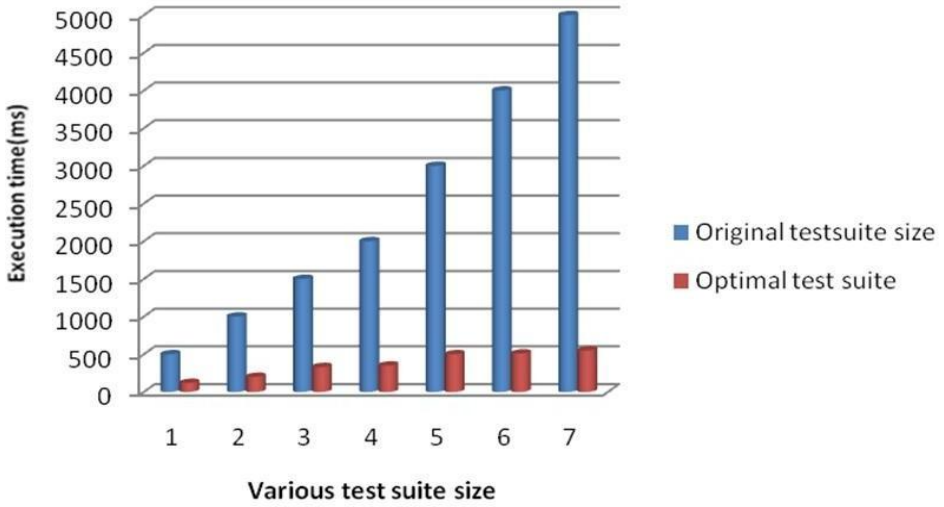

Figure 4.1 Original test suite size vs. Optimal test suite size of Mutual Method

Original test suite sizes that are reduced by the proposed algorithm are represented by a blue cylinder in the cylindrical graph shown in Figure 4.1, optimal test suite sizes that are represented by red cylinder are generated by the proposed algorithm. The largest test suite is 550, the smallest test suite is 120 test cases and average size of reduced test suites is 80.0, the standard deviation is 4.44, due to test case requirements coverage in the smaller test suites and lack of test cases with potentially high contributions, this trend occurs.

Table 4.7 Comparison details of proposed vs. existing algorithms

Parameter	ACO	Genetic Algorithm	Tabu Search	RTST	Mutual Method
Average Iteration level	200	150	144	100	45
Average Optimal test sequence size	600	555	402	389	202
Average Fault Detection rate	190	226	280	230	500
Average Code coverage	200	256	300	320	500

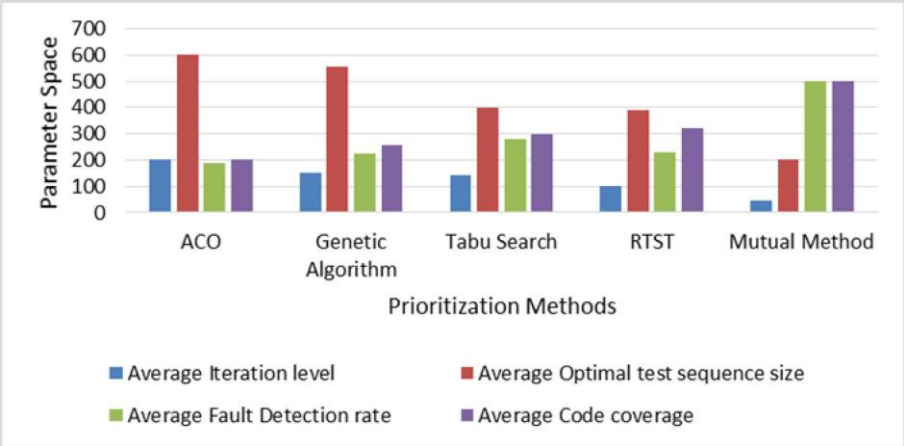

Figure 4.2 Comparison of proposed algorithm with different parameters

The Figure 4.2 shows comparison of mutual method with different parameters. When compared with other algorithm the proposed mutual method requires by average only 45 iterations to arrive an optimal solution. The mutual method covers the code coverage efficiently when compared to other algorithm and also detect the fault detection rate efficiently.

