Generating effective test suite by minimization and optimization of test cases in regression testing

Md. Nurul Amin
Department of Computer Science
University of North Dakota,
Grand Forks, USA
m.amin@und.edu

Chinedu Nwachukwu
Department of Computer Science
University of North Dakota,
Grand Forks, USA
chinedu.nwachukwu@und.edu

Wei Chen
Department of Computer Science
University of North Dakota,
Grand Forks, USA
wei.chen.2@und.edu

***Abstract*—**

Regression testing works in finding defects when modification occurs in a software. To improve the regression testing technique we need to find out the most effective set of test cases that can figure out the faults faster, efficiently and intensively. If the execution of a huge test suite has limited time, we need to prioritize the test cases to keep the most important test cases on top to be executed early. Now, a minimal test suite with proper organization of execution level in test cases can give better results in finding faults. In this paper, we combined a minimization algorithm with a prioritization approach to build the most effective test suite that is very efficient and quick in detecting faults in testing of software systems.

Keywords— Regression Testing, Test case prioritazation, Test case minimization, Fault Detection.

I. INTRODUCTION

Generating effective test cases is an essential part of software testing. Every software made must be tested at some point in time, either during the development stage or at the improvement stage. All software's must be passed through certain constraints to ascertain its durability, strength its ability to meet the customers' requirements ad specifications. These processes can be achieved via various stages in the software development cycle and the results of each test is different at all levels. Because of this, we put our focus on generating effective test suite to be able to optimally run the test suite and get the best possible results from the test execution. Going through various techniques in research papers, journals, and topics of discussion, we could discover that adequate algorithms and methodologies have been proffered for the generation of test cases in new software development stages. Despite this, we further researched into vital & specific areas which uncovered the interesting part and essence of regression testing. Regression's testing objective is to test the changed system per its specification and requirement. Regression testing occurs after some actions and activities have been performed in the system. Regression testing can be said to mean "return of a bug" in simple terms. Regression testing can also be defined as testing of software after its release or an upgrade. Software's always develop bugs when it is being modified over time. Software modifications could be due to adding of new functionality, improving performance, bug fixing, etc. The modified software may break the functionality of the system that previously worked. So, regression testing concentrates on finding the issues after a major code change has been done [3][4].

There are several types of regression testing and different focus areas, likewise several constraints related to regression testing that make it difficult a task. Some of these constraints include but not limited to time, cost constraints, developers target meeting, software deadline, managers trying to keep project cost under control, etc. All or almost all regression tests are based on the functionality (Black box testing) and architecture (Grey box testing) [3]. Our focus in regression testing is based on the software system functionality (Black box test). This enables us to focus on the effectiveness of the software system testing suite which would directly be related to the system requirements or specifications, for the success of the software system design and development process.

A simple way of regression testing is the re-execution of all test case suites for the entire software system which could be extremely expensive. Software project deadlines, restrict the performance of exhaustive testing, to successfully test for all possible outcomes [3][4]. With this in mind, we were able to look in-depth into the importance of regression testing and the importance of eliminating the irrelevant test cases (Un-related sections). We will need to minimize the test case number to focus on the major areas affected by the modification effect, also we need to further improve the strength of the test cases that will be executed. A smaller subset of effective regression tests would effectively aid in faster fault detection and the minimal regression test sets act as a precursor to further testing [3]. Once the minimization of the test cases has been achieved, we then further prioritize the test cases as this would improve the weight and strength of these select test cases. Prioritization organizes the level of execution for the test cases and it gives an improved rate of fault identification, when the test suite cannot be completed [4]. Once the prioritization successfully rearranges the test cases, maximum faults are bound to be detected in the shortest possible time [4]. Our goal is to improve regression testing by finding out the most effective set of test cases that will be used to find

system faults faster and effectively. Our proposed technique works with minimization, that selects the best parts that are directly or closely related to the affected or modified area in the software system and prioritization, by setting high fault value in test cases. The combination of both minimization and prioritization builds a test suite that is most effective, efficient, and quick in testing of software systems. The test suite would be able to detect major faults even if execution of all prioritized test cases cannot be completed. The combination further concentrates on the effectiveness of the test cases and brings out the best from its execution.

II. Related Works

Panigrahi and Mall (2012) [30] developed a Regression Test Selection technique for object oriented programs and the UML state machines for the affected classes. The control and the data dependency also captured. Leung and White emphasized a firewall for regression testing of system integration [17]. Laski and Szemer provided an approach for test case selection which is on the basis of cluster identification technique [18]. Dr. ArvinderKaur and ShubhraGoyal [11] conceived a novel genetic algorithm by implementing the prioritization on regressing testing on time-limited surroundings based on entire fault coverage. Average Percentage of Faults Detected (APFD) is going to assist this algorithm to automate the process and analyze the experiment outcomes.

D. Jefferey et al. [25] Proposed a novel technique serves for test suite minimization in attempt to utilize extra coverage information on test, where a couple of exclusive test cases in the minimized suites that are redundant regarding the testing criteria are exploited for test case minimization. James A. Jones et al. [26] exhibited some original algorithms for test case minimization and prioritization that can be adapted effectively with modified condition/decision coverage, MC/DC. Ahmed and Hermadi (2008) [31] proposed hybrid techniques possesses minimization, combining modification and prioritization based selection to spot a delegate division of all test cases which is going to trigger different output performance on the new software version.

According to, [32] test case reduction and selection optimization in testing web services environment, test execution is expensive as in most cases. The service revenue models are based on the number of innovations or test executions. In principle, no problem or limit to the number of test cases that are generated. Most coverage calculations depend on calculating the number of faults found by the test cases and not the number of test cases. The number of faults or any related attribute are in the numerator, while the number of test cases are in the denominator. Putting all this into consideration, to optimize the coverage, we have to increase the amount of faults or possible faults found while decreasing the number of executed or generated test cases. The focus can be broken down into two categories: - Develop a pre-test execution component that can evaluate generated test cases and optimize the selection from those generated test cases for execution. The second is the utilization of historical usage sessions that can be provided by clients or service provider. Such usage sessions can direct and optimize the process of test case generation and execution. This methodology increases the coverage and reduces the execution cycles in creating a pre-execution component on the client side to perform initial validations on the generated test cases before possible validation for execution.

In, [33] faulty and irrelevant test cases can be selected. Much more factors need to be placed into consideration during the selection process. Test cases can be generated from specification represented using the Unified Modeling Language (UML) [3]. Steiner tree algorithm inputs are given as a set of terminals with directed graph G. The changed nodes are included in the test path by defining them as terminal. Regression test selection techniques have been used in [35] [36] [37] to select a subset of test cases. The objective is to check if the modified program has the same behavior as the previous acceptable version of the program running on T, a set of test cases [34]. UML activity diagrams have been used for specification. A minimal set of test cases are generated to go through the modification path in order to test the system effectively.

Badhera et al. [9] showcased a technique to run the changed code line sections with small scale number test cases. Prioritization tries to select the test cases from the suite by executing fewer code lines. Hence, obtaining faster code coverage should be reached for the sake of detection of faults. Bixin Li et al. (2012) [30] put forward a method of selecting test cases in terms of regression testing of composite service, based on extensible BPEL flow graph. B. Jiang et al. [10] Proposed a method called ART-based prioritization by accepting test suite as input, generates the output in descending order based on one algorithm. Fundamentally, selecting one test case from the candidate set generated first until all test cases have been covered. Goal functions are created for counting the distance between two test cases and how to choose a test case from the candidate set. Code coverage data eventually decide the distance counting of two test cases. After that, a candidate test case that is pertinent to distance test cases which has been prioritized beforehand.

