

# **Minimization of TS using Whole Test Suite Generation method with EA**

**(Multi-Objective Evolutionary Generation of the Whole test Suite)**



TH15407

***Submitted by:***  
**Sadia Ashraf**  
**317-FBAS/MSSE/F10**



***Supervised by:***  
**Dr. Aamer Nadeem**

***Co-Supervised by:***  
**Mr. M Imran Saeed**

Department of Computer Science and Software Engineering  
Faculty of Basic and Applied Sciences  
International Islamic University Islamabad

MS  
006-3  
SAM

1. Evolutionary Computation

2. Evolutionary programming

(computer science)



Department of Computer Science and Software Engineering  
International Islamic University Islamabad

Date: \_\_\_\_\_

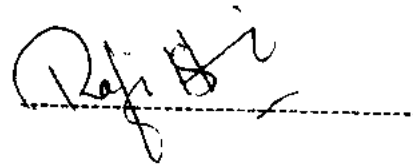
**Final Approval**

This is to certify that we have read the thesis submitted by Sadia Ashraf, 317-FBAS/MSSE/F10. It is our judgment that this thesis is of sufficient standard to warrant its acceptance by International Islamic University, Islamabad for the degree of Masters of Science in Software Engineering (MSSE).

Committee:

**External Examiner:**

Dr Rafi us Shan  
Associate Professor  
Comsat Abbottabad



**Internal Examiner:**

Mr. Muhammad Nasir  
Lecturer  
Department of Computer Science  
IUI



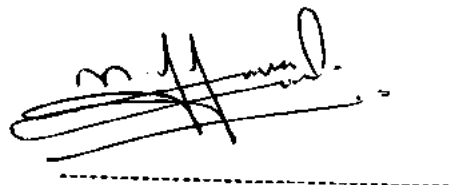
**Supervisor:**

Dr. Aamer Nadeem  
Head of Department  
Department of Bio Informatics  
Muhammad Ali Jinnah University



**Co-supervisor:**

Mr. Imran Saeed  
Assistant Professor  
Department of Computer Science  
& Software Engineering  
IUI



*Dedication...*

*To my family especially my father, mother and sister  
who are an embodiment of Diligence and Honesty,  
without their Prayers and Support  
this dream could have never  
come true.*

**A dissertation Submitted to  
Department of Computer Science and  
Software Engineering,  
Faculty of Basic and Applied Sciences,  
International Islamic University, Islamabad  
As a Partial Fulfillment of  
the Requirement for the Award of the  
Degree of Masters of Science in Software Engineering  
(MSSE)**

# Declaration

I hereby declare that this thesis "*Minimization of TS using Whole Test Suite Generation method with EA (Multi-Objective Evolutionary Generation of the Whole test Suite)*" neither as a whole nor as a part has been copied out from any source. It is further declared that I have done this research with the accompanied report entirely on the basis of my personal efforts, under the proficient guidance of my teachers, especially my supervisor *Dr. Aamer Nadeem*. If any part of the system is proved to be copied out from any source or found to be a reproduction of any project from any of the training institute or educational institutions, I shall stand by the consequences.

---

**Sadia Ashraf**  
**317-FBAS/MSSE/F10**

# Acknowledgement

In the name of Allah, the passionate, whose blessings made it possible for me to complete this complex and hard task. Its completion is a matter of great enthusiasm and pleasure for me. It is all because of Almighty Allah's guidance that made me so able.

I am fortunate enough that a masterful intellect, in the mind of my supervisor Dr. Aamer Nadeem, was with me. I offer my sincerest gratitude to him, who has supported me throughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. I attribute the level of my Masters degree to his encouragement and effort. I have no words to thank the laborious and tiring contributions of this extraordinary personality. One simply could not wish for a better supervisor.

I wish to express my deepest gratitude to Mr. Imran Saeed for his worthy support and kind cooperation particularly near the completion of my thesis. I thank the members of my graduate committee for their worthy comments and valuable criticism. I am also grateful to my friends and colleagues for their love and encouragement.

It will be failing in my duties if I miss to thank my beloved family. I am indebted to my parents and would like to express my deepest gratitude to them for their constant encouragement, affection and motivation. Their prayers always contribute a lot in completing difficult tasks. It is due to their unexplainable care and love that I am at this position today.

My sister deserves my special appreciation for providing me her amusing companionship during her tight and strained schedules, after which I always feel fresh and relaxed. She always motivates me in my work whenever I got stuck in each and every step. Also appreciate her



## ACKNOWLEDGEMENT

spiritual support that would be a key point of my achievement and success. I am also thankful to my friends and batch fellows, who encourage me a lot and guide me as well during this whole time. Especially one of my friends Saba who helps me in each and every step of this research work.

---

Sadia Ashraf

317-FBAS/MSSE/F10

## Project In Brief

<b>Project Title:</b>	Multi-Objective Evolutionary Generation of the whole test Suite
<b>Undertaken By:</b>	Sadia Ashraf 317-FBAS/MSSE/F10
<b>Supervised By:</b>	Dr. Aamer Nadeem
<b>Start Date:</b>	April, 2012
<b>Completion Date:</b>	April, 2014
<b>Tools &amp; Technologies:</b>	Matlab R2015,
<b>Documentation Tools:</b>	Microsoft Office Word 2007 Draw.io Microsoft Office Excel 2007
<b>Operating System:</b>	Microsoft Windows 7, Home Premium
<b>System Used:</b>	Dell Precision M6500

## Abstract

*Due to the recent advances that were made in the field of software testing, it is now possible to automatically generate test cases for the SUT which reach almost any point in the source code. There are two basic issues with the approach of targeting one distinct coverage goal at a time, the first being that the coverage goals to be covered are inter-related and the second that it is not guaranteed that a generated test case will actually succeed in covering the required goal because some testing goals are more difficult than others and some can be infeasible too.*

*Whole test suite generation, is a search-based technique that attempts to generate Whole Test suites and optimize them using Genetic Algorithms so that the generated Test Suites fulfill the coverage criteria, instead of generated test cases for each coverage goal separately. Using this technique up to 18 times more coverage is achieved, compared to targeting single branches.*

*Whole test suite generation, generates a test suite which achieves desirable coverage evolutionally, using genetic algorithm, but the generated test Suite is not minimized and has redundant test cases in it. The Proposed Solution for this issue is to convert the whole test Suite generation technique to a multi Objective genetic algorithm to enable it to produce a minimized solution in one run.*

# Table of Contents

<b>Abstract .....</b>	<b>X</b>
<b>List of Figures .....</b>	<b>XIV</b>
<b>List of Tables .....</b>	<b>XIII</b>
<b>Acronyms and Abbreviations .....</b>	<b>XXV</b>
<b>Chapter 1. Introduction .....</b>	<b>1</b>
1.1 Automated testing .....	3
1.2 Test Data Generation .....	3
1.2.1 Random Test Data Generation.....	4
1.2.2 Symbolic Test Data Generation.....	4
1.2.3 Dynamic Test Data Generation .....	4
1.3 Test Suite Optimization.....	5
1.3.1 Minimization .....	5
1.3.2 Prioritization .....	5
1.3.3 Selection .....	6
1.4 Genetic Algorithm .....	6
1.5 Multi-Objective Genetic Algorithm .....	8
1.6 Problem Statement .....	8
1.7 Research Objective .....	8
1.8 Hypothesis .....	9
1.9 Dissertation Outline .....	9
<b>Chapter 2. Background.....</b>	<b>13</b>

## TABLE OF CONTENTS

2.1 Test Suite optimization.....	13
2.2 Technical Definitions .....	13
2.3 Types of test Cases .....	14
2.4 Test Case Selection .....	15
2.4.1 Modification Revealing test Cases .....	16
2.4.2 Modification Traversing test Cases .....	16
2.4.3 Controlled Regression Testing Assumption .....	16
2.5 Test Case Prioritization .....	17
2.5.1 Coverage Based Prioritization .....	18
2.5.2 Other Approaches.....	19
2.5.3 Cost-Aware test Case Prioritization.....	19
2.6 Test Suite Minimization .....	20
2.6.1 Impact on fault-detection capability .....	21
2.7 Test data Generation .....	21
2.7.1 Test data Generation System.....	22
2.7.2 Type of Test Data Generators .....	23
2.7.3 Importance of Path Selector .....	24
2.7.4 Problems with Automated Test Data Generation .....	25
2.7.5 Meta-Heuristic techniques for test Data Generation .....	26
2.8 Whole Test Suite Generation .....	27
2.9 Meta-Heuristic Techniques, Evolutionary Algorithms .....	29
2.9.1 History .....	31
2.9.2 Popular Meta-Heuristic Algorithms .....	31
2.9.3 Genetic Algorithms .....	33
<b>Chapter 3. Related Work .....</b>	<b>37</b>

3.1 Test Suite Generation Techniques.....	38
3.1.1 Pacheco et al .....	38
3.1.2 M.Harman et al .....	38
3.1.3 J. C. B. Ribeiro et al .....	39
3.1.4 Tonella et al .....	40
3.1.5 S. Wappler and F. Lammermann et al .....	41
3.2 Test Suite Minimization Techniques.....	42
3.2.1 Dale Blue et al.....	42
3.2.2 Dennis Jeffrey and Neelam Gupta et al.....	43
<b>Chapter 4. Problem Definition .....</b>	<b>46</b>
4.1 Issues with Regular Automated test Suite Generation and Evosuite.....	46
4.2 The Research Limitation .....	48
<b>Chapter 5. Implemented Approach for MO-WTS Generation.....</b>	<b>50</b>
5.1 Implemented Approach .....	51
5.2 Diagram of Implemented Approach .....	52
5.3 Fitness Function Formula .....	55
5.4 Chromosome Design .....	56
5.5 Algorithm of whole test Suite Generation .....	56
5.5.1 Algorithm of WTS Generation .....	57
5.5.2 Algorithm of MO-WTS Generation .....	60
5.6 Case Study used for Implemented Approach .....	68
5.7 Flow Chart of the Proposed Approach .....	70
<b>Chapter 6. Tool Implementation.....</b>	<b>68</b>

TABLE OF CONTENTS

6.1 Tool Architecture .....72

    6.1.1 Multi-Objective Genetic Algorithm Program .....72

    6.1.2 Branch Coverage and Method Coverage .....73

    6.1.3 Actual System .....73

    6.1.4 Test Data Set .....73

6.2 Components or Implemented Genetic Algorithm .....74

6.3 System Components .....74

    6.3.1 WTS Generation .....74

    6.3.2 MO-WTS Generation .....84

**Chapter 7. Results and Discussions .....100**

7.1 Experimental Design .....100

    7.1.1 Data set .....100

    7.1.2 Performance Measurement .....100

    7.1.3 Parameters Setting .....100

    7.1.14 Final Total Average Results .....107

7.2 Discussion .....110

7.3 Threats to Validity .....111

**Chapter 8. Conclusion .....114**

8.1 Conclusion .....114

8.2 Future work .....114

**References: .....118**

**Appendix A: Generated Code .....126**

**List of figures:**

- Fig 1.1: Workflow of the Genetic Algorithm
- Fig 1.2: Dissertation Outline
- Fig 2.1: Example from Elbaum et al
- Fig 2.2: Workflow of a Test Data Generator
- Fig 2.3: Workflow of 'Whole test Suite Generation'
- Fig 2.4: Pseudo-code of 'Whole test Suite Generation'
- Fig 2.5: Flow Diagram code of a Genetic Algorithm
- Fig 5.1: Steps of the implemented approach
- Fig 5.2: WTS Gen Algorithm Initial Population
- Fig 5.3: WTS Gen Algorithm Selection
- Fig 5.4: WTS Gen Algorithm Crossover
- Fig 5.5: WTS Gen Algorithm Mutation
- Fig 5.6: WTS Gen Algorithm Main Program flow
- Fig 5.7: MO-WTS Gen Algorithm Initial Population
- Fig 5.8: MO-WTS Gen Algorithm Fast Non-Dominant Sort
- Fig 5.9: MO-WTS Gen Algorithm Crowding Distance
- Fig 5.10: MO-WTS Gen Algorithm Selection
- Fig 5.11: MO-WTS Gen Algorithm Crossover
- Fig 5.12: MO-WTS Gen Algorithm Mutation
- Fig 5.13: MO-WTS Gen Algorithm Main Program Flow
- Fig 5.14: Contraflow Diagram of the Example
- Fig 5.15: Contraflow Diagram of the Multi-Objective Whole test Suite Generation Approach
- Fig 6.1: Architecture of the implemented approach.
- Fig 6.2: Main page of the tool.
- Fig 6.3: Zoomed View of the Tool.
- Fig 6.4: Initialize Process
- Fig 6.5: Create test Cases.
- Fig 6.6: Create test Suites



## LIST OF FIGURES

Fig 6.7: Calculate the Fitnesses

Fig 6.8: Perform Selection

Fig 6.9: Perform Crossover

Fig 6.10: Perform Mutation

Fig 6.11: Fitness of the Child Population.

Fig 6.12: Display the Results

Fig 6.13: Main Interface

Fig 6.14: Zoomed View of the interface

Fig 6.15: Initialize the process

Fig 6.16: Create test Cases

Fig 6.17: Create test Suites

Fig 6.18: Calculate the fitnesses

Fig 6.19: Calculate the Fitness Function2

Fig 6.20: Calculate the crowding distance.

Fig 6.21: Perform Selection.

Fig 6.22: Perform Crossover.

Fig 6.23: Perform Mutation.

Fig 6.24: Performing Iterations.

Fig 6.25: Display the Results.

Fig 6.26: Calculate Average of the existing approach's results.

Fig 6.27: Calculate Average of the implemented approach's results.

Fig 7.1: Total iterations Bar Chart. Experiment 1

Fig 7.2: Total iterations Pie Chart. Experiment 1

Fig 7.3: Total Repetitions Bar Chart. Experiment 1

Fig 7.4: Total Repetitions Pie Chart. Experiment 1

## LIST OF FIGURES

- Fig 7.5: Total Iterations Bar Chart, Experiment 1
- Fig 7.6: Total Iterations Pie Chart, Experiment 1
- Fig 7.7: Iterations of both techniques.
- Fig 7.8: Pie Chart of the Iterations of both techniques
- Fig 7.9: Total repetitions Bar Chart, Experiment 2
- Fig 7.10: Total repetitions Pie Chart, Experiment 2
- Fig 7.11: Total Iterations Bar Chart, Experiment 2
- Fig 7.12: Total Iterations Pie Chart, Experiment 2
- Fig 7.13: Total Iterations Bar Chart, Experiment 2
- Fig 7.14: Total Iterations Pie Chart, Experiment 2
- Fig 7.15: Iterations of both techniques.
- Fig 7.16: Pie Chart of the Iterations of both techniques.
- Fig 7.17: Total repetitions Bar Chart, Experiment 3
- Fig 7.18: Total repetitions Pie Chart, Experiment 3
- Fig 7.19: Total Iterations Bar Chart, Experiment 3
- Fig 7.20: Total Iterations Pie Chart, Experiment 3
- Fig 7.21: Total Iterations Bar Chart, Experiment 3
- Fig 7.22: Total Iterations Pie Chart, Experiment 3
- Fig 7.23: Iterations of both techniques.
- Fig 7.24: Pie Chart of the Iterations of both techniques.
- Fig 7.25: Total repetitions Bar Chart, Experiment 4
- Fig 7.26: Total repetitions Pie Chart, Experiment 4
- Fig 7.27: Total iterations Bar Chart, Experiment 4
- Fig 7.28: Total iterations Pie Chart, Experiment 4
- Fig 7.29: Total iterations Bar Chart, Experiment 4

## LIST OF FIGURES

- Fig 7.30: Total iterations Pie Chart, Experiment 4
- Fig 7.31: Iterations of the both techniques, Experiment 4
- Fig 7.32: Pie Chart of the Iterations of the both techniques, Experiment 4
- Fig 7.33: Total repetitions Bar Chart, Experiment 5
- Fig 7.34: Total repetitions Pie Chart, Experiment 5
- Fig 7.35: Total iterations Bar Chart, Experiment 5
- Fig 7.36: Total iterations Pie Chart, Experiment 5
- Fig 7.37: Total iterations Bar Chart, Experiment 5
- Fig 7.38: Total iterations Pie Chart, Experiment 5
- Fig 7.39: Iterations of the both techniques, Experiment 5
- Fig 7.40: Pie Chart of the Iterations of the both techniques, Experiment 5
- Fig 7.41: Total repetitions Bar Chart, Experiment 6
- Fig 7.42: Total repetitions Pie Chart, Experiment 6
- Fig 7.43: Total iterations Bar Chart, Experiment 6
- Fig 7.44: Total iterations Pie Chart, Experiment 6
- Fig 7.45: Total iterations Bar Chart, Experiment 6
- Fig 7.46: Total iterations Pie Chart, Experiment 6
- Fig 7.47: Iterations of the both techniques, Experiment 6
- Fig 7.48: Pie Chart of the Iterations of the both techniques, Experiment 6
- Fig 7.49: Total repetitions Bar Chart, Experiment 7
- Fig 7.50: Total repetitions Pie Chart, Experiment 7
- Fig 7.51: Total iterations Bar Chart, Experiment 7
- Fig 7.52: Total iterations Pie Chart, Experiment 7
- Fig 7.53: Total iterations Bar Chart, Experiment 7
- Fig 7.54: Total iterations Pie Chart, Experiment 7

## LIST OF FIGURES

- Fig 7.55: Iterations of the both techniques, Experiment 7
- Fig 7.56: Pie Chart of the Iterations of the both techniques, Experiment 7
- Fig 7.57: Total repetitions Bar Chart, Experiment 8
- Fig 7.58: Total repetitions Pie Chart, Experiment 8
- Fig 7.59: Total iterations Bar Chart, Experiment 8
- Fig 7.60: Total iterations Pie Chart, Experiment 8
- Fig 7.61: Total iterations Bar Chart, Experiment 8
- Fig 7.62: Total iterations Pie Chart, Experiment 8
- Fig 7.63: Iterations of the both techniques, Experiment 8
- Fig 7.64: Pie Chart of the Iterations of the both techniques, Experiment 8
- Fig 7.65: Total repetitions Bar Chart, Experiment 9
- Fig 7.66: Total repetitions Pie Chart, Experiment 9
- Fig 7.67: Total iterations Bar Chart, Experiment 9
- Fig 7.68: Total iterations Pie Chart, Experiment 9
- Fig 7.69: Total iterations Bar Chart, Experiment 9
- Fig 7.70: Total iterations Bar Chart, Experiment 9
- Fig 7.71: Iterations of the both techniques, Experiment 9
- Fig 7.72: Pie Chart of the Iterations of the both techniques, Experiment 9
- Fig 7.73: Total repetitions Bar Chart, Experiment 10
- Fig 7.74: Total repetitions Pie Chart, Experiment 10
- Fig 7.75: Total iterations Bar Chart, Experiment 10
- Fig 7.76: Total iterations Pie Chart, Experiment 10
- Fig 7.77: Total iterations Bar Chart, Experiment 10
- Fig 7.78: Total iterations Pie Chart, Experiment 10
- Fig 7.80: Pie Chart of the Iterations of the both techniques, Experiment 10

## LIST OF FIGURES

- Fig 7.81: Cumulative Results bar Chart
- Fig 7.82: Implemented Approach Pie Chart
- Fig 7.83: Existing Approach Pie Chart
- Fig 7.84: Average Cumulative results Bar chart
- Fig 7.85: Average Cumulative results Pie chart
- Fig A.1: Main page of the tool
- Fig A.2: Zoomed View of the Tool
- Fig A.3: Initialize Process
- Fig A.4: Create test Cases
- Fig A.5: Create test Suites
- Fig A.6: Calculate the Fitnesses
- Fig A.7: Perform Selection
- Fig A.8: Perform Crossover
- Fig A.9: Perform Mutation
- Fig A.10: Fitness of the Child Population
- Fig A.11: Display the Results
- Fig A.12: Main Interface
- Fig A.13: Zoomed View of the interface
- Fig A.14: Initialize the process
- Fig A.15: Create test Cases
- Fig A.16: Create test Suites
- Fig A.17: Calculate the Fitnesses
- Fig A.18: Calculate the Fitness Function2
- Fig A.19: Calculate the crowding distance.
- Fig A.20: Perform Selection.