4.3 INTERPRETATION OF RESULTS

The following Figure 4.3 shows the comparison of mutual method with different approaches. To detect the faults from software under testing with the parameters test suite and fault detection rate, the comparison is made between the proposed mutual algorithm and other existing techniques such as fault detection algorithm, average prioritized fault detection algorithm and clustering based techniques. The test suite size is indicated by the x-axis and the number of faults detection rate is indicated by the y-axis. When compared with the existing algorithms, it shows that proposed algorithm is detected with maximum faults.

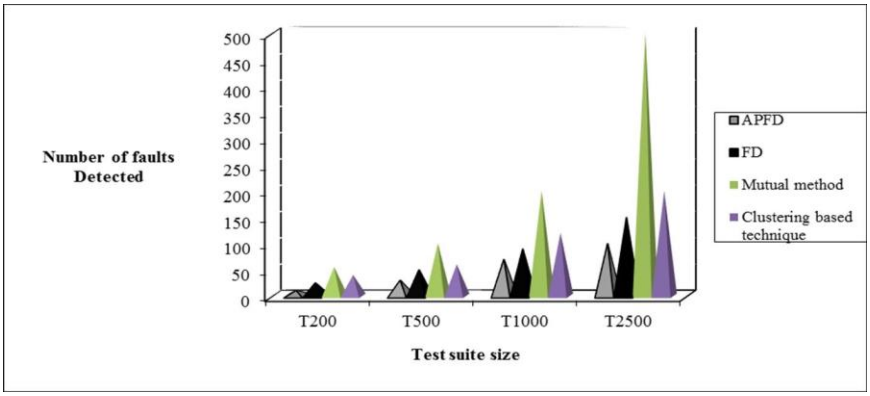

Figure 4.3 Test suite size vs. Number of fault detection

4.3.1 Time vs. Mutual Algorithm

Proposed test suite optimization algorithm and the previous existing optimization algorithms are compared which is shown in Figure 4.4 the maximum times are covered by the frequently used test case parameter and their functions, it takes very less time for the proposed algorithm to list the given modified program. Huge time is taken by the remaining algorithm. Till the iterations are completed, the testing continues as the genetic algorithm depends on the iteration level. The same drawback is seen in Tabu Search and ACO.

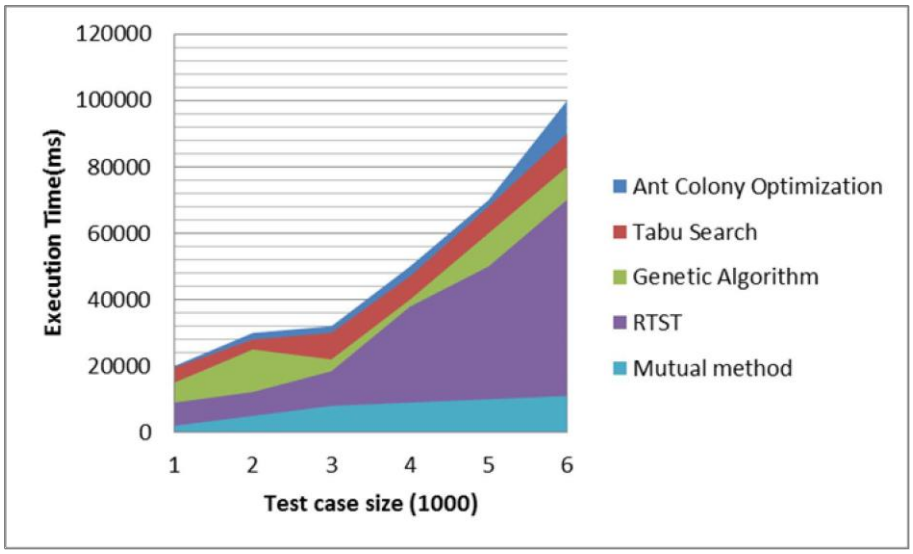

Figure 4.4 Time vs. Comparison of test case optimization algorithms

4.3.2 Cost vs. Mutual Algorithm

From the Figure 4.5 it is inferred the proposed mutual test case selection and prioritization algorithm reduces the cost more than 50% near to next method. The utilization of cost for proposed and existing algorithms is shown in the Figure 4.5. The proposed mutual test case

selection and prioritization algorithm which reduces the maximum cost at the time of software testing.