H. Do et al. [12] illustrated the significance of test case prioritization using time constraints operator and unearthed the constraints which modifies the technique performance. What's more, carried out three groups of experiments to disclose the time constraints. The experiment results showcase that the cost effectiveness and cost benefit trade-offs significantly depend on the time constraint factor by using this technique. Another experiment replicates the first experiment, counting a couple of threats to verify the amount of fault currently. Third experiment operates the amount of program faults to inspect the effect of imprecision on prioritization and showcases the pertinent cost-effectiveness of prioritization techniques.

Park et al. [13] created a cost awareness model serves to test case prioritization and fault severities which disclosed in the former test execution. Simultaneously, it doesn't dynamically modify one result to another. Mohamed A Shameem et al. (2013) Proposed a standard for evaluating the proportion of fault detection. This algorithm screens the fault in advance and validation of prioritized test cases are compared to the non-prioritized cases by Average Percentage of Fault Detection (APFD).

M. Yoon et al. [14] put forward an approach to prioritize original test cases by gauging the requirements of risk exposure value and measuring risk objects. Further it deciding the test case priority via evaluated values after counting the pertinent test cases. Furthermore, we prove our approach is effective by empirical studies in terms of Average Percentage of Fault Detected (APFD) and fault severity.

R. Krishnamoorthi and S. A. Mary [15] exhibited a prototype to prioritizes system test cases on the basis of six factors: customer priority, changes in requirement, implementation complexity, usability, application flow and fault impact. This technique is scrutinized in three periods with student and industrial projects.

S. Raju and G.V. Uma [16] inaugurated a cluster-based test case prioritization technique. Test cases are collected on the basis of their dynamic runtime behavior. Researchers subtracted the necessary number of pair-wise comparisons. Simultaneously, a value-driven approach to system-level test case prioritization was proposed by researches, which process in prioritizing test requirements. In this cases, test cases prioritization is based on four elements: rate of fault detection, requirements volatility, fault impact and implementation complexity.

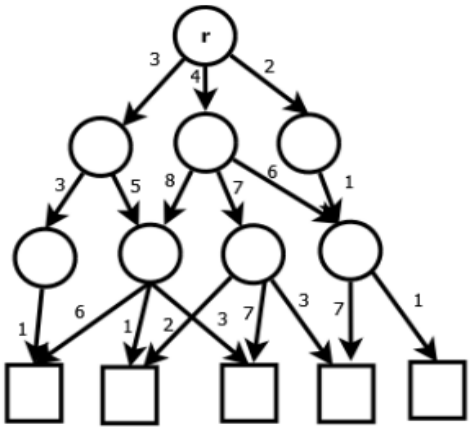
In [17], [20], Rothermel et al. were the leader to concentrate on test case prioritization predicaments which paved the way for them to showcase six varying tactics on the basis of the coverage of statement or branches. In [21], Li et al. offers experiential study results of metaheuristic searching techniques and greedy searching techniques applied to programs for regression test case prioritization. In [24], Praveen et al. commenced an original test case prioritization algorithm that count average faults detected per minute. A Regression Testing Technique for Test Case Prioritization based on Code Coverage criteria is advocated by K.K. Aggarwal in [23]. Devised and execute an experiment under control, scrutinizing If test case prioritization could be validated on Java programs using Junit and assessed that test case prioritization is able to dramatically enhance the rate of fault detection of JUnit test suites. S. Elbaum et al. [27] showcased that all prioritization techniques considered can improve the rate of fault detection of test suites. Huang (2010) [28] has proposed a cost cognizant test case prioritization technique on the basis of the usage of historical records and genetic algorithm. They execute an experiment under control to appraise the proposed technique's effectiveness. This technique however does not care about the test cases similarity. Sabharwal (2011) [29] has put forward an approach for prioritization test case scenarios obtained from activity diagram using the notion of elementary information flow metric and genetic algorithm. Sabharwal (2011) [29] has generated prioritized test case in static testing utilizing genetic algorithm. They have employed a similar approach as to prioritize test case scenarios obtained from source code in static testing.

III. Problem Definition

Test case reduction from a large test suite is a big challenge as regression testing requires very little time to run if efficiency is key. When the need to remove test cases arise, we then should consider focusing on related test cases over affected areas which should be tested due to modification. A graph based test case selection approach can help us in finding out the major areas that are mandatory for testing. After selecting important test cases, we then prioritize them, so that an effective test suite can be generated. This test suite can then figure out maximum defects, in a software system, in the shortest time possible.

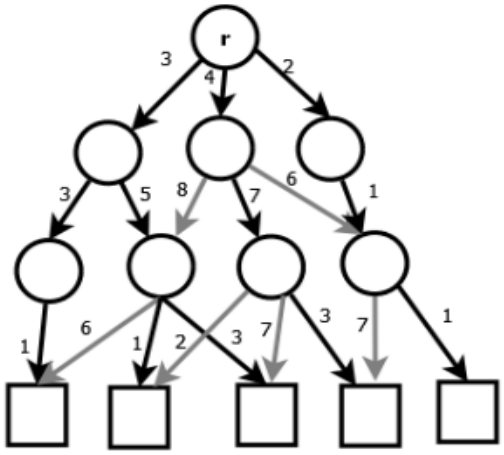
IV. Steiner Tree Algorithm

Steiner tree algorithm is a combinatorial, optimization, problem solving graph based approach that gets the optimal paths. The basic difference with Steiner tree and spanning tree is that unlike spanning tree, Steiner tree does not span all vertices in the graph [3], it only spans a subset of the path from root node to leaf nodes in the terminology. This process brings out the most cost effective set of test cases. In Steiner tree algorithm nodes are divided into two categories, terminals, and non-terminals. Terminals are vertices that must be included in the solution. Non-terminals may be included when necessary to connect with terminals. Edge weight are used to define cost in the path of Steiner tree. In-order to reduce cost, non-terminal vertices with lower edge weights may be included in certain areas.



a. Graph G, with root ,r, terminal nodes and edge weights

FIG a: A Simple Graph with terminal nodes



b. Steiner Tree of graph G, showing in black nodes and edges

FIG b: Steiner tree generation of the Simple graph.

The above figures show an example of a simple graph and its Steiner tree.
In the given directed graph, G(V, E, w) the components are
V, set of vertices
E, set of edges
r, root node.
w, is the weight of edges (> 0).

Our goal is to find out a minimal cost tree path in this directed graph that connects all terminals to root node r. In [1][2] the Steiner algorithm that has been discussed earlier, we adopted the methodology as it can be used in reaching our desired goal. The goal is to find out a subset of edges that comes down from root to terminal node with minimal weight.

Algorithm 1: Steiner Algorithm

Algorithm: MST-Steiner

Input: A graph G = (V, E, w) and a terminal set L $\subseteq$ V
Output: A Steiner tree

1. Construct the metric closure GL on the terminal set L
2. Find an MST TL on GL
3. T <- Φ
4. For each edge e = (u, v) $\in$ E(TL) in a depth-first-search order of TL do
4.1 Find a shortest path P from u to v on G
4.2 If P contains less than two vertices in T then
Add P to T
Else
Let p_i and p_j be the first and the last vertices already in T Add sub paths from u to p_i and from p_j to v to T
5. Output T

For example, consider the **FIG: a –** Simple graph with terminal nodes, graph with 'r' as the root.
The square nodes in the bottom indicate terminals and weights are given beside the edges. **FIG: b** – Steiner tree generation of the simple graph, shows the Steiner tree with minimized nodes and edges.