## LIST OF FIGURES

Fig A.21: Perform Crossover

Fig A.22: Perform Mutation

Fig A.23: Performing Iterations.

Fig A.24: Display the Results.

Fig A.25: Calculate Average of the existing approach's results

Fig A.26: Calculate Average of the implemented approach's results.

**List of tables:**

Table 7.1: Parameters of GA

Table 7.2: Experiment 1 Existing approach Readings

Table 7.3: Experiment 1 Implemented approach Readings

Table 7.4: Experiment 1 Cumulative Results

Table 7.5: Experiment 2 Existing approach Readings

Table 7.6: Experiment 2 Implemented approach Readings

Table 7.7: Experiment 2 Cumulative Readings

Table 7.8: Experiment 3 Existing approach Readings

Table 7.9: Experiment 3 Implemented approach Readings

Table 7.10: Experiment 3 Cumulative Readings

Table 7.11: Experiment 4 Existing approach Readings

Table 7.12: Experiment 4 Implemented approach Readings

Table 7.13: Experiment 4 Cumulative Readings

Table 7.14: Experiment 5 Existing approach Readings

Table 7.15: Experiment 5 Implemented approach Readings

Table 7.16: Experiment 5 Cumulative Readings

Table 7.17: Experiment 6 Existing approach Readings

Table 7.18: Experiment 6 Implemented approach Readings

Table 7.19: Experiment 6 Cumulative Readings

Table 7.20: Experiment 7 Existing approach Readings

## LIST OF TABLES

Table 7.21: Experiment 7 Implemented approach Readings

Table 7.22: Experiment 7 Cumulative Readings

Table 7.23: Experiment 8 Existing approach Readings

Table 7.24: Experiment 8 Implemented approach Readings

Table 7.25: Experiment 8 Cumulative Readings

Table 7.26: Experiment 9 Existing approach Readings

Table 7.27: Experiment 9 Implemented approach Readings

Table 7.28: Experiment 9 Cumulative Readings

Table 7.29: Experiment 10 Existing approach Readings

Table 7.30: Experiment 10 Implemented approach Readings

Table 7.31: Experiment 10 Cumulative Readings

Table 7.32: Average of the Results

Table 7.33: Final Sum-up of Readings

Table 7.34: Comparison of the Readings



## Acronyms and Abbreviations

SUT	System Under test
CUT	Class Under Test
UML	Unified Modeling Language
EA	Evolutionary Algorithm
GA	Genetic Algorithm
SDLC	System Development Life Cycle
IJP	Infeasible Paths
FP	Feasible Paths
SE	Software Engineering
MOGA	Multi Objective Genetic Algorithm
TS	Test Suite
WTS GEN	Whole test Suite Generation
MOWTS GEN	Multi Objective Whole test Suite Generation
APFD	Average Percentage Fault Detection
FEP	Fault Exposing Potential
ACO	Ant Colony Optimization
BCO	Bee Colony Optimization
PSO	Particle Swarm Optimization
CTD	Combinatorial test Design
CFG	Control Flow Graph
ITSM	Interaction Based Test Suite Minimization
RSR	Reduction with selective Redundancy
NSGA	Non-Dominant Sorting Genetic Algorithm



# Chapter 1

## INTRODUCTION



## INTRODUCTION:

Software testing is one of the main and final steps in the software development life cycle. Testing is important to ensure that the system works as it was intended to. Software testing is a method of assessing the functionality of a software program. Testing can be divided into many categories based on the type of system it tests, the kind of faults it is looking for (load testing, broken links etc.) or the method used for testing (automated, manual) but the main types are black box testing and white box testing.

Testing can be done either manually or automatically. Almost all software development projects need testing, which makes testing a heavily used technique, thus the testing phase of a typical project take up to 50% of the total project effort, [1] and therefor contributes significantly to the project costs. Repeated changes in the system under test can affect the results of the tests. For this reason testing has to be repeated often and this repeated testing is called regression testing. This is error-prone, time consuming and expensive. Automating the process may significantly reduce the effort needed for running individual tests. This implies that performing the same test becomes cheaper, or one can do more tests within the same budget. Manual testing is time consuming, unreliable and costly while Automated testing in contrast is reliable and requires less investment in human resources.

Test data generation in program testing, is the process of finding test cases which satisfy the chosen coverage criteria. A test data generator is a tool which is used by a programmer to generate test data for testing the SUT [11]. Two main dynamic approaches are most commonly used for generation of test cases, which achieve the required coverage. One being automatic path selection using randomly selected inputs from the input field [2] and, another is to use meta-heuristic search techniques, where SUT is treated as a problem for search optimization whose goal is to look for tests that provide as high coverage as is possible. One of these meta-heuristics, genetic algorithms, is the most widely used. Genetic Algorithms are particularly popular when used for test case generation problems because test data generation is an undecidable problem for which there exists no single optimum solution. For undecidable problem Genetic algorithms are popular because they give a near optimum solution to the problem.

In the computer science's field of artificial intelligence, the genetic algorithms can be called a digital manifestation of Darwin's theory of natural selection. This heuristic, (also sometimes called a meta-heuristic) belongs to the larger class of evolutionary algorithms (EA), which generate solutions for problems that require optimization; using techniques inspired by nature, such as Bee colony Optimization, Ant Colony Optimization, Particle Swarm Optimization, and etc. and are used to generate useful solutions to search problems [3].

When structural testing is performed, the test cases are generated based on the source code of the SUT, keeping in mind the basic aim of covering the chosen testing criteria. The recent advancements in the field of Software testing allow an efficient derivation of test data from the source code, given the size of the program being considered is reasonable. A common approach is to exercise each branch separately while generating test data for each branch individually [12], [7]. Although feasible, the major issue with this way of test data generation is that it considers all test goals of equal importance and equally reachable and it does not take collateral coverage into consideration. Unfortunately, none of these assumptions holds. This problem is manifested in many ways: Many coverage goals are impossible or infeasible, this means that there exist no test case that can fulfill this coverage goal, this lies in the category of 'undecidable infeasible path' problem [13]. In addition to infeasible path problem the test case targeting a particular branch will almost always cover addition untargeted branched unintentionally, this is called collateral coverage or serendipitous coverage [5]. This shows that the order in which the branches or goals are targeted will always affect the efficiency of the test process and the final result achieved.

Whole test suite generation, an innovative approach that overcomes the infeasible branches and collateral coverage issues by optimizing the whole test suite at once to cover the chosen coverage goals instead of targeting every single branch separately. Because the whole Test Suite coverage optimizes the entire test suite at the same time, this removes the infeasible branch coverage issue plus the choice of the order of the coverage goals does not affect the results either, the issue of collateral coverage disappears too [14].

Whole test suite generation, optimizes the test Suite to generate the final Test Suite which has the desired coverage and smaller size but the generated test Suite has redundant test cases in it. To remove redundancy from the generated test suite it is proposed that the technique should be converted to a multi-objective optimization technique. In doing this the single objective genetic

algorithm used by this technique is modified into a multi objective genetic algorithm by adding another fitness function to it which helps choose the solutions with less redundancy.

## 1.1 Automated Testing

The process of execution of pre-scripted tests or test cases on an application before its release into production by the software testing tool is automated testing (*Margaret Rouse, 2014*). Basically it means automatic generation of test data, running of the test cases on the system under test and the validation of results, using predefined oracles.

With passing time and much advancement in technology, software engineers and project managers face the ever growing challenge of producing a valid and almost fault free systems, within shrinking deadlines and lower budgets.

Hence organizations have lesser time to appropriately test there system and spend extended periods on the task. This is why they turn to automated testing which lets them test large volumes of data in lesser time and using lesser effort to do so because manual testing is error-prone and labor-intensive and it does not support the same kind of quality checks as there are possible with automated Test tools (*Elfriede Dustin, Jeff Rashka, John Paul, 2008*)

An all-encompassing testing process helps in test automation, but a suitable test (automation) plan is even more important. The test plan is responsible for decisions like, the kind of test to be performed in the various stages of Software testing and what tests among these will be done manually and which can be automated.

## 1.2 Test data generation

While testing software one often needs to know input values/parameters that will trigger a particular part of the system being tested, doing which is extremely labor-intensive if done manually. Therefore automation of the process is desirable.

To test a system, usually a test adequacy criteria is chosen, once the criteria is decided upon the next step is to create test cases that will best satisfy the chosen test criteria, for which the test cases are generated automatically to save time and effort.

It is shown through empirical research in the past, if test data is chosen based on an adequacy criteria, the fault detection capability of the test suite improves compared to choosing random test data [16, 17]. But test data that is automatically generated does not guarantee the fulfillment of the chosen adequacy criteria.

Many different types of paradigms of automatic test data generation exist but the three most common types are as follows.

### **1.2.1 Random Test Data generation**

In random test data generation the test cases are generated randomly until a suitable test case is found. This approach becomes inefficient [19] when the adequacy criteria or the program gets complex because the chances of triggering the very specific inputs that will satisfy the chosen adequacy criteria becomes lower while testing complex programs.

### **1.2.2 Symbolic Test data generation**

In symbolic execution the program is executed using symbols instead of concrete values. Symbolic execution consists of allocating symbolic values to variables in the program being tested, in order to come up with an abstract, mathematical characterization of what the program does. Thus, in an ideal case, test case generation can be converted into an algebraic expression solution problem (Christoph C. Michael, Gary E McGraw, Michael A Schatz, Curtis C Walton, 1997) [18]. There are two main issues when practically generation test data using symbolic execution, the first is the accurate representation of loops and the second is to find a way to handle pointers.

### **1.2.3 Dynamic Test data generation**

Dynamic generation is the third category of test data generation [20]. In this paradigm parts of the program under test are treated as functions and the program is run until a certain location is reached and one or more values of that location are recorded. These values are treated as if they were the value of the functions. Such functions are not easy to write but they can calculate the input to reach any location in the program and finding a function that gives the minimal value for the input to satisfy the adequacy criteria is usually possible. In this way the test data generation problem is converted to "function minimization" problem.



### 1.3 Test Suite optimization

A test suite is a collection of test cases meant to exercise a particular component of the SUT, where the final state of one test is often a precursor or the initial state of the next test (*D. Jeya Mala, V. Mohan, 2009*)[21]. While optimizing a test Suite, the aim is to generate effective test data that can exercise the adequacy criteria, consuming minimum possible resources. Following are the three main test Suite optimization methodologies in literature.

#### 1.3.1 Minimization

Minimization is one of the three typical test Suite optimization methods. The formal definition of Test Suite Minimization is as follows [22].

**Given:** A set of test requirements  $r_1, \dots, r_n$ , that must be satisfied to provide the desired testing coverage of the program, a test suite  $T$  and subsets of  $T$ ,  $T_1, \dots, T_n$ , one linked with each of the  $r_i$ 's such each test case exercises or fulfills some requirement.

**Problem:** Find a test suite  $T'$ , of test cases from  $T$  that fulfills all the requirements  $r_i$ .

The testing criterion is satisfied when every test requirement in  $r_1, \dots, r_n$  is covered. A test requirement,  $r_i$  is satisfied by any test case,  $t_j$ , that belongs to the test suite  $T_i$ , which is a subset of  $T$ . Therefore, the representative set of test cases is the hitting set of the  $T_i$ 's. Additionally, in order to make the most of the effect of minimization,  $T'$  should be the smallest possible hitting set of the  $T_i$ 's (*S Yoo, M Herman, 2010*)

#### 1.3.2 Prioritization

The Formal definition for Test Suite prioritization is given below. This is a definition from Yoo et al (2010)

**Given:** A test suite,  $T$ , the set permutations of  $T$ ,  $PT$ , and a function from  $PT$  to real numbers,  $f : PT \rightarrow \mathbb{R}$ .

**Problem:** To find  $T' \in PT$  such that  $(\forall T'')(T'' \in PT)(T'' \neq T') \{f(T') \geq f(T'')\}$ . Average Percent of Fault-Detection (APFD) is commonly used to evaluate test case prioritization techniques. The APFD value for  $T'$  is calculated as follows (Sebastian Elbaum, e Alexey Malishevsky, 2002):  $APFD = 1 - ((T' F_1 + \dots + T' F_m) / nm) + (1/2n)$  [23].

### 1.3.3 Selection

The formal definition of Test Case Selection is given below.

**Given:** The program,  $P$ , the modified version of  $P$ ,  $P'$  and a test suite,  $T$ .

**Problem:** Find a subset of  $T$ ,  $T'$ , with which to test  $P'$

In a given test suite, it can be said that the selection techniques in general aim to find the 'modification-traversing' test cases. The details of how each technique goes about the process of searching and identifying these test cases differ but the basic underlying idea remains the same. (Gregg Rothermel, e Mary Jean Harrold, 1997)[23].

## 1.4 Genetic Algorithm

There are many different types of Evolutionary Algorithms: Evolutionary Algorithms are Algorithms that mimic natural phenomena to solve problems. Genetic Algorithm is a variant of Evolutionary Algorithms that mimics the process of evolution from nature. The basic idea is that given a population of individuals, crossover and mutation is performed on the population to find the fittest solution as described by the fitness function. The basic genetic Algorithm has the following three main steps [25].

### Selection

Selection is the first step in the genetic Algorithm. The main purpose of this step is to improve the overall fitness of our population, for which purpose fittest solutions are chosen for further

operations while less fit individuals are discarded. There are many different types of selection procedures but the main idea is the same, i.e. to select the fitter individuals for the next generation.

### Crossover

In this part the selected individuals are combined to produce the offsprings. pairs of parent population are made to produce offsprings. this step is performed to combine fit parents to produce even fitter child population and eventually a solution that fulfills the stopping criteria of the algorithm.

### Mutation

Mutation is performed to add randomness to the population otherwise, with each passing iteration, the solutions might continue repeating without any improvement, this is done by making small random changes in the individual solutions.

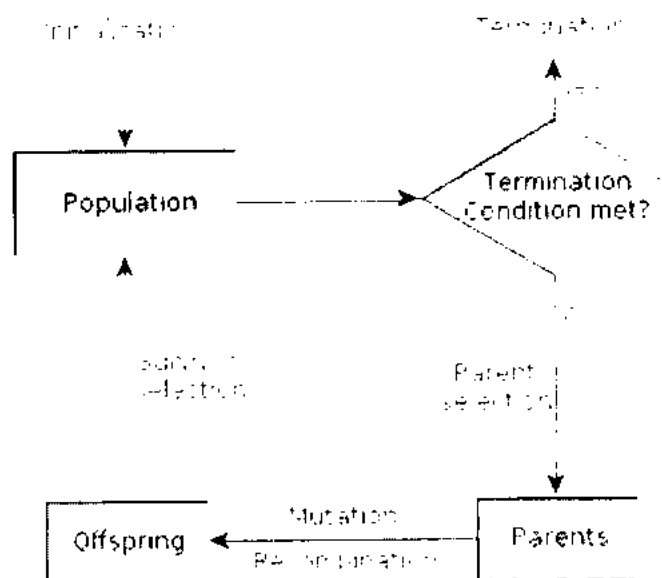


Fig 1.1 Workflow of the Genetic Algorithm

## 1.5 Multi-objective Genetic Algorithm

Multi Objective Genetic Algorithms are Genetic algorithms that are used when there is more than one objective to satisfy. An example of such problem is that when designing a bicycle we want the design to be durable but use as little material as possible or if a car manufacturing company wants to find out a reasonable balance between the cost of manufacturing and the luxury of the car. Generally the methods used to solve problems with multiple objectives can be divided into two categories. The first one is to convert the multiple objectives into a single representative formula using weights and the second technique is to find the entire representative Pareto optimal set of solutions. A Pareto optimal consists of solutions that are non-dominated considering all the objectives to be fulfilled. (*Abdullah Konak , David W. Coit , Alice E. Smith, 2005*)[26].

## 1.6 Problem Statement

In literature there are many Test Suite generation techniques that automatically generate the Test Suite but almost all the techniques are based on the coverage criteria and generate test cases for all coverage goals separately. Whole test suite generation, automatically generates the whole Test Suite for the System under Test using evolutionary algorithm. This overcomes the problems of infeasible test coverage goals and collateral coverage at the same time but the final Test Suite which is generated after running the genetic algorithm may contain redundant test cases. This means that this Test Suite can be further Optimized by reducing the number of test cases in the test suite.

## 1.7 Research Objective

The method proposed is an improvement to the Automated "Whole Test Suite Generation" technique which generates the whole test Suite using the evolutionary algorithm. Currently the Test Suite is being generated using "Single objective Genetic Algorithm", this produces a Test Suite with redundant Test Cases and the test suite is run again to remove the redundant Test cases. The proposed solution is to modify the Genetic algorithm by adding another fitness function to it and converting it to a Multi Objective Genetic Algorithm (M.O.G.A). The multi

Objective genetic algorithm to be implemented will be the Non Dominated Genetic algorithm (N.S.G.A).

## 1.8 Hypothesis

The hypothesis of this thesis is that by applying the whole test Suite generation technique Evosuite, using Multi-objective Algorithm, the coverage objective can be achieved along with the minimization objective.