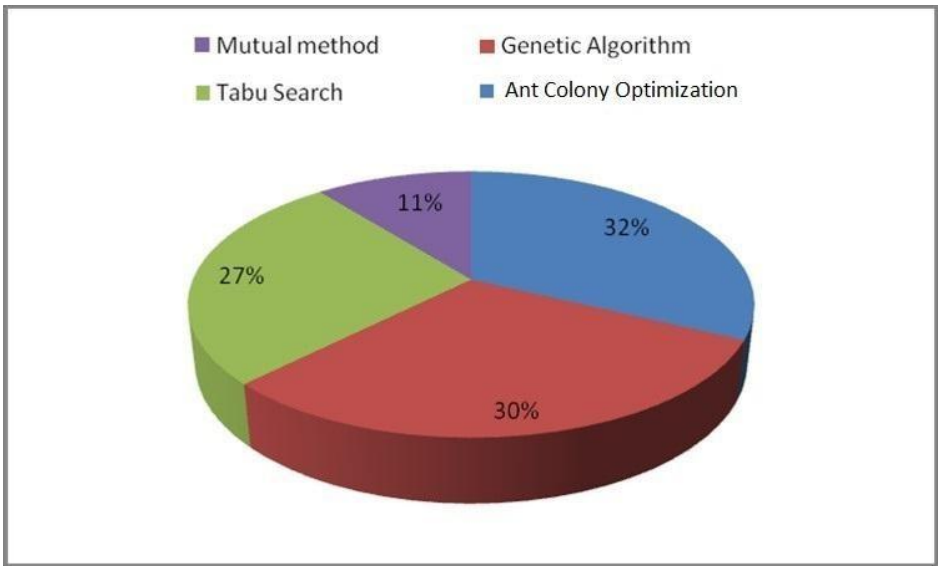

Figure 4.5 Cost vs. Optimization algorithms

4.3.3 Code Coverage vs. Mutual Algorithm

The code coverage of proposed algorithm and existing algorithms is shown in Figure 4.6. The size of modified program is indicated by x-axis and the level of covered lines is represented by y-axis through proposed and existing algorithms. When compared with other existing algorithms, the proposed method is covered with maximum lines.

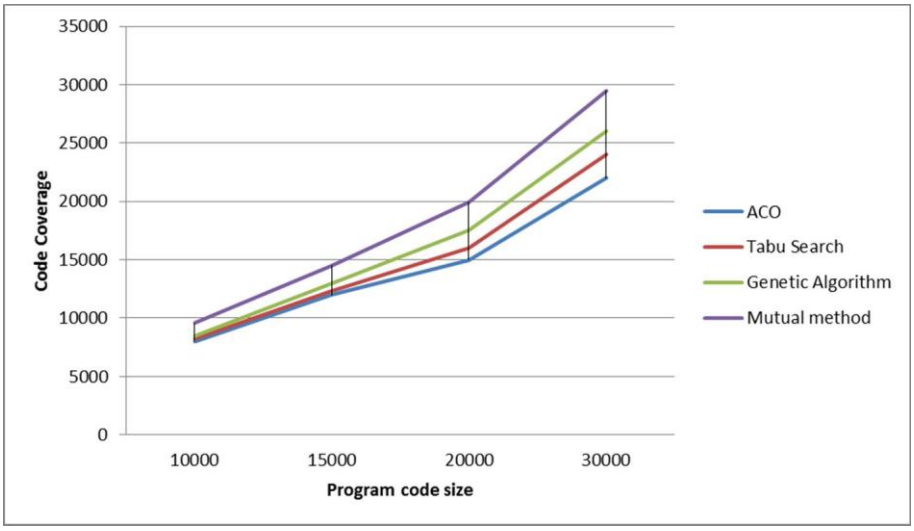

Figure 4.6 Code Coverage vs. Test suite Optimization Algorithms

While testing the modified software, the suitable methods to test case selection and prioritization techniques is shown as proposed test case selection and prioritization algorithm. To compare with proposed methods, the results of existing algorithms are referred from various research works. At the time of testing the modified software, it takes very less execution time and cost for the proposed test case selection algorithm. At the initial time of testing the software, medium code coverage is achieved.