V. Test Case in Graph

A Finite State Machine (FSM) diagram consists of a lot of execution paths, from the start state to the final state consisting of transactions and activities.
A Test Case (TC) can be defined as a full path in finite state machine (FSM) diagram.
tc $\in$ TC, tc = $a_0 \rightarrow t_0 \rightarrow a_1 \rightarrow t_1 \rightarrow \ldots\ldots \rightarrow t_n \rightarrow a_m$
where $a_i \in A$, $t_i \in T$,
a_0 is the initial state,
a_m is the final state.
TC is the set of test cases

VI. Detecting minimized Test suite in Regression Testing

Test Suite minimization is a NP-hard real world problem [8]. It consists of selecting a minimal set of test cases that covers a given set of requirements and minimizes the amount of resources required for its execution. After modifying the new software version, we need to find out the affected areas first. Then we figure out the relevant test cases from root node to terminal node. Our work is to find out the minimized number of test cases that will detect faults in the modification done on the software system. As it is a graph based optimization approach, we would find out the relevant path from root to terminal nodes for test cases.
At first, we need to draw the high-level architecture diagram to find out the interaction with different modules of the software. Secondly, we draw the low-level architecture diagram, from which we figure out the data flow among different states which will help to design the Control Flow Graph (CFG) diagram. Finite State Machine (FSM) diagram is a pre-requisite to draw the Control Flow Graph (CFG) of the software system. From the control flow graph, we can find out the number of nodes connected to our newly modified node. So, we converted our finite state machine (FSM) diagram to Control Flow Graph(CFG). Each state becomes a node in the CFG and control flow lines become the edges. All weights in this case, were measured based on interaction with other nodes which will introduce the coverage of nodes in the program execution. From our references for this work, each weight calculation is based on the number of incoming and outgoing edges from a node.

Weight(e) = $(n_i)_{in}$ X (n_j)out

where $(n_i)_{in}$ is the number of incoming lines in node n_i and (n_j)out is the number of outgoing lines in node n_j and e is the edge connecting n_i and n_j.

VI.I . Algorithm

From our control flow graph, we need to find out the terminal nodes, then we can find the path from root to terminal nodes where the program would stop after some iteration. The process of selecting terminal node is given below:

a. The root node should be a terminal node in CFG.
b. Consider the stopping nodes, where the user will get some value as a terminal node.
c. Make the modification node a terminal node.

Although our testing method is black box based, we need to get the location of the modified node in the program control flow graph by going through the program. This is essential because the modification node will come in the test path as a part of regression testing.

Algorithm 2: Test Case Generating Algorithm

Input: A directed graph G = (V, E, w) with a terminal set L $\subset$ V

Output: TC, a set of Test Cases, that should be used to test the system.

1. For each node in the finite state machine diagram A, do

1.1. If node is a Start State, Stop State, Transition State, Multi State, convert into node in CFG.

2. For an edge in the finite state machine diagram, A do

2.1 If an edge has a cycle(loop), unfold the loop up-to 2 iterations. Then add edges and nodes to CFG.

2.2 Else, add the edge in CFG directly.

3. Calculate the edge values using number of in and outgoing vertices.
4. Define all terminal edges in the graph using square notification.
5.Minimize the graph using Steiner tree algorithm.
6. Generate TCs, from root node to terminal nodes including modification node in execution paths.

VII. TEST CASE PRIORITIZATION

The main objective of this technique is to meet the specific goals, within the stipulated time and cost, faster than it would be if they were not prioritized [7]. From Steiner tree algorithm, we generated a minimal number of test cases that is adequate to test the system after program modification. Sometimes the system could be so complex that Steiner tree algorithm would bring out a large number of test cases. In these circumstances, prioritization can help to sort the test cases in proper level of major faults coverage. In our proposed methodology, we considered the code coverage and fault values of test cases in our prioritization. We had an execution history when our test cases found faults before fixing them or adding new features. We then use the previous results of execution to get the faults values of test cases. The test cases that have a higher fault value with most coverage nodes should be prioritized first. We are now able to get an effective set of test cases with proper level of execution in finding defects.

Prioritization weight was assigned in every test case before sorting them accordingly.

$$\text{Prioritization Weight of a TC} = \frac{\text{Fault value}}{\text{Code coverage nodes}}$$

VIII. METHODOLOGY AND IMPLEMENTATION

For implementation purpose, we developed a C program with basic functions, loops and constructs. This software program was used during the implementation of our approach and this would be explained extensively with the use of Finite State Machine diagrams.

The C program – STUDENT GRADE DATABASE (SGD) is a program that utilizes functions of C programing construct, it has a MENU for selecting the options for execution. This MENU would act as the program start page and direct you to the selected module program, that you wish to execute. Once execution is complete, you will be directed back to the menu for onward action, which includes the following tasks "New Student Entry, Save to Database, Load Database, Search for Student Information, Enter Student Grade, Delete Student from Database and Exit".

FIG: 1 – (HIGH LEVEL ARCHITECTURE) Shows the High Level Finite state machine diagram for the software. This diagram shows the construct and connection of the modules of the program. With this diagram, you will be able to understand the aim and objective of the program.

Looking at the diagram, we are unable to interpret the stages of the program flow as it isn't detailed enough to display all the required nodes.

The program SGD was constructed with the aim of using it as an object of implementation. In accomplishing this, we will need to clearly define the steps and existing paths in the program from which we would be able to derive the initial test cases that will be used for implementation. The clarification and path definition was done by **FIG: 2** – (LOW LEVEL) This shows the Low level finite state diagram for Student Grade Database. It displays various levels and gives you a better understanding of the process flow of the software. Based on the diagram, you will be able to execute the software program step by step just as it would be executed. Because of this diagram, we were able to generate test cases (manually) that will be used in testing the software for errors and defects.

Test cases generated have different execution paths and execution functions.

The execution path of the test cases was clearly defined in **TABLE 1**: - Showing 81 generated test cases. This table shows the code coverage based on the levels and program flow paths during its execution. Based on these paths we were able to detect faults in the software system program and all were indicated in the table. These 81 detected test cases are to be used to test the full performance and functional stability of the software system.

Remembering our goal is not the testing of the whole software system, we then selected one of the major faults detected, modified the program and fixed the error that was detected. Once the selected issue was fixed, we then arrived at our desired destination. This is the stage where regression testing can be implemented.

Recalling our goal – generation of effective test suite for regression testing. Removing the un-related test suite, then considering the affected area that should be tested due to modification. A graph based test case selection approach can help us find the major areas related to the modified area for testing. This graph based approach can only be achieved with the use of a control flow graph for the SGD software program. **FIG: 3** – (CONTROL FLOW GRAPH) Shows the Control Flow Graph of Student Grade Database. This uses a numbering approach of the program flow and execution paths. The Control Flow Graph shows the weights assigned for the edges. With this graph, we are now able to apply Steiner Tree Algorithm (STA) which uses the graph based approach in test case minimization [3]. This algorithm has been selected due to its ability to select efficiently, the best test case paths for execution of the modified area. Using the weight assigned for the edges, we are able to define the root, terminal and non-terminal nodes [3]. Steiner Tree Algorithm (STA) finds the shortest part through the modified area, from root node to terminal node, finding all possible test cases for the modified area [3].