## 1.9 Dissertation Outline

**Figure 1.2** illustrates the structure of this thesis:

**Chapter 2:** Describes the background for understanding the thesis by giving introductory knowledge about Test Suite Optimization, Whole Test Suite generation and Evolutionary Algorithm (Genetic Algorithm). -

**Chapter 3:** Describes work in literature to support MO-WIS Generation and Evolutionary algorithms.

**Chapter 4:** Elaborates the hypothesis that we will attempt to prove.

**Chapter 5:** Describes the implementation of the approach for MO-WIS Gen.

**Chapter 6:** Introduces and describes the implemented tool for MO-WIS Gen.

**Chapter 7:** Discusses compares and evaluates the results of our work with the existing work.

**Chapter 8:** Outlines the conclusion of this work and deliberates the findings.

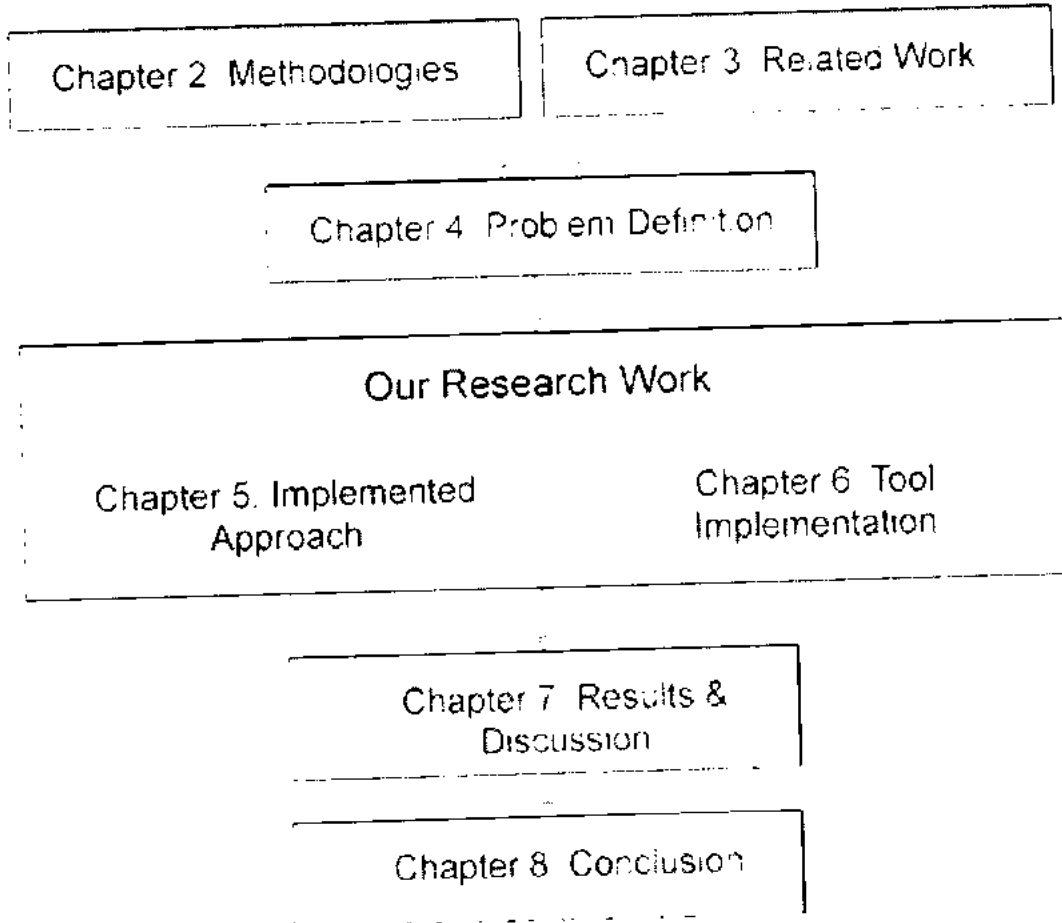


Fig 1.2. Dissertation Outline



# **Chapter 2**

# **BACKGROUND**



## BACKGROUND:

This section is devoted to describe the background for understanding the thesis.

### 2.1 Test Suite optimization

As modifications are done in a code more test cases are added to the existing test suite, this causes the test suite to grow in size, the test therefore gets bigger with the evolution of software. The larger the test suite is, the costlier it gets to run it. To overcome this problem many techniques are studied in literature. These techniques aim to minimize the resources used while running the test suite while improving the fault detection capability. Such techniques are called test Suite optimization techniques and can be divided into three categories.

- i. **Minimization:** Attempts to remove repeating i.e. redundant test cases.
- ii. **Selection:** Attempts to identify test cases that correspond to the modification that needs to be tested.
- iii. **Prioritization:** Orders the test cases to maximize the chances of early fault detection.

The main point similar in all these techniques is that they assume the existence of a test pool which is too large in size to run the complete test suite [27]. Therefore the techniques in all three categories basically try to overcome the problems that arise due to the huge size of the test pool.

### 2.2 Technical Definitions

#### *Definition 1 (Test Suite Minimization Problem)*

**Given:** A set of test requirements  $r_1, \dots, r_n$ , that must be satisfied to provide the desired testing coverage of the program, a test suite  $T$  and subsets of  $T$ ,  $T_1, \dots, T_n$ , one linked with each of the  $r_i$ 's such each test case exercises or fulfills some requirement.

**Problem:** Find a test suite  $T'$ , of test cases from  $T$  that fulfills all the requirements  $r_i$ .

The testing criterion is satisfied when every test requirement in  $r_1, \dots, r_n$  is covered. A test requirement,  $r_i$  is satisfied by any test case,  $t_j$  that belongs to the test suite  $T_i$ , which is a subset of  $T$ . Therefore, the representative set of test cases is the hitting set of the  $T_i$ 's. Additionally, in order to make the most of the effect of minimization,  $T'$  should be the smallest possible hitting set of the  $T_i$ 's.

The minimal hitting set problem is an NP-complete problem as well as a dual problem of the 'minimal set cover problem' [28].

**Definition 2 (Test Case Selection Problem)**

**Given:** The program,  $P$ , the modified version of  $P$ ,  $P'$  and a test suite,  $T$ .

**Problem:** Find a subset of  $T$ ,  $T'$ , with which to test  $P'$ .

**Definition 3 (Test Case Prioritization Problem)**

**Given:** A test suite,  $T$ , the set of permutations of  $T$ ,  $PT$ , and a function from  $PT$  to real numbers,  $f: PT \rightarrow \mathbf{R}$ .

**Problem:** To find  $T' \in PT$  such that  $(\forall T'')(T'' \in PT)(T'' \neq T') \{ f(T') \geq f(T'') \}$ .

## 2.3 Types of Test Cases

According to Leung and White Test cases are of five types based on how they are used in optimization [29].

- i. **Reusable:** These are the test cases that exercise the part of the program or system under test that is unmodified. This part does not change in the newer versions of the System under Test (SUT). These test cases are not executed when testing the modified part of the program but they are kept safe for possible testing the code in the future.

- ii. **Retestable:** These are the test cases that exercise the modified part of the system under test (SUT)  $P'$ . These test cases should be run again to Test  $P'$  after the modification.
- iii. **Obsolete:** The following reasons are why the test cases may become obsolete.
- The requirement they were created to test has been changed because of the change of specifications so they are not needed anymore.
  - The program has been modified and the part they test does not exist or has been modified so that new test cases are needed to test it.
  - They do not provide the required coverage of the structure of the program as before the modifications.
- iv. **New Structural:** These test cases are created to test the modified part in the System under test  $P'$  which is no longer covered by the older test cases.
- v. **New Specification:** These test cases are created to test the part of the program that has been modified or created as a result of modification of the specifications. They too exercise the modified part of the program  $P'$ .

## 2.4 Test case selection

Test case Selection and the Test Suite Minimization problem is similar in the sense that both of them attempt to reduce the size of the test Suite by elimination the unwanted test cases that do not contribute to the test goal of choice. The only difference is that Test cases selection focuses on the modified part of the system under test while the Test Suite minimization tries to reduce the test suites size by eliminating repeated coverage and it focus remains within the same version of the program.

### 2.4.1 Modification Revealing Test Cases

This concept was given by Rothermel and Harrold. A Test Case  $t$  is modification revealing [30] for  $P$  and  $P'$  if and only if  $P(t) \neq P'(t)$ . If the following two suppositions are correct then finding the Fault Revealing test cases is possible after the modification revealing test cases in  $P$  and  $P'$  are found.

- **P-Correct-for-T Assumption:** This assumption says that all the test cases in the test Suite  $T$  executed properly and provided correct results when they were run on the unmodified version of the program  $P$ .
- **Obsolete-Test-identification Assumption:** This assumption holds that for each test case  $t$  in the test suite  $T'$  it can be correctly identified if that test case has become obsolete for the modified version of the program  $P'$ [16].

With the above assumptions it is obvious that every test case in  $T$  provides a correct output for  $P$  i.e.  $P$  is fault free with regard to  $T$  and since no test case in  $T$  is obsolete for  $P$  all the test cases should give the same results for  $P$  and  $P'$ . So if a test case is modification revealing it should also be fault revealing.

### 2.4.2 Modification Traversing Test Case

A modification traversing test case is the test case that exercises new or changed part of the system under test  $P'$  or it used to execute code that has now been deleted in  $P'$ .

### 2.4.3 Controlled Regression Testing Assumption

This assumption says that all the factors that could affect the output when  $P'$  is tested with test case  $t$  except for the code are kept the same as they were when  $P$  was tested with  $t$ . In the case that the controlled regression testing supposition is true, a test case  $t$  that is not obsolete can be modification revealing only if it is also modification traversing for the modified and original code  $P$  and  $P'$ .

If P-Correct-for-T as well as Obsolete-Test-identification assumptions hold and the controlled regression testing assumption holds true too then the following relation between the subsets of

fault revealing test cases. modification revealing test cases. Modification traversing test cases and the Test Suite will be true too.

$T: T_{fr} = T_{mi} \subseteq T_{mt} \subseteq T$ . by using this assumption it is possible to omit all the test cases that dot reveal any faults.

## 2.5 Test case prioritization

This approach was introduced by wong et al[31]. In test case prioritization the test suite size us not altered but the test cases are rearranged in such a way so that any particular required benefits are improved in the early test cases so that if the execution is prematurely halted then most of the benefit of testing i.e coverage. fault detection etc are already achieved.

Harrold and Rothermel [32, 33] did more work on the technique by analyzing it in a more general context.

Test case	Fault revealed by test case									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

Fig 2.1: Example from Elbaum et al [35]

Considering the above table if the aim or prioritization is early fault detection then it is obvious that if we arrange the test case in the B-A-C-D-E formation then the early fault detection capability of the test suite will increase as compared to the original A-B-C-D-E arrangement. A closer look at the table reveals that any arrangement that starts with C-E is superior to any other arrangement because this arrangement detects maximum number of faults early in the test suite execution. With this arrangement, if the testing is stopped before the whole test suite is executed, then at least maximum possible faults will have been detected already, hence saving time and resources.

Although the example is based on fault coverage but the faults in the system under test are not known until it is actually tested for faults, so test case prioritization techniques use alternate methods from fault detection capability. The alternate goals for prioritization are meant to improve fault detection. Some different popular goals based on which the prioritization techniques are created are as follows.

### 2.5.1 Coverage based prioritization

Coverage of the structure of the system under test is the metric which is most commonly used in most test suite optimization techniques [34-40]. It is assumed that early coverage of the system under test will detect more faults earlier. So in prioritizing the test cases based on the system the real aim is to improve early fault detection of the system under test (SUT). Rothermel et al further investigated prioritization techniques[34-40], their study the same algorithms with different test coverage goals. The coverage goals considered were

- **Branch-total:** Branch total considers the total number of branches covered and prioritizes test cases according to that.
- **Branch Additional:** Branch Additional considers the additional branches covered by the test cases and prioritizes the test cases. This is an additional greedy technique.
- **Statement Total:** This considers the total number of statement covered by test cases and prioritizes the test cases.
- **Statement Additional:** This considers the additional coverage of the statements achieved by the test cases and prioritizes based on that criteria.
- **Fault Exposing potential Total:** Fault Exposing potential of a test case is calculated using mutation testing. Mutations, i.e. faults are intentionally introduced in the program and then the test cases are run on the program to calculate the fault exposing potential of the test cases [42]. The mutants(faults) exposed by the test cases are said to be killed by the test cases. The mutation score of the test cases are calculated as follows:

$$\text{Mutation Score} = \text{Killed mutants by the test case} / \text{Total Killable Mutants}$$

The techniques based on FEP prioritizes based on the test cases based on the mutation score of the test cases

- ***Fault Exposing Total Additional:*** The techniques based on this criterion prioritize the test cases based on their additional mutation score of the test cases.

While majority of the techniques in the literature use coverage of the system under test as a goal for prioritization but some techniques also use criteria other than coverage for prioritization.

### 2.5.2 Other approaches

There are relatively fewer number of prioritization techniques not based on coverage [42-45]. A distribution based technique is a technique that minimizes and prioritizes the test suite based on the distribution of the test case profiles[42]. Dissimilarity metric is a number that represents the difference between the input profiles of the two inputs of the test cases. The dissimilarity metric can be used to cluster similar test cases together. Clustering of the test cases gives us interesting information about the similarities between the test cases.

- Test cases in a cluster may indicate a set of redundant test cases. Using this information it may be possible to execute only one test case in the cluster and hence reduce the test suite eliminating redundant test cases.
- It is possible that isolated cluster indicate that the test cases contained in these clusters may induce unusual conditions which can expose errors. Using this information it might be useful to test this area of the profile more thoroughly since this area is less exercised during testing.
- Areas in the profile space with a low-density may indicate that test cases contained in this area show unusual behavior. Rearranging the test cases so that the areas that are isolated and show unusual behavior are exercised first may help expose faults earlier in the testing process.

Besides distribution based approaches there are Requirement based approaches, history based approaches and Probabilistic approaches.

### 2.5.3 Cost Aware Test Case Prioritization

Test case prioritization does not filter out test cases i.e. it does not reduce the size of the test suite which means that it is assumed that the whole test suite will be executed, which is a not practically feasible most of the times because of the resource limitations. A number of test suite prioritization techniques address this issue and propose solutions that cost aware while they prioritize the test cases [38, 46, 47, 48]. Elbaum et al improved the basic APFD metric so that it can take into consideration the severity of the fault detected and the cost incurred to execute the test cases. The new metric is called APFD<sub>C</sub>. Formal definition is as follows.

More formally, Let F be the set of m faults with severity values f<sub>1</sub> ... f<sub>m</sub> and let T be the set of n test cases with costs t<sub>1</sub> ... t<sub>n</sub>. For the purpose of ordering T, let TF<sub>i</sub> be the order of the first test case that reveals the i<sup>th</sup> fault. APFD<sub>C</sub> of T is calculated using the following formula:

$$APFD_C = \frac{\sum_{i=1}^m (t_i \times (\sum_{j=TF_i}^n t_j - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^n t_i \times \sum_{i=1}^m t_i}$$

Do and Rothermel studied some existing prioritization techniques and they studied the effect of time limitation on the cost effectiveness of six different techniques.

### 2.6 Test Suite Minimization

Test suite minimization reduce the size of the test suite to optimize it. the main idea is to remove redundant i.e. repeating test cases from the test suite so that the size of the test suite is reduced without affecting its coverage or Fault detection capacity. Test suite minimization is the **minimal hitting set problem**. A minimal hitting set problem is a problem in which one needs to find the smallest possible solution to using a given set of finite population to fulfill the criteria or goal that is to be achieved. In test suite minimization the aim is to find the smallest possible set of test cases that fulfills the chosen criteria.



Test Suite Minimization is an **NP-Complete** problem [57]. An NP-Complete problem is something for which no solution exists that can solve it in linear time. The only way an NP-Complete problem can be solved is through **heuristics**. A heuristic is an approximate solution for the problem, which works well with the situation at hand but is not a universal solution for all similar problems. All past work done on the minimization techniques can be considered as the development of Heuristics [49-52].

### **2.6.1 Impact on fault-detection capability**

Test suite optimization techniques minimization, selection and prioritization successfully reduce the size of the test suite to be executed but its effects on the fault detection capability of the system under test has been a widely discussed issue in the literature [23, 53-55]. Several studies were done to determine this effect.

Wong et al randomly generated test suites and reduced them using the ATAC tool created by Horgan and London. He used these test suites to test 10 UNIX programs. Rothermel et al generated test suites from the test cases provided in the Siemens suite. He reduced the test suite and then measured its fault detection capability using the same metrics as Wong et al. He concluded that reduction of the test suite resulted in a much smaller test suite from before that there is mathematical relation between the test suite reduction and the fault detection capability. He reported that for the 1000 plus test suites he tested the fault detection capability was severely affected by the reduction of the test suites. More than half of them lost 50% of their fault detection capability and in a few cases the drop in the fault detection capability was about 100%. A study by Yu et al too confirmed the findings of the study done by Rothermel.

### **2.7 Test data generation.**

Studies show that 50% of the cost of Software development is spent in Software testing. Efforts to reduce the cost of software testing are always going on. One way to do this is to automate the process of testing. In automated testing the test cases are generated automatically based on the

target that is to be tested. Generation of test cases is called test data generation in literature. Test data generation techniques can be divided into three categories [59].

- Random Test Data generation
- Path Oriented Test Data generation
- Goal Oriented Test data generation

### 2.7.1 Test Data Generation system

The figure below shows a typical Test data generation system, this consists of three main parts namely, program analyzer, Path selector and test data generator. With these three in place, many search based solutions can be found for the problem [60, 61].

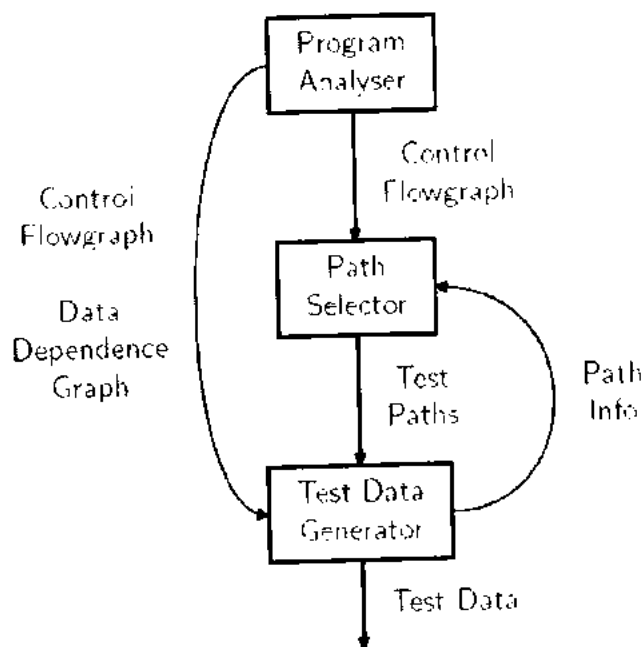


Fig 2.2 Workflow of a Test Data Generator

The automatic Test data generation problem can be defined as follows: Given a program  $P$  and an unspecified path  $u$  generate input  $x \in S$  such that  $x$  traverses  $u$ . A program analyzer is the part of the generator that runs the code and provides the result to the path selector. The path selector selects the paths based on the criteria chosen to fulfill and provides the chosen paths to the test data generator part. The test data generator part then creates test cases based on the input provided to it by the path selector.

### **Program Analyzer**

All the information about the program is provided by the program analyzer. This information contains data dependence graphs and control flow graphs etc.

### **Path selector**

A path selector selects the paths for which the test data generator will generate test data. This selection is done based on the kind of goals the test suite is meant to cover. The selected paths are then transferred to the Test data generator.

### **Test Data Generator**

A Test Data Generator generated inputs to exercise the selected paths. To do this first the path predicated of the path are found and then solved in terms of the input variables.

## **2.7.2 Type of Test Data Generators**

Test data generation techniques can be divided into the following three categories

### ***Random Test Data Generation***

This is the simplest test data generation technique. Techniques in this category generate random input values to create a test case. This data is generated without any attention to any testing goals. Randomly generated test is known to have low performance in terms of fault detection or high coverage. The reason for this is that such techniques usually miss inputs that are in trigger rare errors hidden within the less approachable parts of the program.