5.CONCLUSION AND FUTURE ENHANCEMENT

5.1 CONCLUSION

In this research a proposed technique called mutual method is used to increase test code coverage for modified program at its initial stage of testing. The existing search based techniques select the test cases based on these parameters such as cost, code coverage and fault detection etc. However, these techniques are still lacks to arrive an optimal solution to a satisfactory level. Thus the proposed algorithm is designed to overcome these issues in test case selection and prioritization. The proposed test case selection algorithm uses the core functions of genetic algorithm. Whereas the test cases are selected based on mutation frequency of input parameters i.e. frequently utilized input parameter and number of program version. The result shows the optimal test suites selections that are selected by the proposed test case selection algorithm is very effective for all applications are tested. The test cases are efficiently ranked by the proposed test case prioritization method. Finally, the proposed algorithm reduces the test case execution time when compared with the previous techniques or the algorithms.

5.2 FUTURE ENHANCEMENT

The proposed algorithm does not consider about weight summation of termination event and unrestricted focus event. It also does not support restricted focus event. This has to be considered for further research whether the fault detection capacity is same as the latter. It still continues to analyze the relationship between the importance of events and tries to relax the assumptions and the quantification. Then the fault detection ability of an event interacts with others. There are a lot of interactions that are used to decide the weight value. Additionally, we still have to know about the weight values decided for each interaction.

REFERENCES

[1] Angelo susi et al. (2009), "Clustering test cases to Effective and scalable prioritization incorporating Expert knowledge", In proceedings of eighteenth symposium of software testing and analysis, ACM, pp.19-23.

[2] Bahsoon et al. (2001), "Methods and metrics for selective regression testing", In Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, pp.463-465.

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

[4] Colorni et al. (1991), "Distributed optimization by ant colonies", In Proceedings of the first European conference on artificial life, Vol.142, pp.134-142.

[5] Corey Sandler et al. (2004), "The Art of Software Testing", Second Edition Wiley.

[6] Dorigo and Marco (1992), "Optimization, learning and natural algorithms", Ph. D. Thesis, Politecnico di Milano, Italy.

[7] Feng Lin et al. (2006), "Applying safe regression test selection techniques to java web services", In Proceedings of the International Conference on Next Generation Web Services Practices, pp.133-142.

[8] Jeffrey and Gupta .R (2007), "Improving fault detection capability by selectively retaining test cases during test suite reduction", Software Engineering, IEEE Transactions, Vol.33, No.2, pp.108-123.

[9] Mansour et al. (1999), "Simulated annealing and genetic algorithms for optimal regression testing", Journal of Software Maintenance: Research and Practice, Vol.11, No.1, pp.19-34.

[10] Paul Jogersen .C (2008), "Software testing: A craftsman approach", 3rd edition, CRC press.

[11] Pearl and Judea (1988), "Probabilistic reasoning in intelligent systems: networks of plausible inference", Morgan Kaufmann.

[12] Radatz et al. (1990), "IEEE standard glossary of software engineering terminology", IEEE Std 610121990, pp.121-990.

[13] Rajappa et al. (2008), "Efficient software test case generation using genetic algorithm based graph theory", In First International Conference on Emerging Trends in Engineering and Technology, pp.298-303.

[14] Ribeiro et al. (2008), "A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software", In Proceedings of the 3rd international workshop on Automation of software test, pp.85-92.

[15] Roongruangsuwan et al. (2005), "Test Case prioritization techniques", Journal of Theoretical and Applied Informational Technology, Vol.2, No.4, pp.101-104.

[16] Rothermel et al. (2001), "Prioritizing test cases for regression testing", IEEE Transactions on Software Engineering, Vol.27, No.10, pp. 929-948.

[17] Rothermel et al. (2002), "The impact of test suite granularity on the cost-effectiveness of regression testing", In Proceedings of the 24th International Conference on Software Engineering, pp.130-140.

[18] Rummel et al. (2005), "Towards the prioritization of regression test suites with data flow information", In Proceedings of ACM Symposium on Applied Computing, pp.1499-1504.

[19] Ruth et al. (2007), "A safe regression test selection technique for web services", In Second International Conference on Internet and Web Applications and Services, pp.47-49.