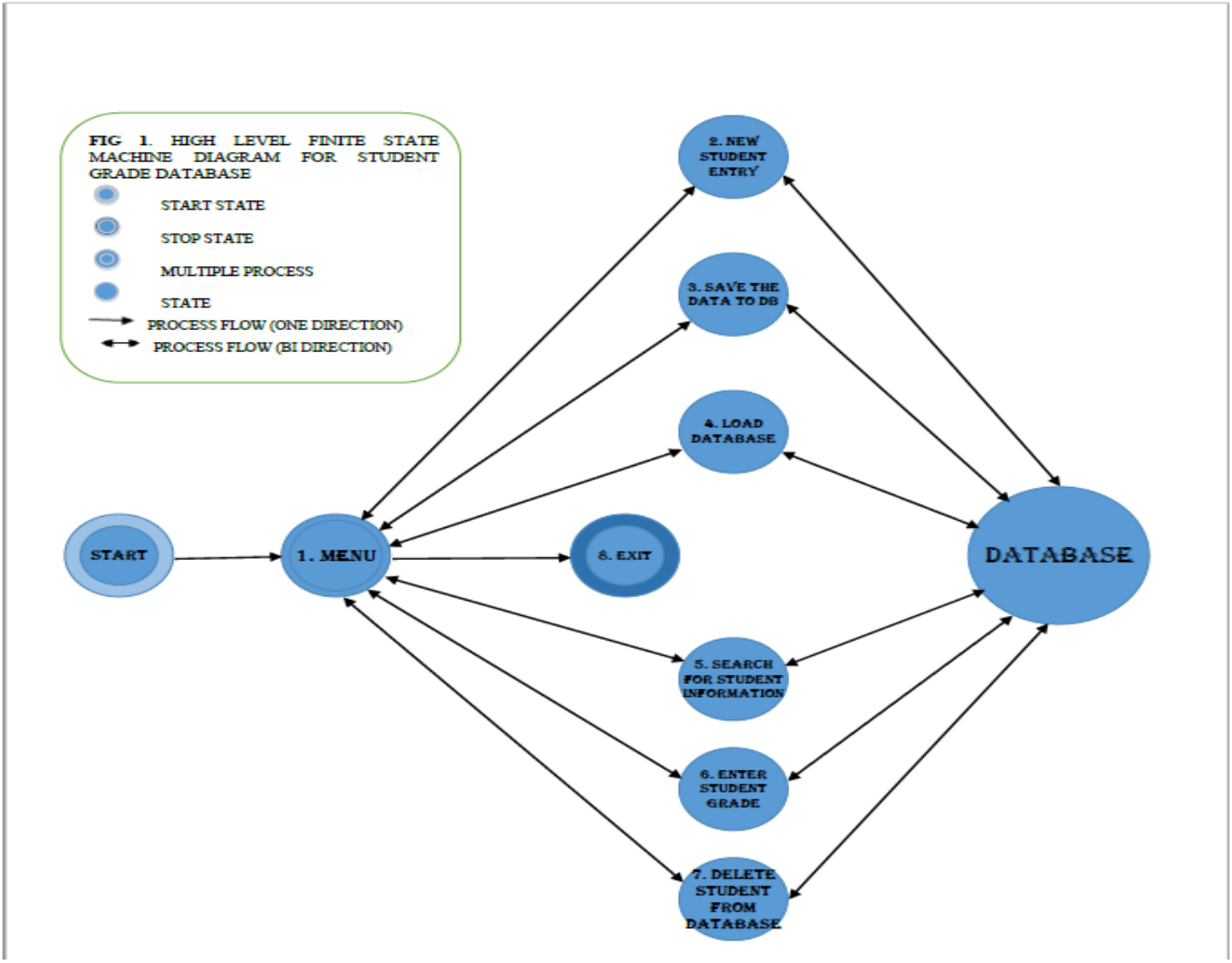


FIG 1: HIGH LEVEL ARCHITECTURE DIAGRAM

FIG: 4 – Shows the Control Flow Graph for SGD with the Root, Terminal and non-terminal nodes defined. Node 1 – Root node – is the program start node in which the execution commences, Node 1 – Terminal node – is the program output node or continuity node for some program iteration and Non-Terminal Nodes – these are the connecting nodes through the execution paths [3]. The Steiner Tree Algorithm (STA) generates a graph used in selecting the minimized test case paths from the graph which is shown below: -

Minimized selected test paths

1st. 1→21→8→9→10→13→1
2nd. 1→22→23→24→27→9→10→13→1
3rd. 1→30→23→24→27→9→10→13→1
4th. 1→31→23→24→27→9→10→13→1
5th. 1→22→23→25→28→9→10→13→1
6th. 1→30→23→25→28→9→10→13→1
7th. 1→31→23→25→28→9→10→13→1
8th. 1→22→23→26→29→9→10→13→1
9th. 1→30→23→26→29→9→10→13→1
10th. 1→31→23→26→29→9→10→13→1

FIG: 5 – Showing the Control Flow Graph for SGD with the selected nodes for the above paths. This paths are the minimized test cases from the initial 81 derived test cases.

TABLE: 2 – Listing out the test cases derived from the paths above, showing the No of Faults that can be detected using this test suite. Also, the Total Coverage node number is shown in the table, showing the paths that it would reach during execution.

These test cases are adequate enough to sufficiently test the software system's modified area but our goal is not to find adequate test cases. Our goal is to generate the most efficient and effective test suite for regression testing.

To achieve this, we need to prioritize this test suite according to some criteria. Test case prioritization helps in sorting the test cases in proper level of covering major faults [6]. In prioritization, we make use of the test case code coverage and fault values of the minimized test cases. A value for weighing the strength of each test case has been defined and it is called "Priority Weight (PW)". The Priority weight is gotten by