### ***Goal-Oriented Test Data Generation***

Goal oriented test data generation guides the test data generator to generate inputs that traverse certain paths which result in test cases that cover the test goals. The path used to generate test data in such techniques is an unspecified path. Unspecified paths are paths with some segments of the path missing from them. Two methods that use this technique are the chaining approach and the assertion oriented approach.

### ***Path-Oriented Test Data Generation***

Path oriented test data generation techniques. Like goal oriented tools it provides the generator with a single path, but the path provided in this case is a specific path. Following a specific path leads to a better coverage of goals but it makes the generation of inputs for the path harder since infeasible path can also be a part of the specific path provided to generate test data.

### **2.7.3 Importance of Path Selector**

The path selector is the part of the system that chooses the paths based on the test goals. If the path selection is done in an effective manner it improves the performance of the data generated using these paths. The stronger a criteria to be fulfilled is the more paths are needed to generate the data. Some of the typical criteria used to generate automated test data are given below.

- Branch coverage
- Statement Coverage
- Path Coverage
- Condition Coverage
- Multiple Condition Coverage

### 2.7.4 Problems with Automated Test Data Generation

Test data generation is a complex problem. Because of the complexity of test data generation most of the work in literature for test data generation results in programs that are not suitable to test real life problems.

#### Arrays and Pointers

Array and pointers create complications when generating data because they are not actual variable but they contain the address of the actual variables location. The actual variable is not known so this causes problems in the substitution and makes test data generation more complicated. A solution to this problem was proposed by Ramamoorthy[63].

#### Objects

Objects are even more complex than arrays and pointers. They are mostly dynamically allocated and there exist the concepts of abstract arrays, inheritance and polymorphism to further complicate the matters.

#### Loops

Loops are responsible for the repetition of the code in the program. Most of the times the no of these repetitions that will take place is not known, they usually depend on a variable. The loops are problematic only if they lie in the unspecific part of the programs path because to generate data for a loop it is important that the path for the closed form of the loop is present. Ramamoorthy [63] suggested a solution for this to execute the loop a randomly chosen number of times. The number is chosen either by the use or by the system.

#### Modules

Programs are usually divided into modules and functions. This causes problems because the called function's code is often not accessible because it mostly lies in precompiled libraries so a complete analysis of the function is not possible. Ramamoorthy suggested to use the inlined version of the called function to overcome this problem or to analyze the function first and generating its path separately [66].

### **Infeasible Paths**

Infeasible paths are the paths in the flow graph of the system that cannot be reached. In test data generation the test cases are generated based on the path provided by equating the path and finding inputs to traverse the paths. This becomes very difficult when the path that needs to be equated to find the input is infeasible. The only known solution in the literature for this to exercise the path a number of times before concluding that it is infeasible and that an alternate path should be taken.

### **Constraint Satisfaction**

All the testing methods have to deal with constraints but test data generation is affected by it more than other methods because there are function calls in the program and due to this symbolic execution is not possible. Search methods are usually used to solve constraints [66, 67, 63, 61, 64].

### **Oracle**

It is really important to have an oracle in automatically generated test because the number of test cases generated is usually very high plus some test cases produced might be inconceivable, but the problem lies in generating the test data's oracles. To generate an oracle either extra information about the specifications has to be provided or asserts have to be inserted in the code.

### **2.7.5 Metaheuristic techniques for test data generation**

The Meta heuristic algorithms have been used by many researchers in the recent years [69, 70]. The search based algorithms are highly adaptable and ideal for the solution of any problem that is classified as NP-Complete or even NP-Hard. These programs have been applied in the following areas of test data generation.

- To complete the coverage of a program under test, when it is being test using a white box testing strategy.
- Exercising particular parts or aspect of the System under test according to the specifications.

- To automatically falsify assertions and regarding the systems safety and disprove grey-box properties.
- To verify non-functional properties of the system under test.

Techniques like simulated annealing, hill climbing and genetic Algorithm are popular among researchers for the creation of automated test data generators.

## 2.8 Whole Test Suite generation/Evosuite

When a program is being tested under structural testing, usually some sort of coverage criterion is to be satisfied and the test cases are generated with that aim in mind. Recent advances show that now it is possible to automatically generate test cases for a moderately sized system. This approach that is used to do this is that test cases are generated to satisfy each goal after considering every goal separately. Though this strategy works, it has the following main flaws:

- This technique assumes that all the coverage goals are equally important.
- It is assumed that all the coverage goals are equally difficult to reach, while in reality many test cases are infeasible, these test cases are impossible or really difficult to reach and they take up a lot of systems time and effort to handle.
- The technique also assumes that the goals are all independent of each other but actually that is not the case. When one test case is targeted, it results in an automatic coverage of many other coverage goals. This is called collateral coverage [72] or serendipitous coverage.

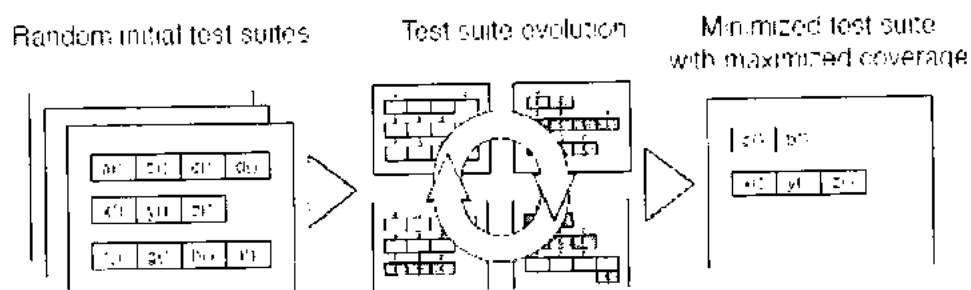


Fig 2.3: Workflow of 'Whole test Suite Generation'

Whole test suite generation/Evosuite is a technique that generates a test suite that tries to optimize the whole test suite at once without considering one coverage goal at a time. The test suites generated this way are neither affected by the order of choice of the coverage goals to satisfy nor are they affected by the infeasible parts of the system under test. Evosuite uses a search based technique to evolve a population of test suites to find a test suite that achieves coverage based on the coverage criteria of choice. This technique improves the following aspects of the test suite optimization research area [71].

- It handles dependencies among the predicates.
- It handles test case length dynamically, so that the exploration is not affected by a fixed size test cases and test suites.

Below is the pseudo-code for the whole test suite optimization software also known as Evosuite.



**Algorithm 1** The genetic algorithm applied in EVOsuite

```

1 current_population ← generate random population
2 repeat
3   Z ← elite of current_population
4   while  $|Z| \neq |current\_population|$  do
5      $P_1, P_2$  ← select two parents with rank selection
6     if crossover probability then
7        $O_1, O_2$  ← crossover  $P_1, P_2$ 
8     else
9        $O_1, O_2$  ←  $P_1, P_2$ 
10    mutate  $O_1$  and  $O_2$ 
11     $f_p = \min(\text{fitness}(P_1), \text{fitness}(P_2))$ 
12     $f_o = \min(\text{fitness}(O_1), \text{fitness}(O_2))$ 
13     $l_p = \text{length}(P_1) + \text{length}(P_2)$ 
14     $l_o = \text{length}(O_1) + \text{length}(O_2)$ 
15     $I_B$  = best individual of current_population
16    if  $f_o < f_p \vee (f_o = f_p \wedge l_o \leq l_p)$  then
17      for  $O$  in  $\{O_1, O_2\}$  do
18        if  $\text{length}(O) \leq 2 \cdot \text{length}(I_B)$  then
19           $Z \leftarrow Z \cup \{O\}$ 
20        else
21           $Z \leftarrow Z \cup \{P_1 \text{ or } P_2\}$ 
22      else
23         $Z \leftarrow Z \cup \{P_1, P_2\}$ 
24    current_population ← Z
25 until solution found or maximum resources spent

```

Fig. 2.4: Pseudo-code of 'Whole test Suite Generation' [71]

## 2.9 Meta-heuristic techniques, Evolutionary Algorithms/Genetic Algorithms

The literal meaning of the word heuristic is something that helps to learn, discover or solve any problem through experimental methods especially through trial and error. In the field of

computer science a heuristic technique or algorithm is an algorithm that finds an approximate solution for any given problem for which the classic algorithms could not find a precise solution.

Heuristic techniques provide an approximate solution for the problem at hand which means that the solution provided may not be the best solution for the problem but it is still a valuable solution because there do not exist any algorithms that would provide a definite solution for the problem yet and this solution is provided in a reasonable time frame.

Meta means beyond or something that is of a high level. So a Meta-Heuristic technique is a technique that performs better than the regular heuristics. The term Meta-Heuristic was put forward by Fred Glover (Fred 1986)[73]. A Meta-Heuristic is something that combines other heuristics to provide a better heuristic solution. Another thing common in all the Meta-Heuristic algorithms is that they all contain a search space which they traverse to find a solution and they keep randomizing the solutions to cover as much area as possible and not diverge towards the local optima.

Solutions of good quality can be found within reasonable time but there is no guarantee that the technique will definitely find the solution. i.e. such techniques don't work 100% times. All Meta-Heuristic techniques, according to (voss 2001) are suitable to global optimization.

Two main features in all the metaheuristic techniques are intensification and diversification or also known as exploitation and exploration (Blum and Roli, 2003) [74]. These two work in opposite direction and are the opposite of each other.

**Exploration/Diversification:** Diversification means to introduce randomness to the solution so that more of the search space is covered. The more diverse the solutions are the more global the focus of the search becomes.

**Exploitation/Intensification:** Intensification means to focus on one area in the search knowing that better solutions exist in that area. Too much focus on intensification can result in a solution that is locally optimal and ignoring a better solution that exists globally. A good balance between the two should be found to achieve a better solution that is capable of finding a solution through intensification and can also focus on the global optima.

### 2.9.1 History

Heuristic methods have been used during 1940s,50s and 60s in applications to solve problems but they were not used as a scientific method for optimizations. The breakthrough for the heuristic techniques came with John Holland's Genetic Algorithm. John Holland introduced the genetic algorithm in 1960 [75] after which L.J. Fogel [76] developed Genetic Programming in 1966.

In 1983 Kirkpatrick [77] developed Simulated Annealing, which was inspired by the annealing process of the metals. In annealing the metals are heated at high temperatures and then cooled down slowly to reduce their brittleness. In simulated annealing the algorithm slowly decreases its acceptance of lower fitness solutions to improve the coming populations while exploring the search space. In 80s Farmer et al developed the artificial Immune systems and the Tabu search by Glover was introduced too.

In the 90s Ant Colony optimization Algorithm by Marco Dorigo [78] was published. This algorithm was inspired by the social behavior of ants. John R Koza wrote a book on Genetic Programming and James Kennedy and Russell C. Eberhart [79] introduced the Particle Swarm Optimization. In 1997 R.Storn et al developed differential evolution which proved more effective than the genetic algorithm in many applications.

### 2.9.2 Popular Meta-heuristic Algorithms

Some of the popular Meta-Heuristic Techniques are as follows.

- *Simulated Annealing* [76]: In each iteration, the algorithm decides to move the system from the current state  $s$  to some neighboring state  $s'$  which eventually leads the system to a state of lower energy. This process is repeated until the system reaches a state that is good enough for the system.
- *Evolutionary algorithms* [75]: Of all the Meta-heuristic techniques available in literature the genetic algorithms are the most popular, the reason for this is their high adaptability, with little or no modification they can be applied to a problem of any domain. Genetic algorithms are based on the theory of natural selection by Charles Darwin. The process is

mainly divided into three steps, selection, crossover and mutation. The population of given solutions is optimized using the three steps to find the required solution i.e. the solution with the high enough fitness value.

- *Differential Evolution*: Differential evolution is a vector based version of the genetic algorithm. It does everything in terms of vectors, the population is a set of vectors. The three main steps mutation, crossover and selection are performed on the vectors.
- *Ant Colony Optimization* [78]: Ant Colony optimization is a technique that is based on the behavior of the ants when they collect food. Ants live in large colonies which can reach upto 25 million, when an ant find a food source it lays a path of pheromone to it which can be traced by other ants to and from the food source. But the trace of pheromone is not permanent; it evaporates constantly, which means that the path that is travelled more by the ants is a better food source than the one that is travelled less. This idea is used to find the best food source that is the best solution from all the available solutions.
- *Bee Colony Optimization*: Bee Colony optimization Algorithm [80] is a technique based on the behaviors of the bees when they forage for flower patches. When a bee finds good flower patch it collects the nectar and returns to the hive after which it performs a waggle dance through which it communicates the location of the food source. If there are more than one food sources available the bees divide their forces so as to maximize the nectar collected from the flowers.
- *Particle Swarm optimization* [79]: This Algorithm is based on the behavior of the swarms of fish or birds when they flock or school. The particles (solutions) are moved through the space towards a global optima which is constantly updated while other particles keep finding better solutions. At the same time the particles immediate movement is guided by its own best known location, but the particle shows random movement too. This way eventually the whole swarm is moved towards global optima.
- *Tabu Search* [81]: Tabu search uses the memory or the search history in its computations. The tabu algorithm makes a tabu list of recently visited tried solutions

which helps it to ignore the local solutions. In the long run these tabu records help save a large amount of time and hence improve the efficiency of the search.

- *Harmony Search* [82]: This algorithm is based on the musician's way of working. The musician when he has to improvise has three options. 1) He can present the piece as is without any adjustments: exactly as he remembers it 2) Play something similar but slightly adjusting the pitch a bit. 3) Create a new piece using random notes. These three steps correspond to the use of the search space, the creation of new individuals and the addition of randomness to the solutions.
- *Firefly Algorithms* [83]: Firefly Algorithm is based on the mating habits of real fireflies. A firefly is attracted to another firefly based on its brightness. The attraction between two fireflies decreases based on the increase of distance between them. So if two fireflies are considered the brighter one will attract the dimmer one but if there is no difference in their light than a random move is made. In terms of the algorithm the fireflies are the potential solutions and their brightness is determined by the position they hold in the search space.

### 2.9.3 Genetic Algorithms

Genetic Algorithms [75] are the most popular methods used for optimization. This algorithm is highly adaptable and thus can be applied to problems from many domains with ease. The basic idea behind the algorithm is based on the theory of natural selection by Charles Darwin from his book *Origin of Species*.

Genetic Algorithms are applied in many different fields to find solutions for difficult problems including, automotive design, engineering design, robotics, ships and telecommunications routing, encryption and code breaking, marketing and commerce, genetics and hardware design etc.

The main body of a Genetic Algorithm has three main parts, Selection, Crossover and Mutation. The initial population on which the genetic algorithm is run is stored in a population pool. Individuals of the population are assigned a fitness value based on how well they display

the qualities we need. A predefined number of individuals from the population are chosen based on their fitness to apply the three steps on.

*Selection:* Selection is the process in which the individual from the population are selected to apply crossover on them. The selection is done based on the fitness function of the population. The better the fitness of the individual the greater the chances of it getting chosen are. Once the individuals are chosen the next step is applied

*Crossover:* The selected individuals are crossed over to form a new set of child population. In crossover the individuals are broken and swapped at a point which is usually randomly chosen. This can be done in many ways but the simplest way is the single point crossover.

*Mutation:* This part is used to introduce randomness in the solution so that they can better cover the search space.

After mutation, the resulting child population is checked for a solution that fulfills our fitness criteria. If a solution is found, the search is halted, if not the whole process is repeated until either a required fitness is achieved or a pre-defined number of iterations are completed.

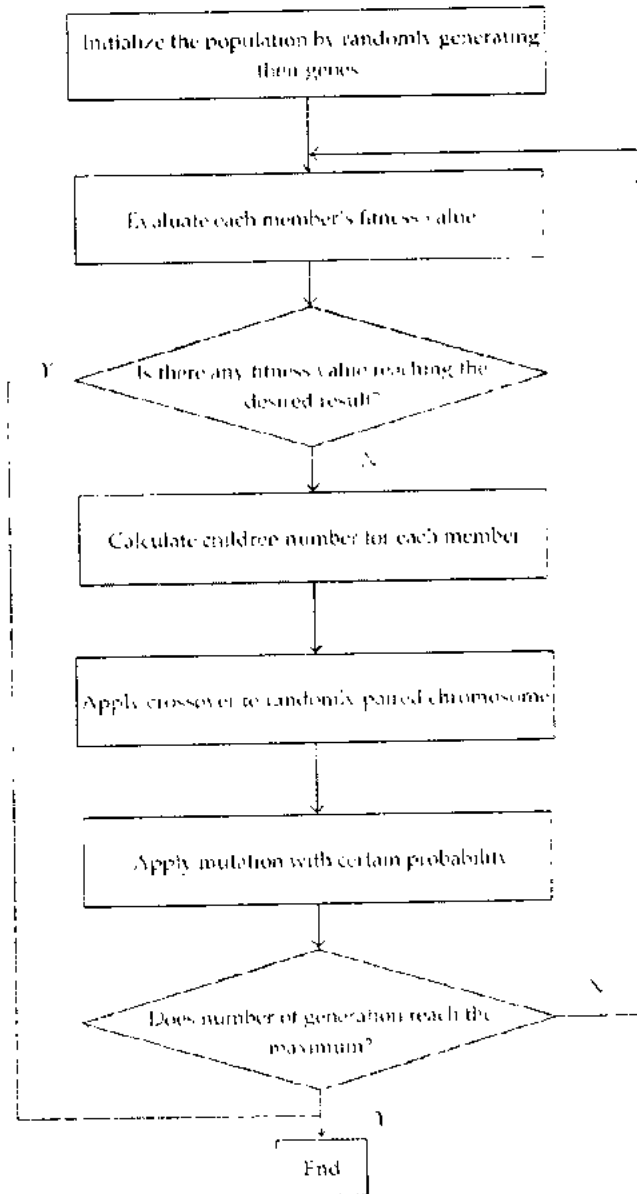


Fig 2.5: Flow Diagram code of a Genetic Algorithm





# **Chapter 3**

## **RELATED WORK**

## **RELATED WORK:**

There are numerous techniques for Test Suite generation and minimization in the literature: the techniques proposed in recent literature are listed below.

### **3.1 Test Suite Generation Techniques**

Techniques and algorithms proposed in recent literature for Test Suite generation are as follows.

#### **3.1.1 Pacheco et al (2007)**

In this work the researcher describes an automated unit test case generator called Randoop. Randoop uses a technique that is inspired by random testing which is based on execution feedback. Feedback is gathered by executing the test inputs at the time of creation. The tests generated are unit tests. RANDOOP [4] creates method sequences incrementally, by randomly selecting a method call to apply and selecting arguments from previously constructed sequences. As soon as it is created, a new sequence is executed and checked against the set of contracts. Sequences that lead to contract violations are output to the user as contract-violating tests.

Sequences that exhibit normal behavior (no exceptions and no contract violations) are output as regression tests. Finally, sequences that exhibit illegal behavior are discarded. Only normally-behaving sequences are used to generate new sequences.

#### **Limitations:**

The tool is too code dependent.

It does not perform integration testing since it generates tests for each unit separately.