[20] Shahid et al. (2011), "A study on test coverage in software testing", Advanced Informatics School (AIS), Universiti Teknologi Malaysia, International Campus, Jalan Semarak, Kuala Lumpur, Malaysia.

[21] Singh et al. (2006), "A New Technique for Version Specific Test Case Selection and Prioritization for Regression Testing", Journal of Computer Society of India, Vol.36, No.4, pp.23-32.

[22] Singh et al. (2010), "Test case prioritization using ant colony optimization", ACM SIGSOFT Software Engineering, Vol.35, No.4, pp.1-7.

[23] Smith et al. (2007), "Test suite reduction and prioritization with call trees", In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp.539-540.

[24] Srivastava et al. (2009), "Application of genetic algorithm in software testing", International Journal of software Engineering and its Applications, Vol.3, No.4, pp.87-96.

[25] Sthamer and Harmen-Hinrich (1995), "The automatic generation of software test data using genetic algorithms", Diss. University of Glamorgan.

[26] Suri et al. (2011), "Analyzing test case selection and prioritization using ACO", ACM SIGSOFT Software Engineering, Vol.36, No.6, pp.1-5.

[27] Suri et al. (2011), "Regression Test Suite Reduction using an Hybrid Technique Based on BCO and Genetic Algorithm", Special Issue of International Journal of Computer Science and Informatics, pp.2231-5292.

[28] Tai et al. (2002), "A test generation strategy for pairwise testing", Software Engineering, IEEE Transactions, Vol.28, No.1, pp.109-111.

[29] Tarhini et al. (2008), "Regression testing web applications", In International Conference on Advanced Computer Theory and Engineering, pp.902-906.

[30] Tsai .W.T., et al. (2005), "Rapid embedded system testing using verification patterns", Software, IEEE, Vol.22, No.4, pp.68-75.

[31] Walcott .K (2005), “Prioritizing regression test suites for time- constrained execution using a genetic algorithm”, Technical Report TR-CS05-11, Department of Computer Science, Allegheny College.

[32] Walcott et al. (2006), “Time aware test suite prioritization”, In Proceedings of the 2006 international symposium on Software Testing and Analysis, pp.1-12.

[33] Wappler et al. (2006), “Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm”, In Evolutionary Computation, pp.851-858.

[34] Wappler et al. (2005), “Using evolutionary algorithms for the unit testing of object-oriented software”, In Proceedings of conference on Genetic and evolutionary computation, pp.1053-1060.

[35] Xu et al. (2007), “Regression test selection for AspectJ software”, In 29th International Conference on Software Engineering, pp.65-74.

[36] Yoo .S and Harman .M (2012), “Regression testing minimization selection and prioritization: a survey”, Software Testing, Verification and Reliability, Vol.22, No.2, pp.67-120.

[37] Yoo Shin and Mark Harman (2007), “Pareto efficient multi- objective test case selection”, In Proceedings of International Symposium on Software Testing and Analysis, pp. 140-150.

Glossary

A

- Automated Testing – The use of scripts and tools to execute test cases without manual intervention.
- Acceptance Criteria – Conditions that a software product must meet to be accepted by stakeholders.
- Ad-hoc Testing – An informal and unstructured testing approach without predefined test cases.
- Assertion – A statement in a test case that verifies the expected outcome.

B

- Boundary Value Analysis (BVA) – A test design technique that focuses on values at the edges of input ranges.
- Bug Severity – The impact of a defect on the system, categorized as Critical, High, Medium, or Low.
- Bug Priority – The urgency of fixing a defect, usually determined based on business impact.

C

- Code Coverage – A metric that measures the percentage of code exercised by test cases.
- Critical Path Testing – Testing the most important user workflows to ensure core functionality is intact.
- Coverage Criteria – The rules that define how much of the application needs to be tested (e.g., statement, branch, or path coverage).

D

- Defect Density – The number of defects per unit of code, often used for test prioritization.
- Decision Table Testing – A technique used to test different combinations of inputs and their corresponding outputs.
- Dependency Analysis – Identifying dependencies between test cases to prioritize execution.

E

- Equivalence Partitioning – A test design technique that divides input data into groups where test cases cover one representative value per group.
- Exploratory Testing – A testing approach where test cases are not pre-written but dynamically created while testing.