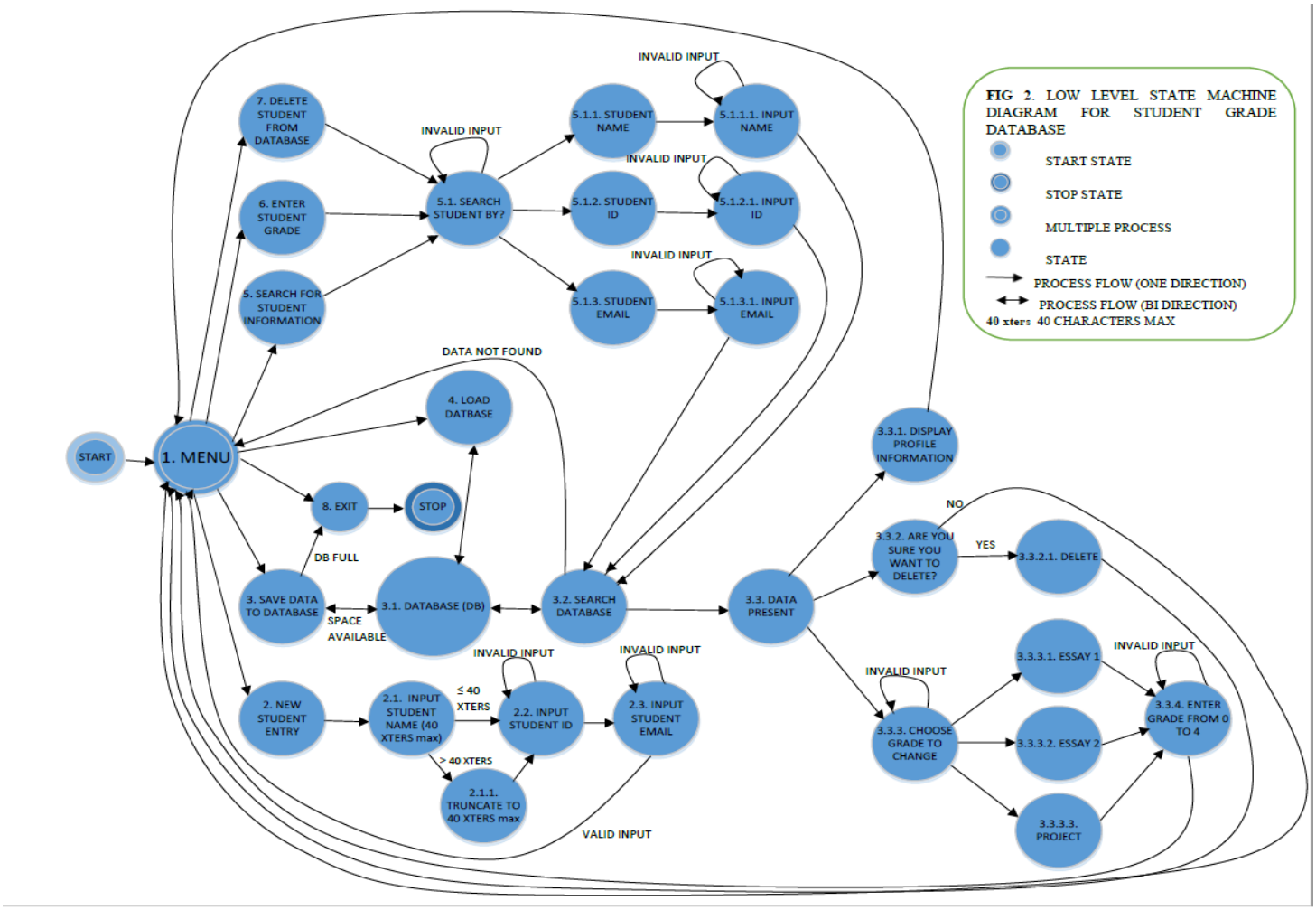


FIG 2: LOW LEVEL ARCHITECTURAL DIAGRAM

TABLE 1: INITIAL TEST CASES GENERATED FOR THE SGD

INPUT VALUES	1	2	2.1(4)	2.1.1	2.2(3)	2.3(2)	3(3)	3.1	3.2(1)	3.3	3.3.1	3.3.2(1)	3.3.3(1)	3.3.2.1(1)	3.3.3.1	3.3.3.2	3.3.3.3	3.3.4	3.4	4	5	5.1(1)	5.1.1	5.1.2	5.1.3	5.1.1.1(4)	5.1.2.1(3)	5.1.3.1(2)	6	7(1)	8	Number of faults
TC 1(1-2)	x	x	x	x	x	x														x												9
TC 2	x	x	x		x	x																										9
TC3 (1-3)	x						x												x													3
TC4	x						x	x	x																							4
TC5	x						x	x	x	x	x																					4
TC6	x						x	x	x	x		x																				5
TC7	x						x	x	x	x		x		x																		6
TC8	x						x	x	x	x			x		x			x														5
TC9	x						x	x	x	x			x			x		x														5
TC10	x						x	x	x	x			x				x	x														5
TC11(1-4)	x							x	x											x												1
TC12	x							x	x	x	x									x												1
TC13	x							x	x	x		x								x												2
TC14	x							x	x	x		x		x						x												3
TC15	x							x	x	x			x		x			x		x												2
TC16	x							x	x	x			x			x		x		x												2
TC17	x							x	x	x			x				x	x		x												2
TC18 (1-5)	x								x												x	x	x			x						6
TC19	x								x	x	x										x	x	x			x						6
TC20	x								x	x		x									x	x	x			x						7
TC21	x								x	x		x		x							x	x	x			x						8
TC22	x								x	x			x		x			x			x	x	x			x						7
TC23	x								x	x			x			x		x			x	x	x			x						7
TC24	x								x	x			x				x	x			x	x	x			x						7
TC25	x								x												x	x		x			x					5
TC26	x								x	x	x										x	x		x			x					5
TC27	x								x	x		x									x	x		x			x					6
TC28	x								x	x		x		x							x	x		x			x					7
TC29	x								x	x			x		x			x			x	x		x			x					6
TC30	x								x	x			x			x		x			x	x		x			x					6
TC31	x								x	x			x				x	x			x	x		x			x					6
TC32	x								x												x	x			x			x				4
TC33	x								x	x	x										x	x			x			x				4
TC34	x								x	x		x									x	x			x			x				5
TC35	x								x	x		x		x							x	x			x			x				6
TC36	x								x	x			x		x			x			x	x			x			x				5
TC37	x								x	x			x			x		x			x	x			x			x				5
TC38	x								x	x			x				x	x			x	x			x			x				5
TC39 (1-6)	x								x													x	x			x			x			6
TC40	x								x	x	x											x	x			x			x			6

TC41	x							x	x		x											x	x			x			x			7
TC42	x							x	x		x		x									x	x			x			x			8
TC43	x							x	x			x		x			x					x	x			x			x			7
TC44	x							x	x			x			x		x					x	x			x			x			7
TC45	x							x	x			x				x	x					x	x			x			x			7
TC46	x							x														x		x			x		x			5
TC47	x							x	x	x												x		x			x		x			5
TC48	x							x	x		x											x		x			x		x			6
TC49	x							x	x		x		x									x		x			x		x			7
TC50	x							x	x			x		x			x					x		x			x		x			6
TC51	x							x	x			x			x		x					x		x			x		x			6
TC52	x							x	x			x				x	x					x		x			x		x			6
TC53	x							x														x			x			x		x			4
TC54	x							x	x	x												x			x			x		x			4
TC55	x							x	x		x											x			x			x		x			5
TC56	x							x	x		x		x									x			x			x		x			6
TC57	x							x	x			x					x					x			x			x		x			5
TC58	x							x	x			x			x		x					x			x			x		x			5
TC59	x							x	x			x				x	x					x			x			x		x			5
TC60 (test 1-7)	x							x														x	x			x				x			7
TC61	x							x	x	x												x	x			x				x			7
TC62	x							x	x		x											x	x			x				x			8
TC63	x							x	x		x		x									x	x			x				x			9
TC64	x							x	x			x		x			x					x	x			x				x			8
TC65	x							x	x			x			x		x					x	x			x				x			8
TC66	x							x	x			x				x	x					x	x			x				x			8
TC67	x							x														x		x			x			x			6
TC68	x							x	x	x												x		x			x			x			6
TC69	x							x	x		x											x		x			x			x			7
TC70	x							x	x		x		x									x		x			x			x			8
TC71	x							x	x			x		x			x					x		x			x			x			7
TC72	x							x	x			x			x		x					x		x			x			x			7
TC73	x							x	x			x				x	x					x		x			x			x			7
TC74	x							x														x			x			x		x			5
TC75	x							x	x	x												x			x			x		x			5
TC76	x							x	x		x											x			x			x		x			6
TC77	x							x	x		x		x									x			x			x		x			7
TC78	x							x	x			x		x			x					x			x			x		x			6
TC79	x							x	x			x			x		x					x			x			x		x			6
TC80	x							x	x			x				x	x					x			x			x		x			6
TC81	x																														x		0