#### **3.1.2 Harman et al (2010)**

The researcher in this paper takes some basic steps towards tackling the re-formulated version of the automated test data generation problem, making the following contributions:

It introduces a new way to handle the problem of search based structural test data generation, in which the main two goals are that of maximizing coverage and minimizing test suite size, while keeping in mind the fact that the human oracle costs that will incur in cases of complex Test suites and it introduces three algorithms that form the new technique for test data generation using search based testing [5]. The technique seeks to use test inputs of such a kind so that the collateral coverage is maximized along with the targeted branch coverage. In this the total number of test cases required to fulfill the coverage criteria is reduced.

#### **Limitations:**

The proposed algorithms are unable to handle infeasible branches.

#### **3.1.3 Ribeiro et al (2008)**

The focus of the researcher in this paper is to employ genetic algorithms to generate unit tests for the SUT, which is an object oriented java program. Strongly typed genetic programming is used to evolve the test cases. The results are traced using instrumentation, the objects are instrumented to track the traversal of the SUT by the generated test data. The search process gives priority to the test cases that traverse the problematic areas of the code and the control flow graph. Test objects' java Bytecode is used to perform instrumentation and static analysis of the system [6]. Important contributions consist of the introduction of innovative methodologies for automation, search guidance towards early coverage of troublesome parts and reduction of the input domain plus a tool called eCrash is presented which automatically generates test cases.

Evolutionary Computation in Java (ECJ) package is used for the representation and evolution of the test cases. Linearization of the STGP trees is used to generate the source code for the test cases. The tree linearization process produces the method call sequence; the source-code is generated by translation of the method call sequences using the method signature encoded in each node. The CFG nodes that are traversed by the test cases are removed from the uncovered nodes' list. The search ends when there are no uncovered CFG nodes left or a pre-defined no of iterations are made.

**Limitations:**

The proposed technique e-crash does not consider collateral coverage.

The tool is code dependent and focuses only on object oriented paradigm in Java.

**3.1.4 Tonella et al (2004)**

In this paper, an evolutionary algorithm is exploited to produce unit tests for classes automatically. The chromosomes that make up the test cases are the responsible for the decision of what methods need to be called, what objects are to be created and what input values are to be used. Mutation is performed on the test cases with the aim of maximizing the search space covered. Further description of the algorithm and a few implementation details are discussed below.

The basic process followed for the unit testing of classes consists of the following steps, applied to each method of the CUT and possibly repeated under different execution conditions:

1. Object creation of the class using any available constructors.
2. To bring the object to a desired state a sequence of method calls is executed.
3. The method currently being tested is called.
4. To assess the results of the test cases execution the final state of the object after the execution of tests is examined [7].

This procedure can be applied functionally (black-box testing), by deriving the expected final states from the class specifications. The thoroughness with which the testing is done can be assessed using some coverage criteria for testing. Traditional coverage criteria (white-box testing) can be used, for e.g. structural (like statement, branch) coverage or data flow (e.g., all-uses) coverage. The above steps can be repeated until the required coverage is achieved.

**Limitations:**

The proposed technique cannot handle infeasible paths or collateral coverage.

The technique focuses only on object oriented paradigm and unit testing.

**3.1.5 Wappler et al (2005)**

This paper presents an approach for the automatic generation of test data for a SUT that is object-oriented. Unit tests are generated for the system using Universal Evolutionary Algorithms. These evolutionary Algorithms are provided with popular toolboxes that are domain-independent and provide a wide range of evolutionary operators. Using the popular testing frameworks the generated test data can be converted into test classes [8].

For the purpose of using the universal evolutionary algorithms, object-oriented test programs are encoded as basic type value structures. Multi-level optimizations are considered to optimize search of the genetic algorithm. The encoding used does not void the creation of individuals which cannot be decoded back without issues. Therefore, three measures to be used by the objective function are given using which the genetic algorithm can generate more and more test classes over time that can be successfully decoded.

**Limitations:**

The encoding and decoding of elements adds complexity and does not handle inconvertible elements.

## 3.2 Test Suite minimization Techniques

Techniques and algorithms proposed in recent literature that use coverage as the basis for Test Suite minimization are as follows.

### 3.2.1 Blue et al (2013)

Combinatorial Test Design (CTD), also known as combinatorial testing, is an effective test planning technique, in which the test space is modeled by a set of parameters, their respective values, and restrictions on the value combinations. The test space represented by this model is any assignment of one value to each parameter, which does not violate the restrictions.

A subset of the space is then automatically constructed so that it covers all valid value combinations (a.k.a interactions) of every  $t$  parameters, where  $t$  is usually a user input. In other words, for every set of  $t$  parameters, any combination of  $t$  values to them will appear at least once in the test plan (unless there is no valid test that contains it, according to the restrictions). In general, one can require different levels of interaction for different subsets of parameters. The most common application of CTD is known as pairwise testing, in which the interaction of every pair of parameters must be covered. Each test in the result of CTD is an assignment of values to all the parameters, and represents a high level test, or a test scenario, that needs to be translated to a concrete executable test.

This work proposes to use Interaction-based Test-Suite Minimization (ITSM) as a complementary approach to CTD [9], for cases where standard CTD may be best practice but cannot be applied due to the requirements described above. Rather than constructing a new test suite that provides full interaction coverage, ITSM reduces an existing test suite, while preserving its interaction coverage. Similarly to CTD, ITSM requires defining the parameters of the test space and their values, but it does not require defining restrictions between the values. It is then given a test suite, where each test is in the form of an assignment of values to the parameters, and selects a subset of the test suite that preserves its  $t$ -wise value combinations.

**Limitations:**

ITSM requires that the set of existing tests be represented as tuples of values to parameters.

**3.2.2 Gupta et al (2007)**

The key step of the approach is that when a test case  $t$  is selected into a reduced suite because it satisfies an additional requirement with respect to some testing criterion  $C$ . The following is then checked: Among those other test cases  $R$  that become redundant with respect to  $C$  as a result of the selection of  $t$ , those test cases are selected from  $R$  into the reduced suite that satisfy additional requirements with respect to some other testing criterion. Thus, the approach selectively retains those test cases that are redundant with respect to initial testing criterion, if those test cases are not redundant according to some other testing criterion. The approach is called "Reduction with Selective Redundancy (RSR)" [10]. It was implemented by the researchers and experiments were conducted with several programs to evaluate and compare the effectiveness of the approach with prior experimental studies on test suite minimization.

**Limitations:**

Increases the length of the test Suite as compared to traditional minimization techniques.





# Chapter 4

## PROBLEM DEFINITION

removes each test case and check for any change in the fitness of the test suite. In case of any change in the fitness of the test suite the test cases is retained otherwise it is removed.

Main issues with this technique are as follows:

- High redundancy rate in the test suite after the Genetic algorithm and the simple minimization is run
- High rate of iterations needed to optimize and minimize the test suite.
- Optimization and minimization are both done separately i.e. after Evosuite is run, it is required to run the 'Simple minimization Algorithm' separately after that to remove the redundant Test Cases.
- Redundancy isn't completely removed even after the 'Simple minimization Algorithm' is run.
- This technique considers only one objective and its main focus is coverage not optimization.
- A need for a better more efficient optimization that considers more than one objective at the same time is needed.

*Our hypothesis* is that if instead of using the single objective genetic Algorithm if multi-Objective Genetic algorithm is used it will improve efficiency of the algorithm. The two Objectives to be considered will be coverage and minimization at the same time. We believe that this will reduce the number of iterations needed to get an optimized Test suite and it will give as output an optimized as well as minimized test suite after running the genetic algorithm once. The need to separately optimize and minimize the test suite will not be required. Thus the main aim of our work is to reduce the latency rate, the total no of iterations and the redundancy of the algorithm and improve the overall efficiency of the algorithm.

## 4.2 The Research limitation

The research gaps in the existing literature that motivated this work are listed below.

- All existing techniques are either too programming language dependent or paradigm dependent.
- The existing approach generates Test Suites with only coverage in mind and do not focus on minimization [71].
- Minimization and optimization are both done separately.
- Two techniques using multi-objective Algorithms, to generate test data but both of them do not consider collateral coverage and fail to affectively overcome the issue of infeasible paths [84, 85].

# **Chapter 5**

## **IMPLEMENTED APPROACH FOR MO-WTS GENERATION**

## IMPLEMENTED APPROACH FOR MO-WTS GENERATION

Chapter 4 and 5 include the literature survey of Multi-Objective Whole Test Suite generation, through which the gaps and shortcomings in the existing techniques are identified, which forms the basis and motivation for our work. The current chapter discusses the details of the implementation of the approach which was proposed.

The test data generation techniques in the past like Pacheco [4] et al and Ribiero [6] et al were effective and efficient methods of test data generation but most of them were meant for unit testing of the system under test and if the technique was not meant for unit testing then it either heavily code dependent or paradigm dependent. Such techniques are inflexible because of their lack of generality.

Genetic algorithms are used to generate test cases automatically in the past but all these techniques traversed the paths of the control flow graph based on the test goals, considering one goal at a time [71]. This means that there is always some collateral coverage involved and no work is done to overcome infeasible paths except the classic 'wait till a pre-defined number of attempts are made' approach.

The only approach that overcame the platform dependability, code dependability, collateral coverage and infeasible paths problem is Evosuite [71]. Evosuite is a search-based test data generation technique which has coverage as its main focus. The technique was tested on industrial large scale case studies and it showed good results but this technique does not address the issue of redundant test data. So to produce a technique that is more efficient than Evosuite and which checks collateral and infeasible coverage too Multi-Objective Whole Test Suite generation Algorithm/technique was implemented.

The main reason or achievements in implementing this technique are as follows.

- Generation of Random test data that can cover the chosen coverage criteria in the single run of the Genetic Algorithm.
- Reduce the size of the resulting Test suite while achieving a complete coverage at the same time.

- Reduce repetitive coverage as compared to Evosuite.
- The implementation gives positive results and hence is an encouragement to perform more work in this direction in the future.
- Reduce the number of repetitions required to achieve the desired coverage of the chosen test goals.
- The technique is platform, code and paradigm independent and can be easily implemented in any programming language without any modification to the basic Algorithm.
- The technique is flexible enough to be easily modifiable without losing efficiency. It can easily be implemented for a different coverage goal from the one chosen in our work and it will be just as efficient.

## 5.1 Implemented Approach

The approach that we implemented is called 'Multi-Objective Whole Test Suite Generation' and the details about this approach will be discussed in coming sections of this chapter.

The Existing Approach known as 'Evosuite' is a single objective genetic algorithm based technique that tests the system under Test using 'branch coverage' as coverage criteria. The initial population is generated randomly, which consists of a pool of Test suites. The size of the Test Suites is left random. Each Test Suite contains a random number of Test cases which too have random sizes. The random sizes are so that the limit on the size may not affect reaching the required goal of coverage, to control the bloat affect which results from randomly sized suites. A limit after the size reaches a certain maximum is kept.

A typical Test Case contains input values and the expected output but the test cases used in this tool are based on Tonella's [7] format of test cases which creates test cases with a set of random function calls, initialization statements, and constructor statements from the code under test. The code is completely object oriented so it is assumed that at some point complete coverage is possible.

Based on the fitness of the Test suites a set number of test suites are selected. These Test Suites are crossed over. In crossover the test cases are swapped, which are independent of each other as far as the code is concerned after that the off-spring population is mutated. Mutation in this approach is done one of the three ways randomly. The three kinds of mutation is called insertion, deletion and modification [71].

The whole process is repeated iteratively until a Test suite with the required fitness (i.e. 0) is achieved. Once the required fitness is achieved the algorithm is terminated and the resulting Test Suite is given as an output.

The resulting Test Suite is then minimized using a Simple minimization Algorithm to remove any redundancy that can be removed without affecting the coverage of the test suite. The 'simple Minimization Algorithms' runs the whole code to check the fitness after test suite after attempting to remove each statement iteratively. This part of the algorithm greatly increases the number of iterations required to minimize the system.

The technique that was proposed by us is multi-objective technique. The multi objective Genetic Algorithm used in the technique is the classic Multi-objective Algorithm, Non Dominant Sorting Genetic Algorithm (NSGA) [87-89]. This algorithm was proposed by Kalyanmoy Deb et al. The efficient version of the algorithm was later proposed by the same authors and it was called NSGAI. The algorithm used in this work is NSGAI [88].

The population used is the same as the existing technique for the purpose of comparison, the test Suite and test case size is still random as before, the selection part is replaced by Tournament selection and there are now two fitness functions that represent the two objectives. In the end the results obtained by both the techniques are compared to see if the hypothesis proposed was proved or disproved.

## 5.2 Diagram of Implemented Approach

- The first step is to choose the code that will be our system under test. Since this technique is a white-box testing technique we need the code to be able to test it using our technique.

The Code we selected is that of a scientific calculator. This code has 40 functions which

can form a large number of combinations when combined randomly to form a Test Suite.

- Both Evosuite and Multi-Objective Evosuite are implemented to be able to draw a fair comparison between the two approaches.
- First Evosuite is implemented in which a randomly generated population of test Suites is optimized by iteratively running the genetic algorithm on it until a suitable solution is found.
- Secondly the multi-objective Evosuite is implemented. The Multi-Objective Evosuite uses the same population as that of the Evosuite i.e. a pool of randomly generated test Suites. The first step is to calculate the fitness for all the solutions, using the fitness achieved with two fitness functions the population is divided into fronts. The first set of non-dominant solutions are put in the first front and removed from the population. The dominant solutions in the remaining population are put in the second front F<sub>2</sub> and are removed from the population pool. This process is repeated until the whole population is divided into fronts.
- After dividing the data into fronts, the crowding distance of each test Suite in each front is calculated. This crowding value and the front value show how fit a solution is.
- The next step is selection: the data is selected using tournament selection. In tournament selection a set of random data is selected and the fittest among them is chosen.
- The crossover and mutation remain the same in the multi-objective genetic algorithm too. The only difference in this phase is that now the values are chosen not on the basis of fitness alone but on the basis of the 'Crowding Comparison operator'.
- After this it is checked if a solution with a desired fitness is found. If not the child population and the parent population is combined to ensure elitism. This combined set of test Suites forms the next population. The process is repeated until a desirable solution is found.
- The two fitness functions have conflicting objectives



- The first fitness function attempts to achieve complete coverage. Therefore it attempts to find a solution that covers all the branches of the SUT.
- The second fitness function's objective is to find a test Suite that achieves the coverage with minimum possible repetitions.
- The Chromosome structure is as follows: The Main Chromosome is each Test Suite and the genes are the test cases that form the test Suite. Therefore when the crossover is applied on the genes i.e. test cases are swapped and when the mutation is performed it too deals with the test cases (modification, addition or deletion of the test cases) [71].
- The graphical flow of the process of the implementation phase is shown below.

Create a pool of Randomly generated Population of Test Suites.

Implement Evosuite, which is Whole Test Suite Generation Technique with the objective of coverage.

Implement the the proposed technique which is a multi-Objective Test Suite generation Technique.

Run both the techniques on the data from the test pool and record the results.

Compare the results obtained from running the two implemented approaches on the Test Data.

Fig 5.1: Steps of the implemented approach

### 5.3 Fitness Function Formulae

Fitness function one  $f(T)$  deals with the coverage of the test Suite. It subtracts the number of covered methods from the total number of methods, to calculate the uncovered number of methods. Then uncovered number of branches, within the method under consideration are then added to the number of uncovered methods. Using this method the uncovered methods and branches are calculated.

$$\bullet \quad f(T) = |M| - |M_T| + \sum_{bk \in B} d(bk, T)$$

$$d(b, T) = \begin{cases} 0 & \text{if the branch has been covered.} \\ \nu(d_{min}(b, T)) & \text{if the predicate has been} \\ & \text{executed at least twice.} \\ 1 & \text{otherwise.} \end{cases}$$

- $T$  is the current Test Suite
- $M$  is the Methods to be executed
- $M_T$  is no of methods covered by the Test Suite.
- $bk$  is the branch in the control flow graph.
- $B$  is the total no of branches to be executed.
- $d$  is a function that gives a normalized value for the branch distance covered within a method.

Fitness Function two  $g(T)$  is responsible for the calculation of the repeated coverage, also known as redundancy in the test Suite. It is calculated by adding the number of times functions are repeatedly covered in the test Cases. In this way all the redundant method calls within all the test cases in a test Suite are considered.

$$\bullet \quad g(T) = \sum_{tc=1}^{max} MRep_{tc}$$

- $T$  is the test Suite under consideration.

- **MRep** is the sum of the times any method is repeated within a test case.
- **tc** is the test case under consideration.

## 5.4 Chromosome design

The chromosome design in the EBNF notation is given below.

<Chromosome>:: = <Test Suites>

<Test Suites>:: = <Test Cases>

<Test Cases>:: = \*{<Function calls>|<initialization statements>|<constructors>}

<Initial Population>::= +{<Test Suites>}

<Parameters>:: = ?<No of Iterations>

<Generator>:: = \*<Test Suite>

<Fitness Value>::= <Target Method – Traversed Method>

<Target Method>:: = Code statements.

<Traversed Method>:: = Code statements.

## 5.5 Algorithms for whole test Suite generation

The Algorithm for the Existing technique, the “Whole test Suite Generation (WTS Gen)” and the implemented technique “Multi-Objective Whole test suite Generation (MO-WTS Gen)” are given below.

### 5.5.1 Algorithm for Single Objective Whole test Suite Generation is as follows:

The Single objective WTS Gen Algorithm uses the Genetic Algorithm to Generate test Suites which provide complete coverage without while overcoming the issues of collateral coverage and infeasible path coverage. The Algorithm for the WTS Gen is given below.

#### Generate Initial Population

The first step is to generate the initial population for the genetic algorithm. The initial population consists of randomly generated test suites of varying sizes. Each Test Suite Consists of a random number of Test Cases while each Test Case is a set of random function calls, constructor statements and initialization statements.

```

ARRAY AllStatements[]: (Populate array with statements from the system under Test);
.....Generate TestCases.....
FOR(i=0 TO MAXTestCase)
  ARRAY TestCase[] = RAND(AllStatements[])
END FOR

.....Generate TestSuites.....
FOR(i=0 TO MAXTestSuite)
  ARRAY TestSuite []= TestSuite[] + TestCase[RAND]
END FOR

```

Fig 5.2. WTS Gen Algorithm Initial Population

**Selection**

The second step is to select the test cases based on their fitness to form the first parent population. The no of test Suites that are used for selection is taken as an input from the user and the population size can be changed in each run.

```

.....Select the fittest Solutuions.....
FOR(i=0 TO MAXTestSuite)
  ARRAY Fitness[i] = MTotal[i] - Mcovered[i]
END FOR
FOR(i=0 TO MAXPopulation)
  ARRAY Selected[] = max(Fitness[])
END FOR

```

Fig 5.3. WTS Gen Algorithm Selection

**Crossover**

Crossover is performed on the selected Test Suites by swapping the test cases in the test suites using One point Crossover. After crossover the child population of the same size as the parent population is generated.

```

.....Crossover Selected Solutions.....
FOR (i=1 TO MAXPopulation: i=i+2)
  FOR (j=1 TO MAXTestCase)
    Child[i][j] = Selected[i][jto(j/2)]+Selected[i+1][(j/2)toMAX]
    Child[i+1][j] =Selected[i+1][jto (j/2)]+Selected[i][(j/2)to MAX]
  END FOR
END FOR

```

```
END FOR
```

Fig 5.4: WTS Gen Algorithm Crossover

### **Mutation**

Mutation in the Genetic Algorithm is meant to maintain diversity in the search space exploration. In WTS Gen the mutation is of three types. A random number is generated to decide what type of mutation should be performed. The three possible types are i. insertion ii. Deletion and iii. Modification.

```

.....Perform Mutation on the child Population.....
MutationProb = RAND(1-3)
.....Remove a Random Test Case.....
If MutationProb = 1
    MutationTestCase[] = TestCase[RAND]
    Remove MutationTestCase [RAND]
.....Add a Random Test Case.....
ELSE IF MutationProb = 2
    MutationTestCase[] = TestCase[RAND]
    Add MutationTestCase [RAND]
.....Replace a Random Test Case.....
ELSE IF MutationProb = e
    MutationTestCase[] = TestCase[RAND]
    Modify MutationTestCase [RAND]
END IF

```

Fig 5.5: WTS Gen Algorithm Mutation

### **Main Flow of Algorithm**

The Main flow of the program performs Crossover and Mutation on the selected Population in each iteration until the coverage criteria is fulfilled i.e. the fitness function  $f(T)$  becomes 0.