F

- Failure Rate – The frequency at which a software component fails during testing.

- Fault-Based Testing – A testing strategy that selects test cases based on known defect patterns.
- Functional Testing – Testing that evaluates whether a system meets its functional requirements.

G

- Granularity of Tests – The level of detail in a test case, from high-level system tests to low-level unit tests.
- Gray-Box Testing – A testing approach that combines both black-box and white-box testing techniques.

H

- High-Risk Test Cases – Test cases that cover functionalities with a high impact on business or system stability.
- Historical Data Analysis – Using past test execution results to guide test case selection and prioritization.

I

- Impact Analysis – Assessing the effect of code changes on existing test cases.
- Integration Testing – Testing interactions between different modules or components of a system.
- Incremental Testing – Testing that gradually adds new functionality while ensuring previously tested features work.

J

- Just-in-Time Testing – Selecting and executing test cases dynamically based on real-time changes in software.

K

- Key Performance Indicators (KPIs) – Metrics used to evaluate the effectiveness of test case selection and prioritization.
- Kill Ratio – The percentage of defects detected by a specific set of test cases.

L

- Load Testing – Evaluating system performance under expected or peak load conditions.
- Low-Priority Test Cases – Test cases that have minimal impact on critical functionalities and can be deferred.

M

- Mutation Testing – A technique where small changes (mutations) are introduced to the code to check if test cases detect them.
- Model-Based Testing – Using abstract models of system behavior to generate and prioritize test cases.

- Manual Testing – The process of executing test cases manually without automation tools.

N

- Non-Functional Testing – Testing aspects like performance, security, and usability rather than functionality.
- Negative Testing – Testing how the system handles invalid or unexpected inputs.

O

- Orthogonal Array Testing – A systematic approach to selecting a minimal set of test cases while covering all variable combinations.
- Optimization-Based Prioritization – Using algorithms to prioritize test cases for maximum effectiveness.

P

- Path Coverage – A metric that evaluates how many execution paths in the code have been tested.
- Prioritization Criteria – Factors used to rank test cases, such as business impact, failure rate, or execution time.
- Performance Testing – Evaluating system responsiveness and stability under different conditions.

Q

- Quality Risk Assessment – Analyzing risks associated with different software components to guide test selection.
- Quarantine Tests – Temporarily isolating test cases that fail intermittently until the issue is resolved.

R

- Regression Testing – Retesting modified software to ensure that new changes do not break existing functionality.
- Risk-Based Testing – A prioritization approach that focuses on testing high-risk areas of the application.
- Requirement Coverage – The extent to which test cases cover documented software requirements.

S

- Smoke Testing – A subset of test cases that check whether the basic functionality of an application works.
- Sanity Testing – A quick, focused test to verify that recent code changes did not break critical functionality.
- Static Analysis – Reviewing code for errors without executing it, often used for early defect detection.

- System Testing – Testing the entire system as a whole to validate its compliance with requirements.

T

- Test Case Prioritization – The process of ranking test cases to maximize defect detection and efficiency.
- Test Selection Criteria – Rules that determine which test cases should be executed based on changes, risk, and coverage.
- Test Suite – A collection of related test cases that verify specific aspects of a software system.
- Traceability Matrix – A document mapping test cases to requirements, ensuring complete coverage.

U

- Usability Testing – Evaluating how easy and user-friendly a software product is.
- Unit Testing – Testing individual components or functions of a software application in isolation.
- Update-Based Selection – Choosing test cases based on recent code changes.

V

- Validation Testing – Ensuring that the software meets user requirements.
- Verification Testing – Checking whether the software meets its specified requirements before deployment.
- Version Control Testing – Testing software versions to ensure changes do not introduce new defects.

W

- White-Box Testing – A testing approach that examines internal code structures and logic.
- Weighted Prioritization – Assigning weights to test cases based on factors like business impact and failure history.

X

- XP (Extreme Programming) Testing – A testing practice that emphasizes automated unit testing in agile development.

Y

- Yield Measurement – Evaluating the effectiveness of test case selection by measuring defect detection rates.

Z

- Zero-Day Testing – Testing performed immediately after a feature or update is introduced to identify critical defects.

Index