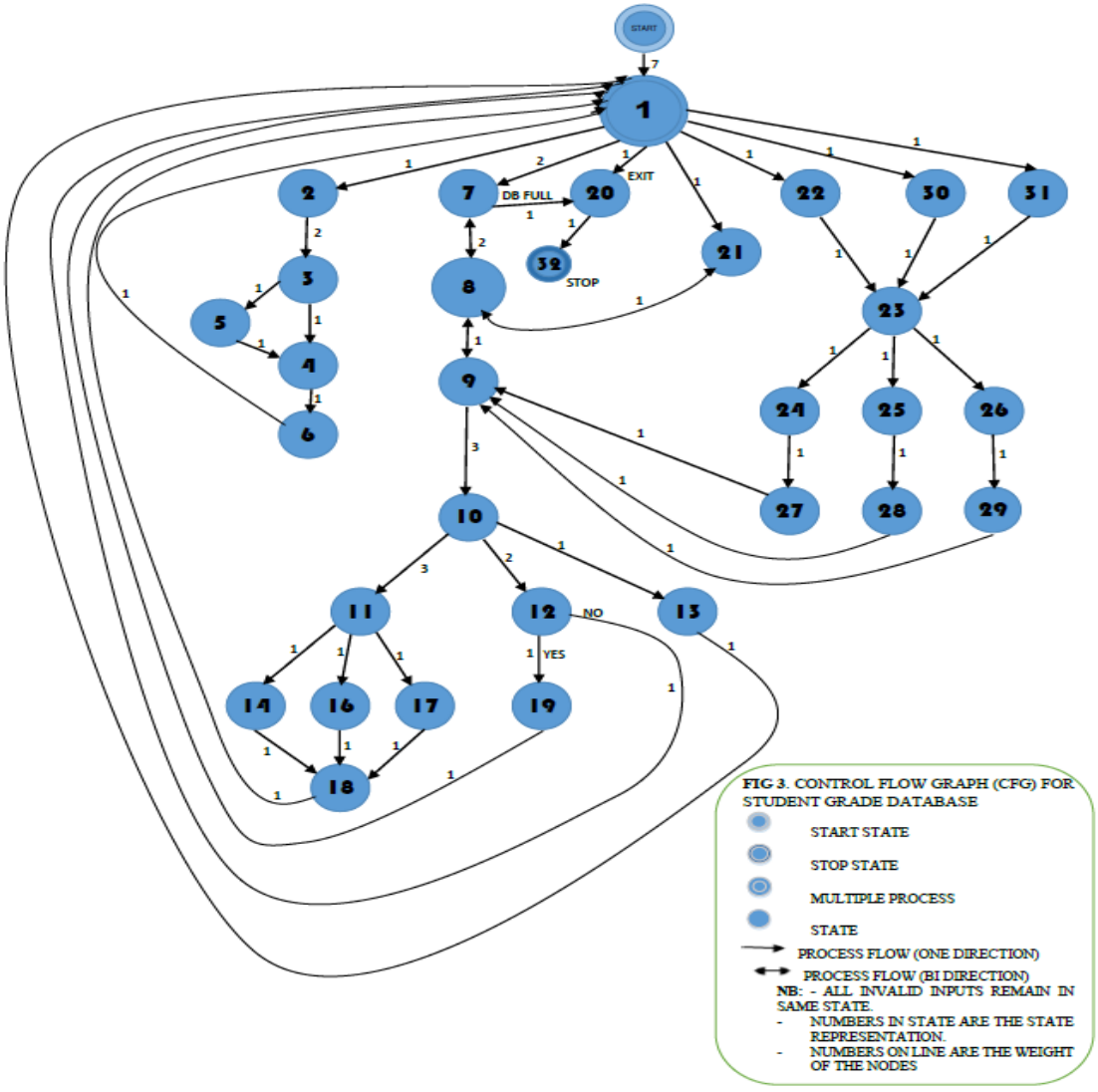


FIG 3: CONTROL FLOW GRAPH FOR SGD

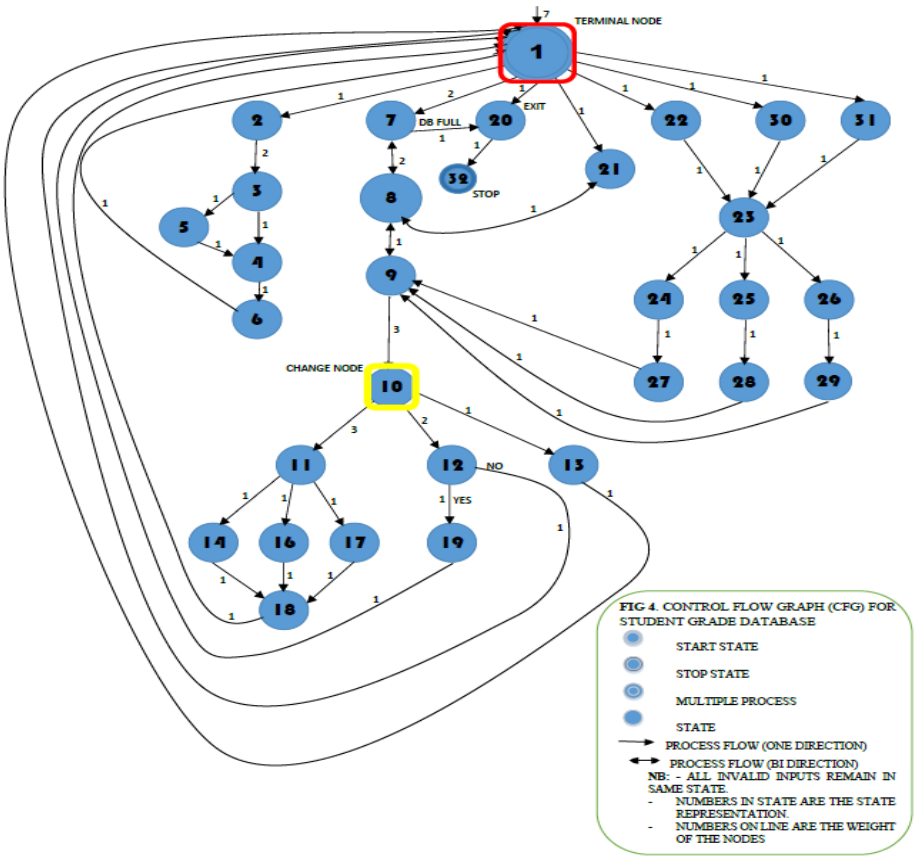


FIG 4: CFG SHOWING ROOT, TERMINAL AND CHANGE NODE

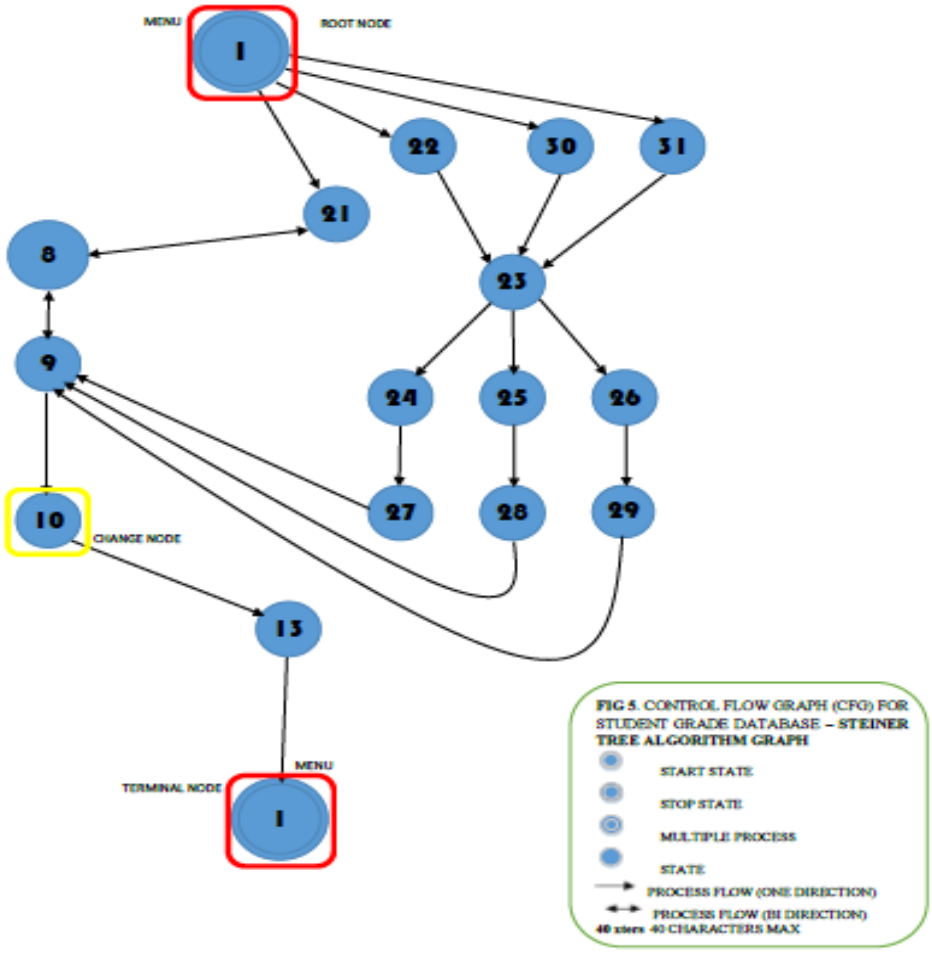


FIG 5: CFG WITH SELECTED FLOW PATHS USING STEINER ALGORITHM

TABLE 2: MINIMIZED TEST CASES

Minimized TCs (TC ID)	TC Index	1	3(3)	3.1	3.2(1)	3.3	3.3.1	5	5.1(1)	5.1.1	5.1.2	5.1.3	5.1.1.1(4)	5.1.2.1(3)	5.1.3.1(2)	6	7(1)	Number of faults	Coverage Node	Priority Weight
TC5	tc 1	x	x	x	x	x	x											4	6	0.666
TC19	tc 2	x			x	x	x	x	x	x			x					6	9	0.666
TC26	tc 3	x			x	x	x	x	x		x			x				5	9	0.555
TC33	tc 4	x			x	x	x	x	x			x			x			4	9	0.444
TC40	tc 5	x			x	x	x		x	x			x			x		6	9	0.666
TC54	tc 6	x			x	x	x		x			x			x	x		4	9	0.444
TC47	tc 7	x			x	x	x		x		x			x		x		5	9	0.555
TC61	tc 8	x			x	x	x		x	x			x				x	7	9	0.7777
TC68	tc 9	x			x	x	x		x		x			x			x	6	9	0.6666
TC75	tc 10	x			x	x	x		x			x			x		x	5	9	0.555

dividing the Fault Value (No of faults detected) with the Code Coverage Nodes (No of execution nodes it passed). This value shows the strength of each of the test cases and we would be able to list them in descending order of priority.

$$\text{Priority Weight (PW)} = \frac{\text{Fault Value}}{\text{Code Coverage Nodes}}$$

Once the priority weight for all the test cases have been derived, as shown in **TABLE: 3** – Showing No of Faults, Code coverage nodes and Priority Weight (PW), we are now able to rearrange the minimized test cases according descending order of priority.

TABLE 3: Derived priority weights for all test cases

Minimized TCs	TC Index	Number of faults	Coverage Node	Priority Weight
TC5	tc 1	4	6	0.666
TC19	tc 2	6	9	0.666
TC26	tc 3	5	9	0.555
TC33	tc 4	4	9	0.444
TC40	tc 5	6	9	0.666
TC54	tc 6	4	9	0.444
TC47	tc 7	5	9	0.555
TC61	tc 8	7	9	0.7777
TC68	tc 9	6	9	0.6666
TC75	tc 10	5	9	0.555

The best, efficient and most effective test cases would then be executed first as arranged in order of priority.

TABLE: 4 – Showing the prioritized test cases and all its corresponding values used in achieving the prioritization.