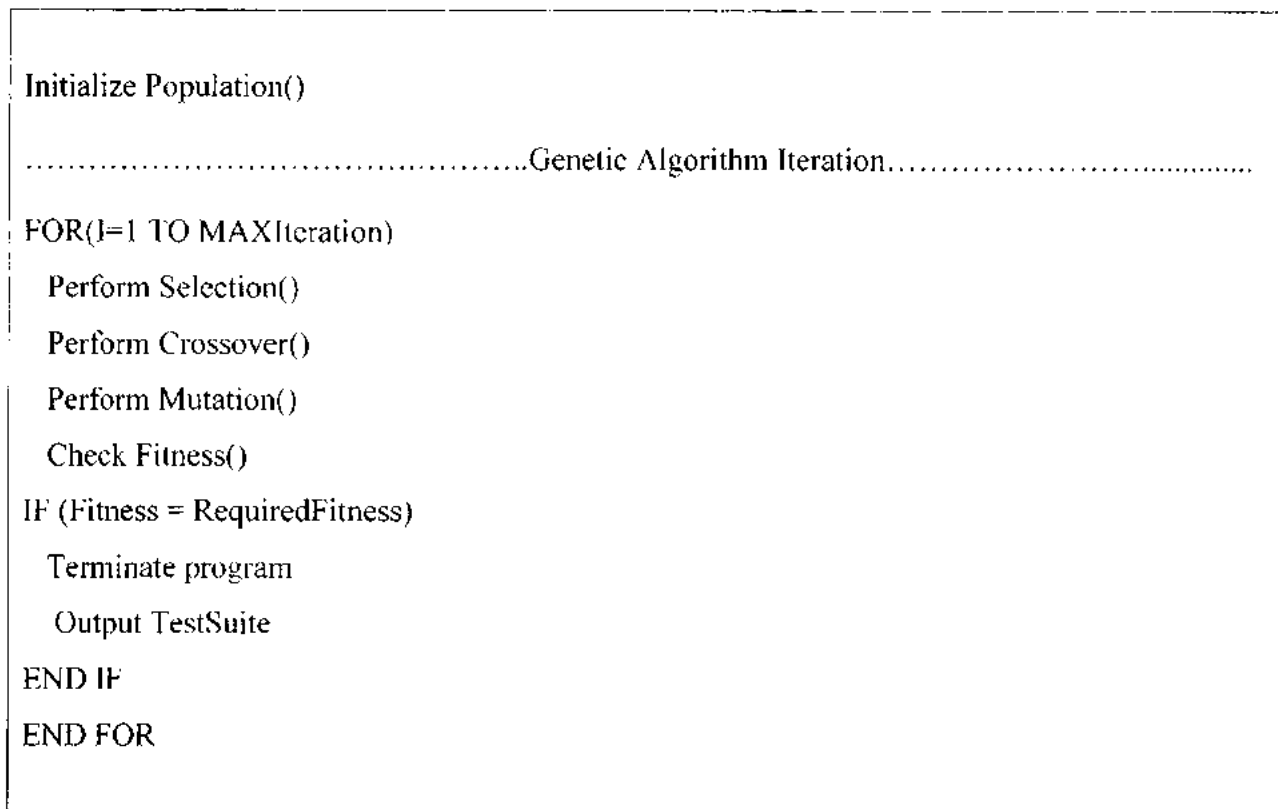


Fig 5.6: WTS Gen Algorithm Main Program flow

### **5.5.2 Algorithm for Multi-Objective Whole Test Suite Generation is as follows:**

The MO-WTS Gen Algorithm uses a Multi-Objective Test Suite based on the NSGA II Kalyanmoy Deb [35]. Two Objectives are considered instead of one, the additional objective considered here is minimization. The Algorithm is given below.

### **Generate Initial Population**

The first step is to generate the initial population for the genetic algorithm. The initial population consists of randomly generated test suites of varying sizes. Each Test Suite Consists of a random number of Test Cases while each Test Case is a set of random function calls, constructor statements and initialization statements.

```

ARRAY AllStatements[]: (Populate array with statements from the system under Test):
.....Generate TestCases.....
FOR(i=0 TO MAXTestCase)
  ARRAY TestCase[] = RAND(AllStatements[])
END FOR
.....Generate TestSuites.....
FOR(i=0 TO MAXTestSuite)
  ARRAY TestSuite []= TestSuite[] + TestCase[RAND]
END FOR

```

Fig 5.7. MO-WTS Gen Algorithm Initial Population

### **Fast Non-Dominated Sort**

The Fast Non-dominated Sort is used to divide the initial population into fronts based on the two Fitness functions. The non-dominated solutions in the whole population form the first front, the non-dominated solutions in the remaining population form the second front and this process is repeated until the whole population is divided into fronts.



```

.....Calculate both fitnesses.....
FOR(i=0 TO MAXTestSuite)
  ARRAY Fitness1[i] = MTotal[i] – Mcovered[i]
  j=1
  WHILE NOT End of TestCase
    ARRAY [i] – SUM(TestCasemethodRepetitions[j])
    j=j+1
  END WHILE
END FOR
i=0
k=1
.....Fast Non Dominant Sorting .....
WHILE ANY POPULATION NOT EMPTY
  WHILE NOT END OF POPULATION
  FOR(j=1 TO Population)
    IF Fitness1[i] AND Fitness2[i] > Fitness1[j] AND Fitness2[j]
      ARRAY DominantPool[] =TestSuite[i] AND REMOVE TestSuite[i] from Population
    ELSE IF Fitness1[i] AND Fitness2[i] < Fitness1[j] AND Fitness2[j]
      ARRAY Remove TestSuite[i] From Dominant Pool[] AND Population = TestSuite [i]
      Add TestSuite[j] to DominantPool[]
    ELSE IF Fitness1[i] AND Fitness2[i] ≠ Fitness1[j] AND Fitness2[j]
      Add TestSuite[i] AND TestSuite[j] to DominantPool[] AND Remove TestSuite[i] AND
      TestSuite[j] from Population

```

```

END IF
i=i+1
END WHILE
ARRAY Front[k] =DominantPool[]
k=k+1
END WHILE

```

Fig 5.8: MO-WTS Gen Algorithm Fast Non-Dominant Sort

### Calculate Crowding Distance

Crowding Distance is calculated for each solution after dividing the population into fronts. This value represents the distance of a solution from its neighbors in the search space. Solutions with higher values of the crowding distance are given priority during selection. First the crowding distance according to the first fitness function is calculated, then it is calculated based on the second fitness function and finally the cumulative crowding distance is calculated.

```

.....Calculate Crowding distance according to first fitness function.....
FOR (i=0 TO MAXFronts)
  Sort TestSuites by Fitness1
  FOR (j=0 TO MAXTestSuites)
    If (j=1 OR j=MAX)
      crowdingDistance1[j] = ∞
    ELSE
      crowdingDistance1[j] =ABSOLUTE(Fitness1(TestSuite[j-1]) - Fitness1(TestSuite[j+1]))
    END IF
  END IF
END IF

```

```

END FOR
.....Calculate Crowding distance according to second fitness function.....
Sort TestSuites by Fitness2
  FOR (j=0 TO MAXTestSuites)
    If (j=1 OR j=MAX)
      crowdingDistance2[j] = ∞
    ELSE
      crowdingDistance2[j] =ABSOLUE(Fitness2(TestSuite[j-1]) – Fitness2(TestSuite[j+1]))
    END IF
  END FOR
.....Calculate The Cumulative Crowding Distance.....
  FOR (j=0 TO MAXTestSuites)
    crowdingDistance[j] = crowdingDistance1[j] + crowdingDistance2[j]
  END FOR

```

Fig 5 9: MO-WTS Gen Algorithm Crowding Distance

**Selection**

The second step is to select the test cases based on their fitness to form the first parent population. The no of test Suites that are used for selection is taken as an input from the user and the population size can be changed in each run.

```

.....Select a pre-defined number of random test Suites.....
FOR(i=0 TO PredefinedRand)
  ARRAY RandomChoice[] = RAND(TestSuite[])
END FOR
.....Choose the fittest among the randomly chosen.....
FOR(i=0 TO PredefinedRand)
  ARRAY Selected[] = Selected[] + MAX(CrowdingComparison Operator(TestSuite[i]))
  REMOVE MAX(CrowdingComparison Operator(TestSuite[i]))
END FOR

```

Fig 5 10: MO-WTS Gen Algorithm Selection

**Crossover**

Crossover is performed on the selected Test Suites by swapping the test cases in the test suites using One point Crossover. After crossover the child population of the same size as the parent population is generated.

```

.....Crossover Selected Solutions.....
FOR (i=1 TO MAXPopulation; i=i+2)
  FOR (j=1 TO MAXTestCase)
    Child[i][j] = Selected[i][jto(j/2)]+Selected[i+1][(j/2)toMAX]
    Child[i+1][j] =Selected[i+1][jto (j/2)]+Selected[i][(j/2)to MAX]
  
```

```

END FOR
END FOR

```

Fig 5.11: MO-WTS Gen Algorithm Crossover

### **Mutation**

Mutation in the Genetic Algorithm is meant to maintain diversity in the search space exploration. In MO-WTS Gen the mutation is of three types. A random number is generated to decide what type mutation should be performed. The three possible types are i. insertion ii. Deletion and iii. Modification.

```

.....Perform Mutation on the child Population.....
MutationProb = RAND(1-3)
If MutationProb = 1
    MutationTestCase[] = TestCase[RAND]
    Remove MutationTestCase [RAND]
ELSE IF MutationProb = 2
    MutationTestCase[] = TestCase[RAND]
    Add MutationTestCase [RAND]
ELSE IF MutationProb = e
    MutationTestCase[] = TestCase[RAND]
    Modify MutationTestCase [RAND]
END IF

```

Fig 5.12. MO-WTS Gen Algorithm Mutation

### **Main Program Flow**

The Main flow of the program performs Fast Non-Dominated Sort, Crossover and Mutation on the selected Population in each iteration until the required Objectives are fulfilled i.e. the fitness function  $f(T)$  becomes 0 and the second fitness function  $g(T)$  which is responsible for keeping track of redundant coverage becomes zero too.

.....Iteratively Run the genetic Algorithm until optimized, minimized solution found.....

Generate Initial Population

FOR(i=0 TO Population)

    Fitness1 = fitness(TestSuite[i])

    Fitness2 = fitness(TestSuite[i])

Front[i] = FastNonDominantSort(Population)

END FOR

FOR(i=0 TO NO OF Fronts)

    WHILE NOT END OF FRONT

        crowdingDistance[i] = crowdingDistance(TestSuite[i])

    END WHILE

END FOR

Selected [] = TournamentSelection(Fronts[])

ChildPopulation[] = Crossover(Selected[])

MutatedPopulation = Mutation(ChildPopulation[])

If (Fitness1(Population) And Fitness2(Population) = RequiredFitness1 AND RequiredFitness2 )

    Terminate Program

    Return TestSuite[] with Required Fitnesses

END IF

Fig 5 13: MO-WTS Gen Algorithm Main Program Flow

## 5.6 Case study used for implemented approach

The Example used in the case study is a program for a 'Scientific calculator'. All the functions of the program are divided into classes. There are 40 classes which are randomly called to form the test cases. The test cases are randomly combined to form test Suites of varying sizes. The partial diagram of the Example used in the case study as a System under test (SUT) is given below.

The example was chosen because it has enough loops and conditions so that it can provide enough challenge to test. The functions in the example are called by the program during testing to exercise code and the relevant branch. Evosuite uses a strong example to validate its cause but now the example will be used to run both the Single objective Whole test Suite Generation and Multi-Objective Whole test Suite Generation.

The intention is of testing this example through both the approaches. It is expected that our hypothesis will be proved and we will succeed in making improvements in the technique by reducing the number of run and the repetitions in the test Suite.

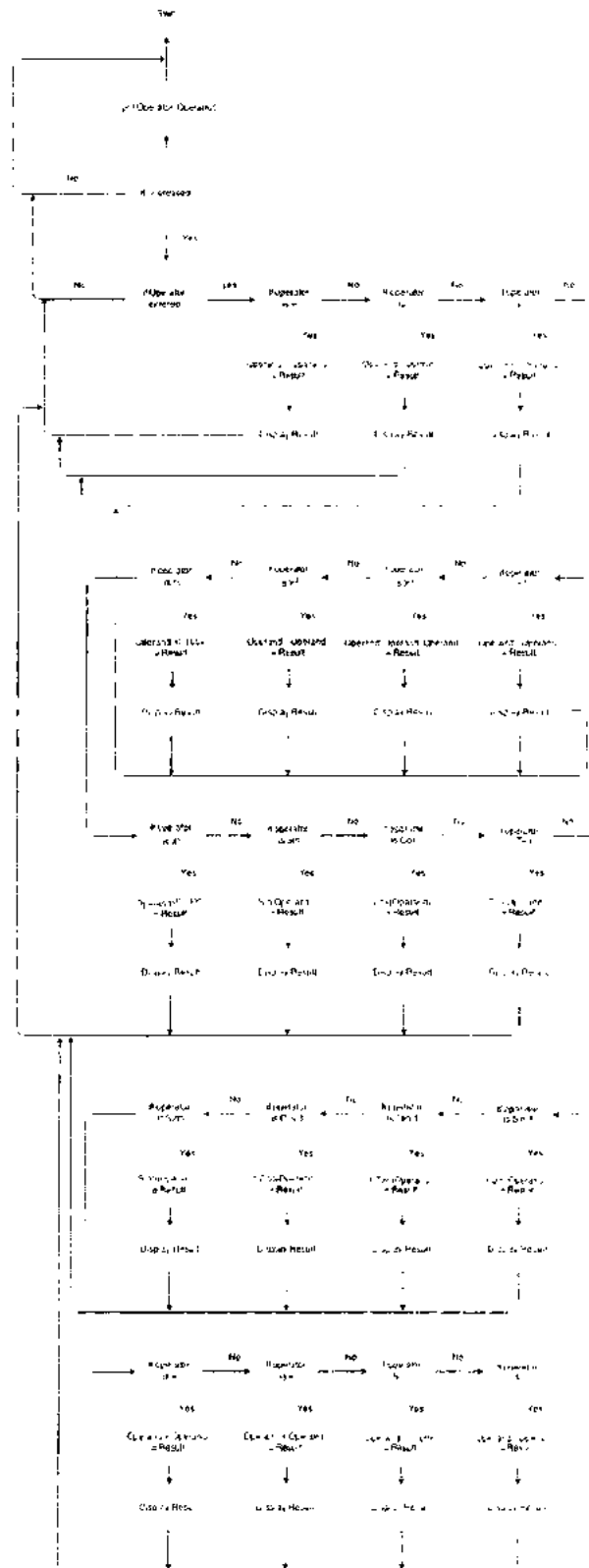


Fig 5.14: Contraflow Diagram of the Example



### 5.7 Flow Chart of proposed approach

The flow Chart of the proposed technique is given below. The initial population is generated randomly, it is divided into fronts and the crowding distance is calculated of the Test Suites in each front. Selection, crossover and Mutation is performed on the selected data and to maintain elitism the parent and child population after each iteration is combined. This process is repeated until a Test Suite which fulfills both the objectives is found.

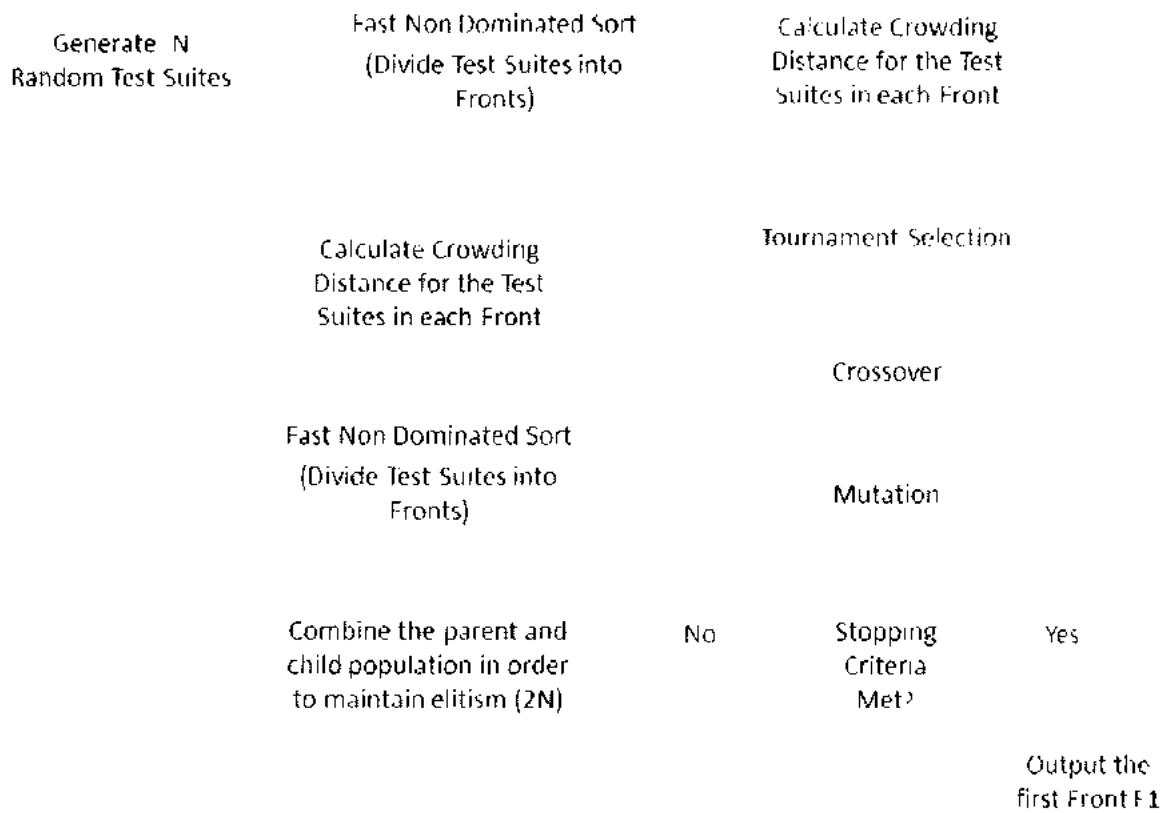


Fig 5.15: Contraflow Diagram of the Multi Objective Whole test Suite Generation Approach

# **Chapter 6**

## **TOOL**

# **IMPLEMENTATION**

## TOOL IMPLEMENTATION:

The tool was implemented successfully and favorable results were found. This section of the write-up is devoted to the tool's features and working.