TABLE 4: Prioritized Test Cases arranged in descending order

Prioritization TCs	TC Index	Number of faults	Coverage Node	Priority Weight
TC61	tc 1	7	9	0.7777
TC68	tc 2	6	9	0.6666
TC5	tc 3	4	6	0.666
TC19	tc 4	6	9	0.666
TC40	tc 5	6	9	0.666
TC26	tc 6	5	9	0.555
TC47	tc 7	5	9	0.555
TC75	tc 8	5	9	0.555
TC33	tc 9	4	9	0.444
TC54	tc 10	4	9	0.444

IX. RESULTS

Test Case fault detection effectiveness, was evaluated by a metric called Average Percentage of Fault Detected (APFD). To calculate APFD value, we needed to consider the index of test cases after prioritization in minimized test suite. If T be a test suite with n number of test cases, F be a set of m faults detected by T, and TF_i be the first test case index in ordering T that reveals fault i. The following equation shows the APFD value for prioritizing T.

$$APFD = 1 - \frac{TF1 + TF2 + \ldots\ldots + TFm}{nm} + (1 / 2n)$$

Researchers have been using this metric for evaluating their prioritization techniques and found that it produces very significant result [6]. To find the average number of faults detected in each test suite APFD metric is used significantly.

APFD values range from 0 to 1 and its percentage shows how long the faults have been covered. In our paper the APFD metric value before prioritization is 0.975, and the APFD value after prioritization is 0.9769.

Previous Test Case Order:
TC1, TC2, TC3, TC4, TC5, TC6, TC7, TC8, TC9, TC10

APFD value: 1- (1*3+1+2+2*4+3*3+4*2+8) / (10*52) + 1 / (2*10) = 0.975

After Prioritization Test Case Order:
TC8, TC9, TC1, TC2, TC5, TC3, TC7, TC10, TC4, TC6.

APFD value: 1- (3*3+1+1+4*1+3*2+2*8+1) / (10*52) + 1 / (2*10) = 0.9769

The following graph shows the APFD value comparison for both prioritized and non-prioritized test suites.

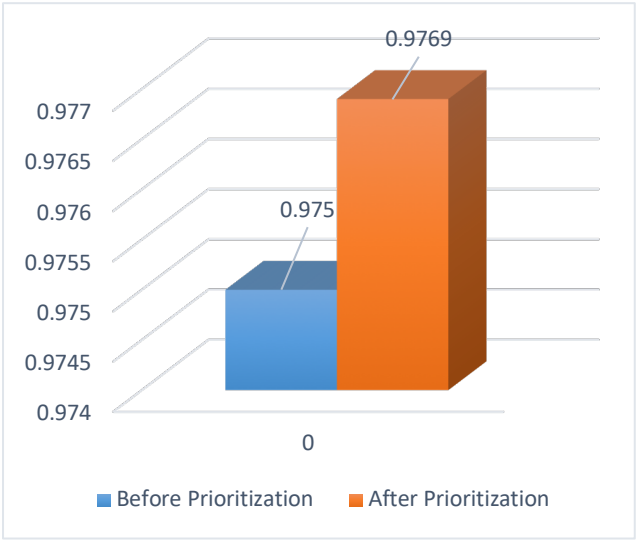


Figure: APFD metric value for test suites

The above graph shows that more faults can be detected when test cases are prioritized rather than random execution.

X. CONCLUSION

XI. ACKNOWLEDGMENT *(HEADING 5)*

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g." Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

XII. REFERENCES

[1] T. Rothvo. Directed Steiner tree and the Lasserre hierarchy. CoRR, abs/1111.54-73, 2011. [10

[2] Bang Ye Wu: A simple approximation algorithm for the internal Steiner minimum tree. CoRR abs/1307.3822, 2013. I.S. Jacobs and C.P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G.T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271-350.

[3] P. G. Sapna, B. Arunkumar. An Approach for Generating Minimal Test Cases for Regression Testing. Procedia Computer Science 47 (2015) 188 – 196. Elsevier B.V. March 2015

[4] T. Muthusamy, K. Seetharaman. Effectiveness of Test Case Prioritization Techniques based on Regression Testing. International Journal of Software Engineering and Application (IJSEA), No 6. November 2014

[5] -→S. Elbaum, A. Malishevsky, G. Rothernel. Test Case Prioritization: A Family of Empirical Studies. IEEE transactions on Software Engineering. 28(2), 159 – 182. February 2002

[6] Thillaikarasi Muthusamy and Dr. Seetharaman.K, Effeciveness of test case prioritization techniques based on regression testing. International Journal of Software Engineering & Applications (IJSEA), Vol.5, No.6, November 2014.

[7] Ahlam Ansari , Anam Khan , Alisha Khan, Konain Mukadam, Optimized Regression Test using Test case Prioritization. 7th International Conference on Communication, Computing and Virtualization 2016.

[8] Martin Pedemonte, Francisco Luna , Enrique Alba. A systolic Genetic Search for reducing the exdcution cost of regression testing. Applied Soft Computing 49 (2016)1145 – 1161.

[9] Mohamed A Shameem and N Kanagavalli.2013. Dependency Detection for Regression Testing using Test Case Prioritization Techniques. International Journal of Computer ApplicationsVol 65(14): pp:20-25.

[10] M. Yoon, E. Lee, M. Song and B. Choi.2012. A Test Case Prioritization through Correlation of Requirement and Risk. Journal of Software Engineering and Applications. Vol. 5 No. 10. pp. 823835. doi: 10.4236/jsea.2012.510095.

[11] R. Abreu, P. Zoeteweij, A.J.C. van Gemund.2009. Spectrum-based multiple faultlocalization, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 88–99.

[12] R. Krishnamoorthi and S. A. Mary.2009.Factor Oriented Requirement Coverage Based System Test Case Prioritization of New and Regression Test Cases. Information and Software Technology. Vol. 51.No. 4. pp. 799-808.

[13] S. Raju and G.V. Uma.2012. An Efficient method to Achieve Effective Test Case Prioritization in Regression Testing using Prioritization Factors. Asian Journal of Information Technology. Vol:11.issue:5.pp:169-180.DOI: 10.3923/ajit.2012.169.180

[14] R. Ramler, S. Biffl and P. Grunbacher "Value-Based Management of Software Testing",Book Chapter.

[15] L. Zhang, S. S. Hou, C. Guo, T. Xie and H. Mei "Time Aware Test- Case Prioritization using Integer Linear Programming", ISSTA'09 , Chicago, Illinois,USA, Jul 2009

[16] Z. Li, M. Harman and R. M. Hierons "Search Algorithms for Regression Test Case Prioritization",IEEE Trans. on Software Engineering, vol. 33, no. 4, Apr 2007

[17] Leung HKN, White L. Insights into testing and regression testing global variables. Journal of Software Maintenance 1990; 2(4):209–222.

[18] Laski J, Szermer W. Identification of program modifications and its applications in software maintenance. Proceedings of the International Conference on Software Maintenance (ICSM 1992), IEEE Computer Society Press, 1992; 282–290.

[19] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, "Test Case Prioritization: An Empirical Study," Proc. Int'l Conf. Software Maintenance, pp. 179-188, Sept. 1999.

[20] S. Elbaum, A. Malishevsky, and G.Rothermel Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering, February 2002.

[21] Z. Li, M. Harman, and R. M. Hierons. Search Algorithms for Regression Test Case Prioritization, IEEE Transaction on Software Engineering, vol. 33, no. 4, pp. 225-237, 2007.

[22] Praveen Ranjan Srivastava, Test Case Prioritization, Journal of Theoretical and Applied Information Technology, pp. 178-181, 2008.

[23] K. K. Aggrawal , Yogesh Singh , A. Kaur, Code coverage based technique for prioritizing test cases for regression testing, ACM SIGSOFT Software Engineering Notes, v.29 n.5, September 2004.

[24] Do, H., Rothermel, G. and Kinneer, A. (2006) Prioritizing JUnit Test Cases: An Empirical Assessment and Cost- Benefits Analysis. Springer Science Empire Software Engineering, 11, 33-70.

[25] Jeffrey, D. and Gupta, N. (2007) Improving Fault Detection Capability by Selectively Retaining Test Cases during a Test Suite Reduction. IEEE Transactions on Software Engineering, 33, 108-123.

[26] Jones, J.A. and Harrold, M.J. (2003) Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Transactions on Software Engineering, 29, 195-209. http://dx.doi.org/10.1109/TSE.2003.1183927

[27] [19] Elbaum, S., Malishevsky, A.G. and Rothermel, G. (2002) Test Case Prioritization: A Family of Empirical Studies. IEEE Transactions on Software Engineering, 28, 159-182. http://dx.doi.org/10.1109/32.988497

[28] Huang, Y., 2010. Hypergraph based visual categorization and segmentation. The State University of New Jersey.

[29] Sabharwal, S., 2011. A genetic algorithm based approach for prioritization of test case scenarios in static testing. Proceedings of the 2nd International Conference on Computer and Communication Technology, Sept. 15-17, IEEE Xplore Press, Allahabad, pp: 304-309. DOI: 10.1109/ICCCT.2011.6075160

[30] Panigrahi, C.R. and R. Mall, 2012. A hybrid regression test selection technique for object-oriented programs. Int. J. Soft. Eng. Applic., 6: 17-34.

[31] Ahmed, M.A. and I. Hermadi, 2008. GA-based multiple paths test data generator. J. Comput. Operat. Res., 35: 3107-3124. DOI: 10.1016/j.cor.2007.01.012

[32] Test cases reduction and selection optimization in testing web services by Izzat Alsmadi & Sascha Alda, I.J. Information Engineering and Electronic Business, 2012, 5, 1-8, 2012

[33] Dr. D. Jeya Mala & Dr V Mohan., Quality improvement & optimization of test cases - A hybrid genetic algorithm based approach, ACM SIGSOFT Software Engineering Notes Page 1 May 2010 Volume 35 Number 3.

[34] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1999.

[35] Sapna P.G. and Hrushikesha Mohanty. Prioritization of scenarios based on UML activity diagrams. In 1st International Conference on Computational Intelligence, Communication Systems and Networks(CICSYN 2009), pages 271-276. IEEE Computer Society, 2009.

[36] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. Software Engineering,27(10):929-948, 2001.

[37] L.C. Briand, Y. Labiche, G. Soccar. Automating Impact Analysis and Regression Test Selection based on UML Designs. Proceedings of the International Conference on Software Maintenance (ICSM 2002), IEEE Computer Soceity, pp.1-10, 2002.