Section 6.1 presents the architecture of the tool, the implementation details of the tool are explained in section 6.2 and the user interface is illustrated in section 6.3.

### 6.1 Research Methodology

The diagram illustrating the abstract architecture of the implemented approach is given below.

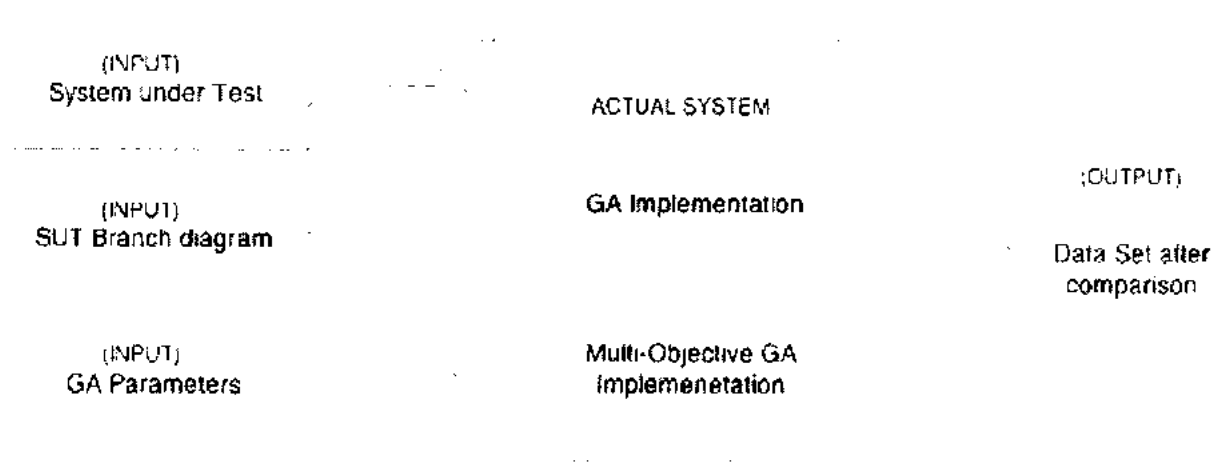


Fig 6.1. Architecture of the implemented approach

#### 6.1.1 Multi-Objective Genetic Algorithm Program

Multi-Objective Genetic Algorithm is actually an extension of an existing work. The existing work has been extended to fulfill multiple objectives simultaneously instead of working on each objective separately. Both the approaches use the same test data pool and apply two different kinds of GA on them to optimize them.

### 6.1.2 Branch coverage and method coverage

Our programs basic target is to check for the coverage of the methods but it checks the branch coverage of the methods too. This provides an in depth and thorough coverage check. This coverage criterion is what one of the main fitness functions of the system is based on.

### 6.1.3 Actual System

The actual system is what processes the input and provides results. The two main things in the actual system are the Single Objective Genetic Algorithm and the Multi-Objective Genetic Algorithm.

The two genetic Algorithms are implemented based on the fitness criteria we aim to achieve. The first part of the main system has one fitness function. This fitness function contains two parts. The first part is the number of methods that are as yet uncovered and the second part calculates the number of branches within a method that are uncovered. After combining these two calculations the complete fitness function that calculates the overall coverage achieved by the test Suite is formed.

The second part of the actual system is the Multi-Objective Genetic Algorithm. This contains two Fitness functions. The first fitness function remains the same, since we still need to take into account the amount of coverage achieved but the second fitness function calculates the redundancy while achieving the required coverage. And it attempts to minimize this redundancy. This fitness function calculated the repeated number of methods within a test Case. And after calculating the sum of the repetition in each test case calculates the cumulative Repetition or redundancy in the test suite.

### 6.1.4 Test Data Set

The test Data set contains the results achieved by the system after the comparisons are made between the approaches. These results are achieved after a thorough analysis of the results given by the two approaches is done.

## 6.2 Components of implemented Genetic Algorithm

The implemented Genetic Algorithm has the following components.

- Generation of the initial population
- Calculation of the two fitness functions
- Fast Non-Dominant Sort [88, 89]
- Calculation of the crowding value
- Performing the crowding comparison operation
- Selection of the Test Data
- Crossing over of the Test Data
- Mutation of the test Data
- Elitist recombination of the test data.

## 6.3 System Components

The following Section presents the system components of the Implemented tool.

### 6.3.1 Whole test Suite Generation

The System Components for the whole test Suite generation is given below.

## Initial menu

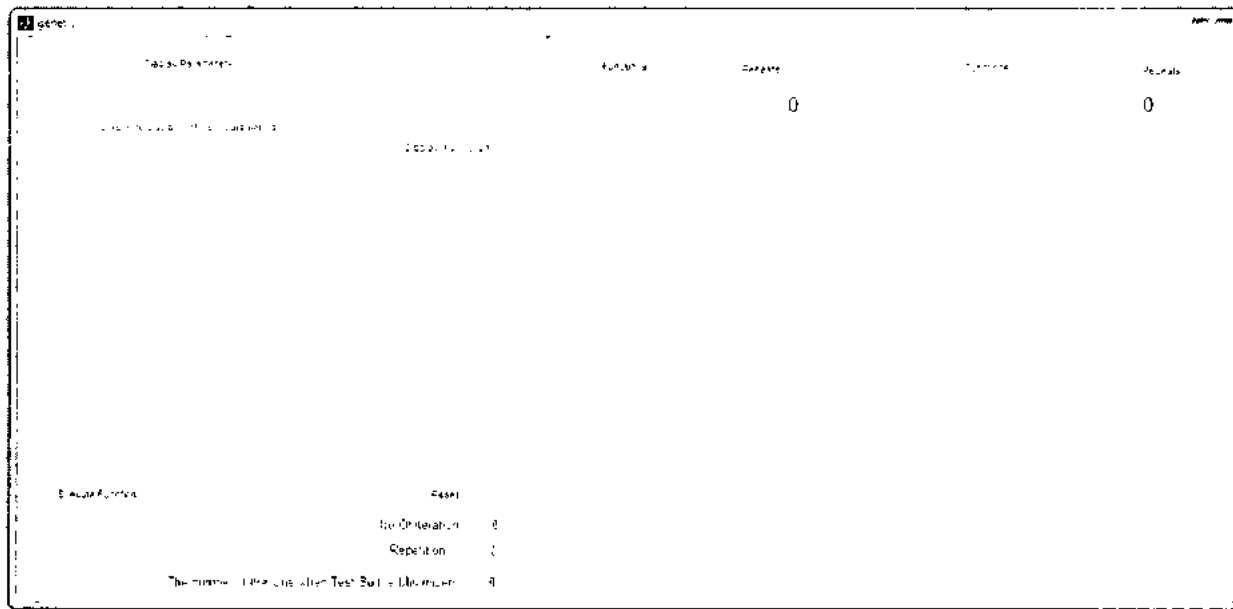


Fig 6.2: Main page of the tool

A zoomed view of the left side of the main interface.

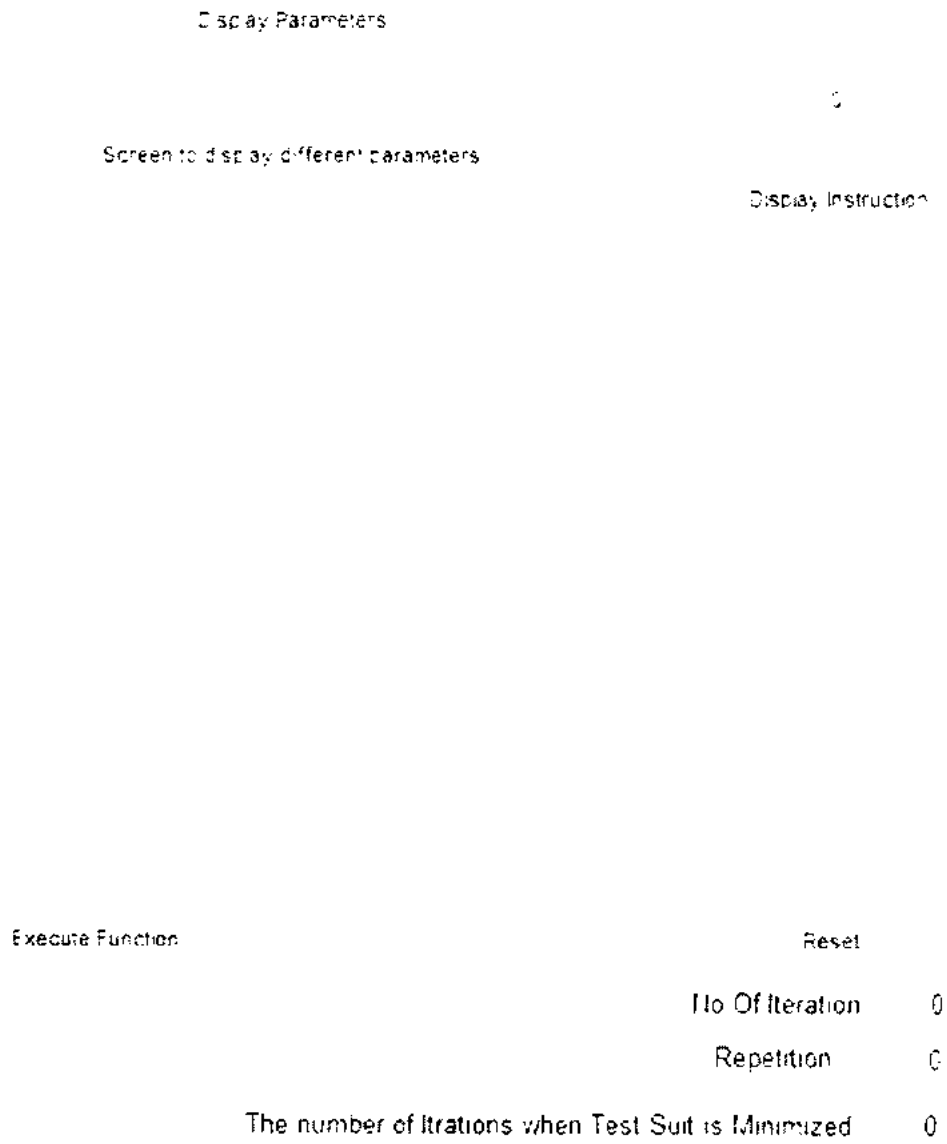


Fig 6.3: Zoomed View of the Tool

Initialize the process by pressing the Execute Function button. After the process is initialized the Create test Cases Button is activated and made clickable.

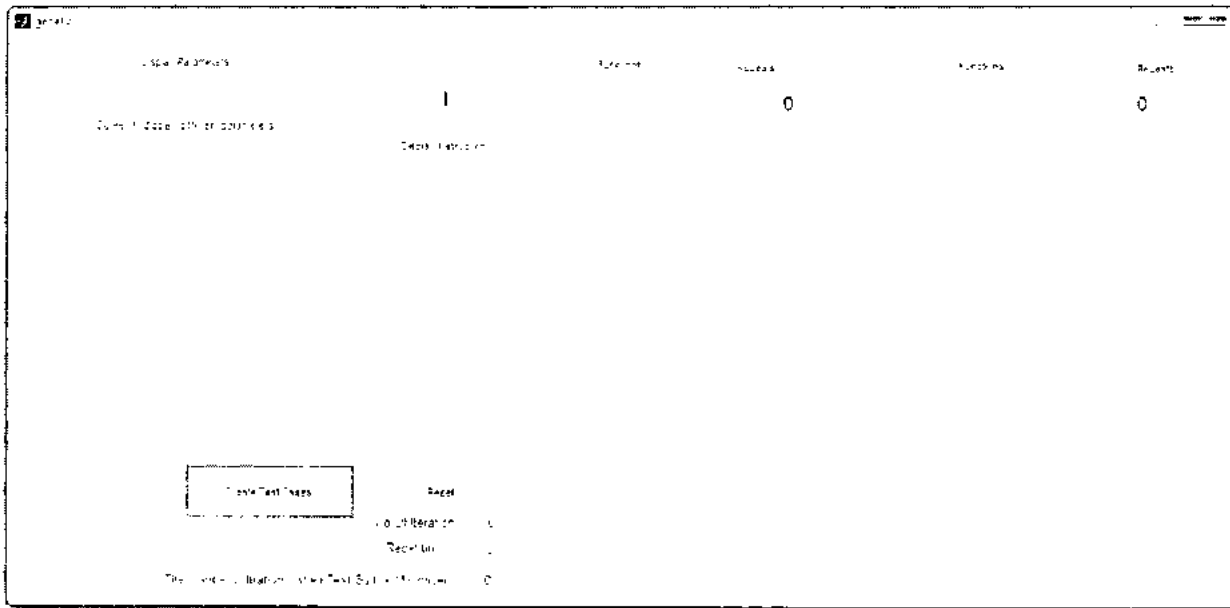


Fig 6.1: Initialize Process

### Initial Population Generation

Randomly generate the test cases by clicking the Create test Cases Button. The test cases are randomly generated the selected random method calls etc. to form test cases, which are combined to form Test Suite.

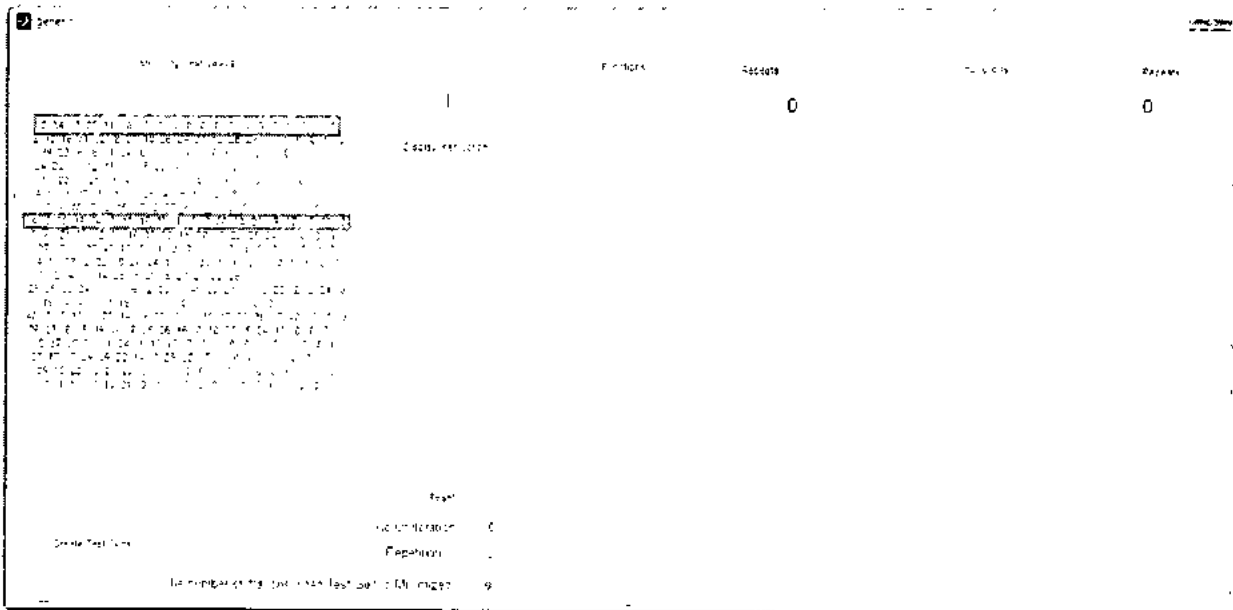




Fig 6.5. Create test Cases

After clicking this button the initial population of random test Cases is generated. Each line represents 1 test cases and the numbered area is the part that contains methods while the part with zeros is the empty part of the matrix. The first test case has 7 methods while the 8<sup>th</sup> test case has 20 methods. This is because the test cases are randomly generated and their size is kept variable. After the test cases are generated the create test Cases button is disabled and the Create test Suite button is activated.

Randomly generate the test Suites by clicking the Create test Suites Button.

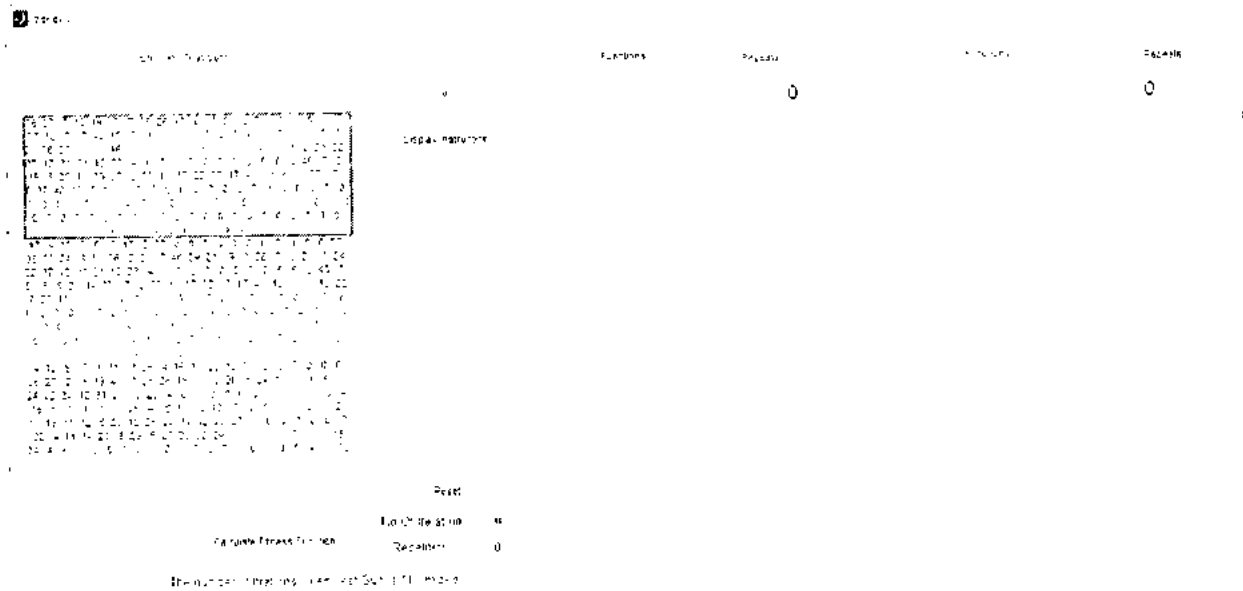


Fig 6.6. Create test Suites

This will generate a random number of test Suites from the test cases created in the previous step. After this the Create test Suites button is disabled and the Calculate fitness function button is activated. The generated test suites and the test cases contained in them are showed in the panel on the left.

## Fitness Function Calculation

Calculate the Fitness of each test suite that was generated by clicking the Calculate Fitness function button. When this button is pressed, the fitness for all the test Suites in the system is calculated. The calculated fitnesses are displayed in the panel on the left side. After this the Calculate Fitness function button is disabled and the Best Select button is activated. After this step all the horizontal buttons are disabled and the core GA operation are the only thing left. The buttons for these are listed vertically.

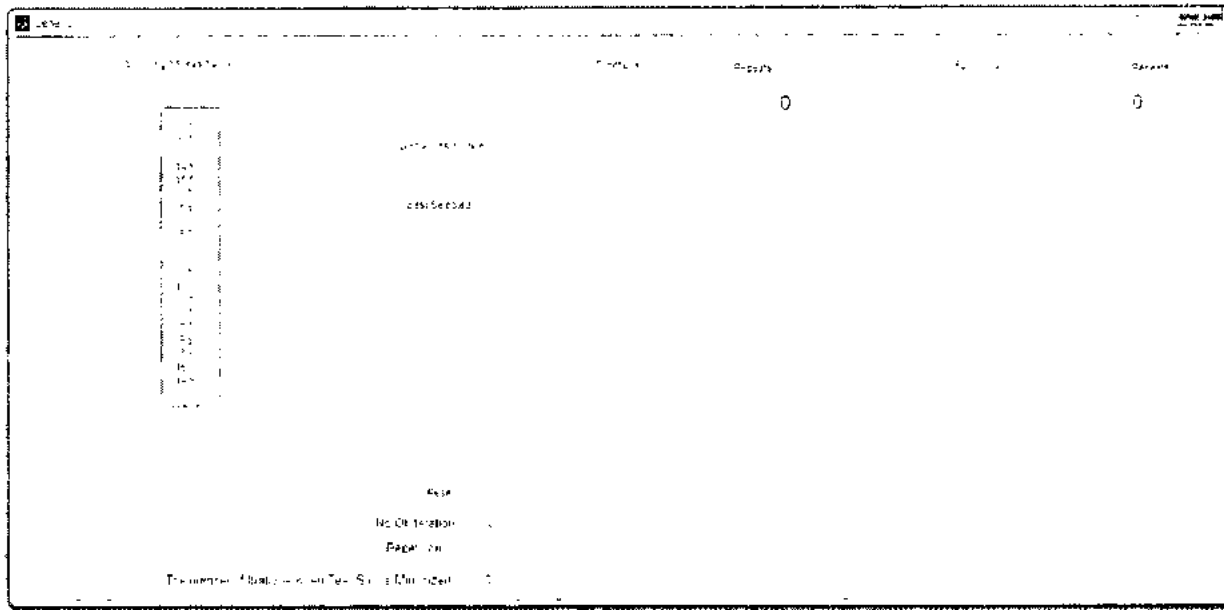


Fig 6 7 Calculate the Fitnesses

**Selection**

Perform Selection by clicking the Best Selected button. On clicking this button the test Suites with the best fitness are selected and the button for selection is disabled.

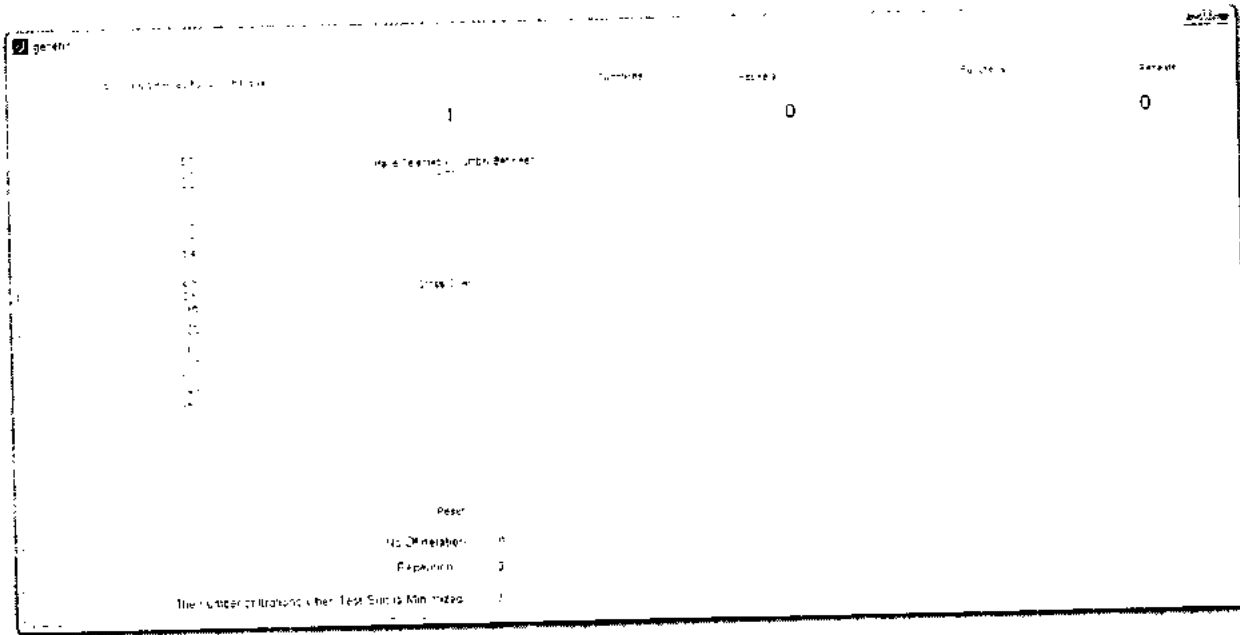


Fig 6.8. Perform Selection

The fitness values of the selected test Suites is displayed in the panel on the left.

## Crossover

Perform crossover by clicking the Crossover button. On clicking this button the test Suites with the best fitness are crossed over and the button for crossover is disabled.

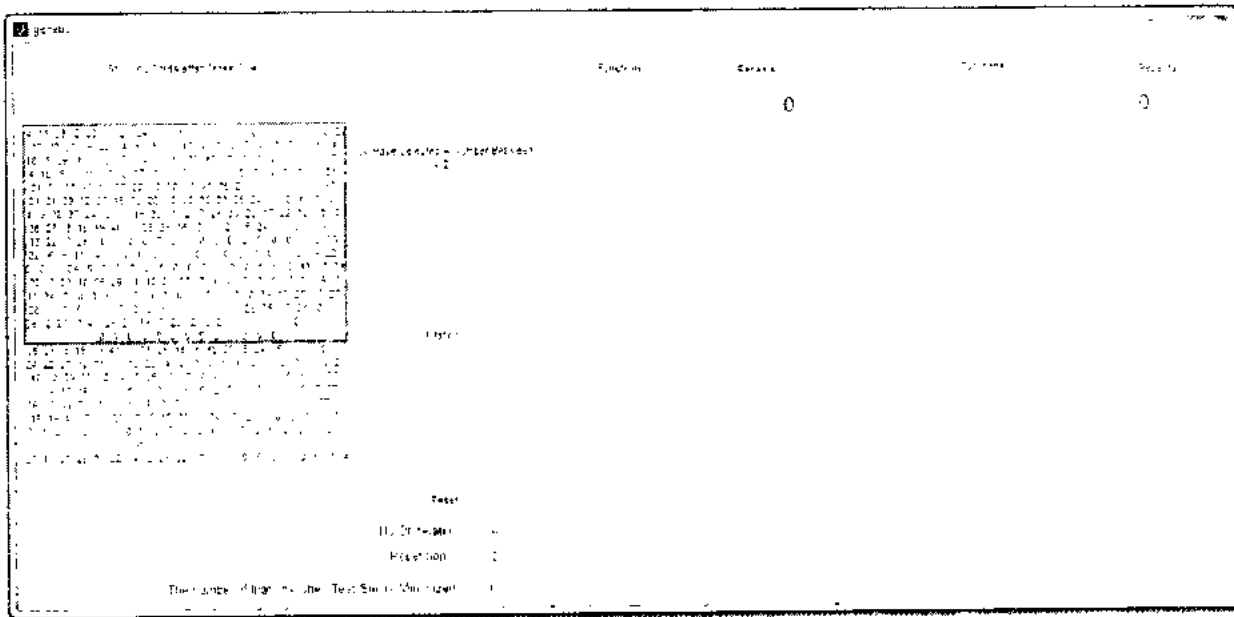


Fig 6.9: Perform Crossover

Clicking this button will generate the child population after the crossover and the resulting test Suites are displayed in the panel on the left side.

## Mutation

Perform mutations by clicking the Crossover button. On clicking this button the child population is mutated. Mutation is performed in three different ways

- A random test case is added to the Test Suite.
- A random test Case is removed from the Test Suite
- A Random test case is modified in the Test Suite

The test Suite after Mutation is displayed in the panel on the left. The mutation button is disabled and the child fitness function is enabled.

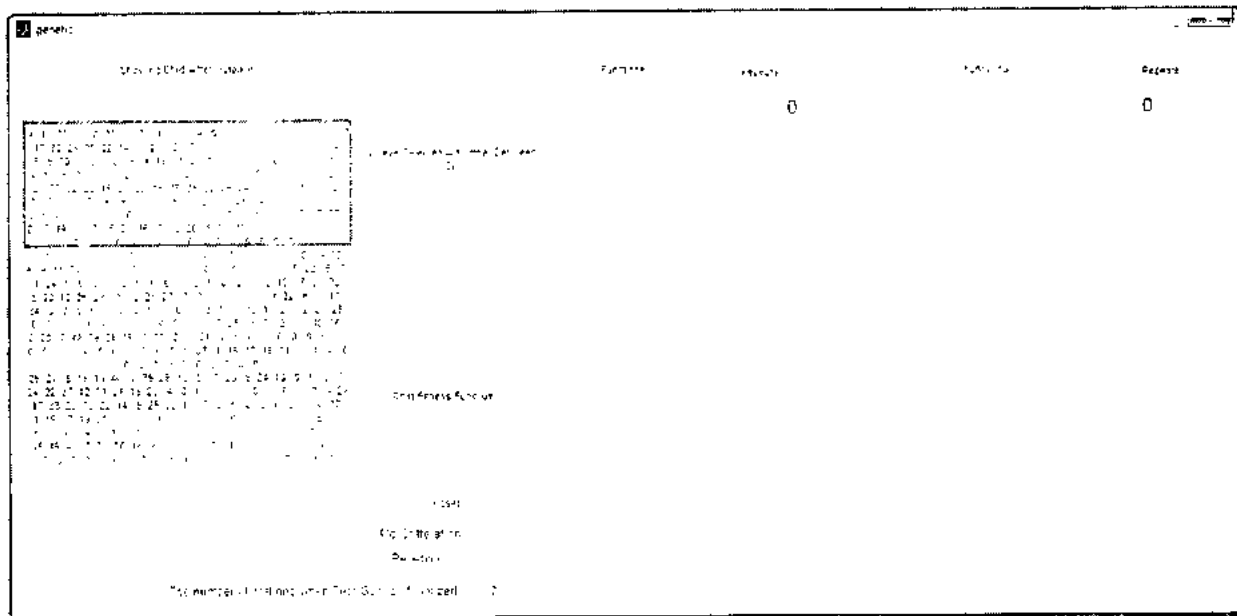


Fig 6.10: Perform Mutation

*Calculate the fitness of the child population.* The child population is a new set of test Suites whose fitness value is as yet unknown. This step calculates the fitness value for all the test Suites in the child population. The fitness values for the new population is displayed in the panel on the left. After this the child fitness button is disabled.

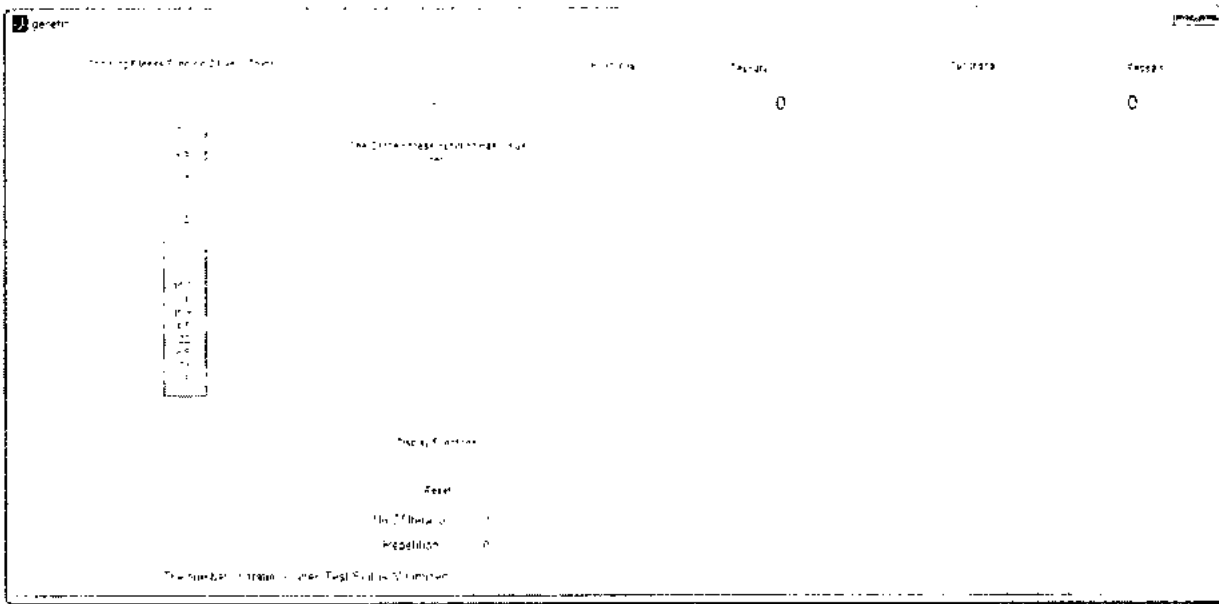


Fig 6.11: Fitness of the Child Population

After this button is clicked the program runs the subsequent iterations of the GA automatically without pressing the buttons, this is repeated until a test Suite with the coverage value '0' is found.

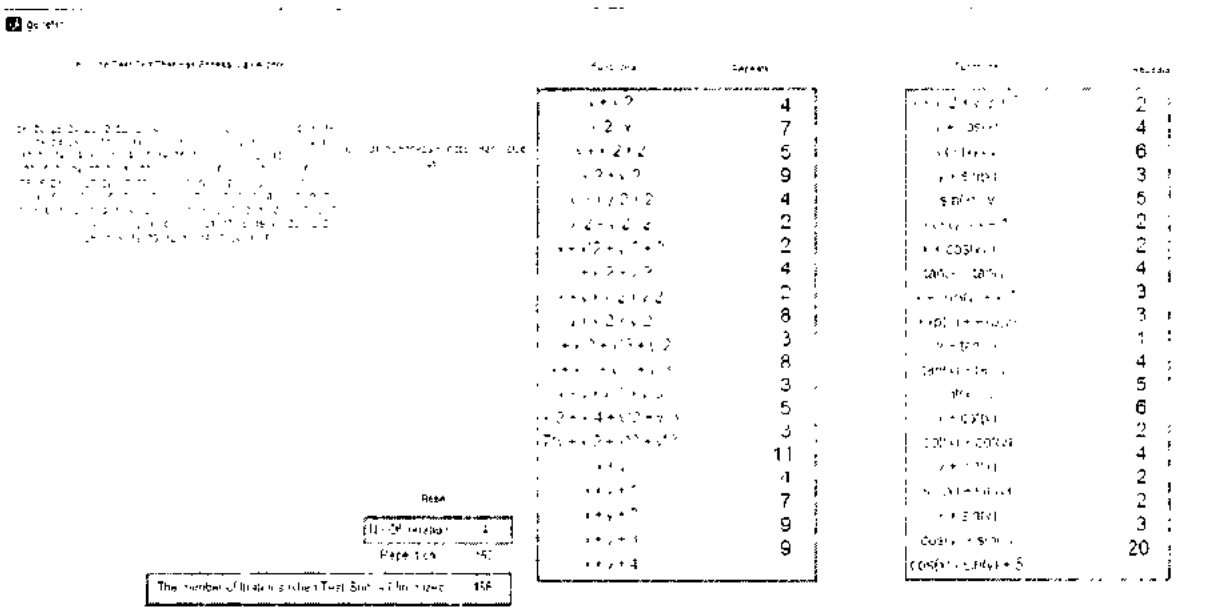


Fig 6.12: Display the Results

Once the test Suite with the fitness value '0' is found. The program displays the iterations it took to generate the result and the amount of redundancy in the resulting test suite. If the display functions button is pressed the test suites functions are displayed in the panel on the right side. With the Reset button the whole process can be repeated without having to rerun the program.

### 6.3.2 Multi-Objective Whole test Suite Generation

The interface and its working for the Multi-Objective Whole Test Suite Generation is given below.

#### Main Menu

Start the program and open the initial form. The initial interface is divided into three parts. The leftmost panel gives information about the data currently being processed. The horizontal buttons are related to test data generation activities and the vertical set of buttons are used to perform the actual process. The Panels on the right display the test Suite that is achieved as an output after the process is done.

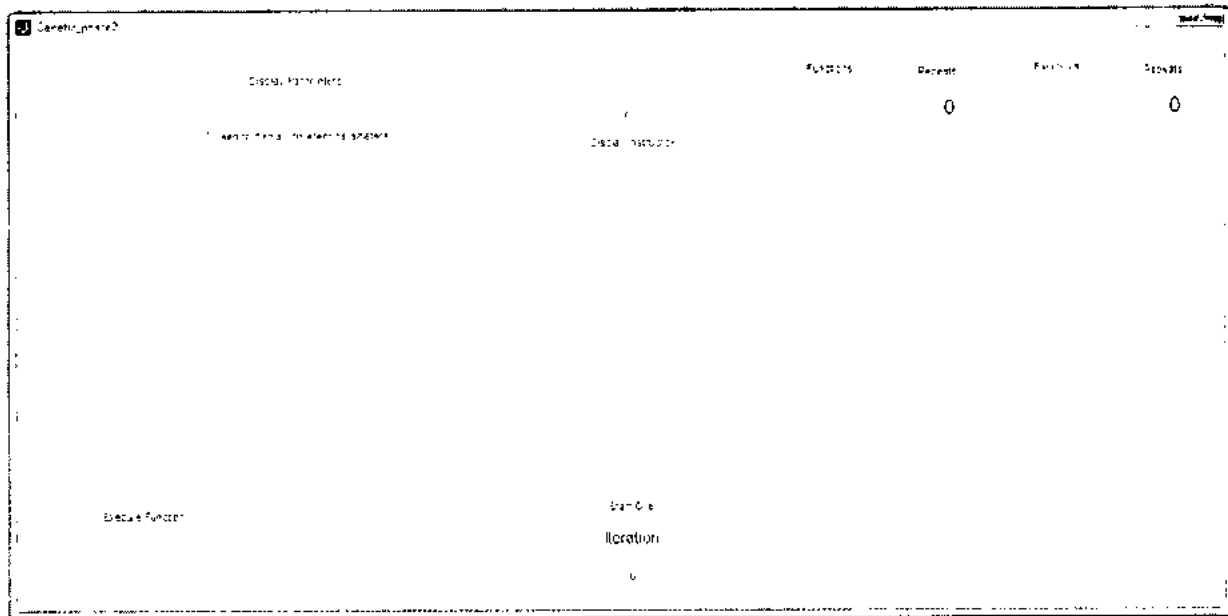


Fig 6.13. Main Interface

The zoomed view of the interface is given below.

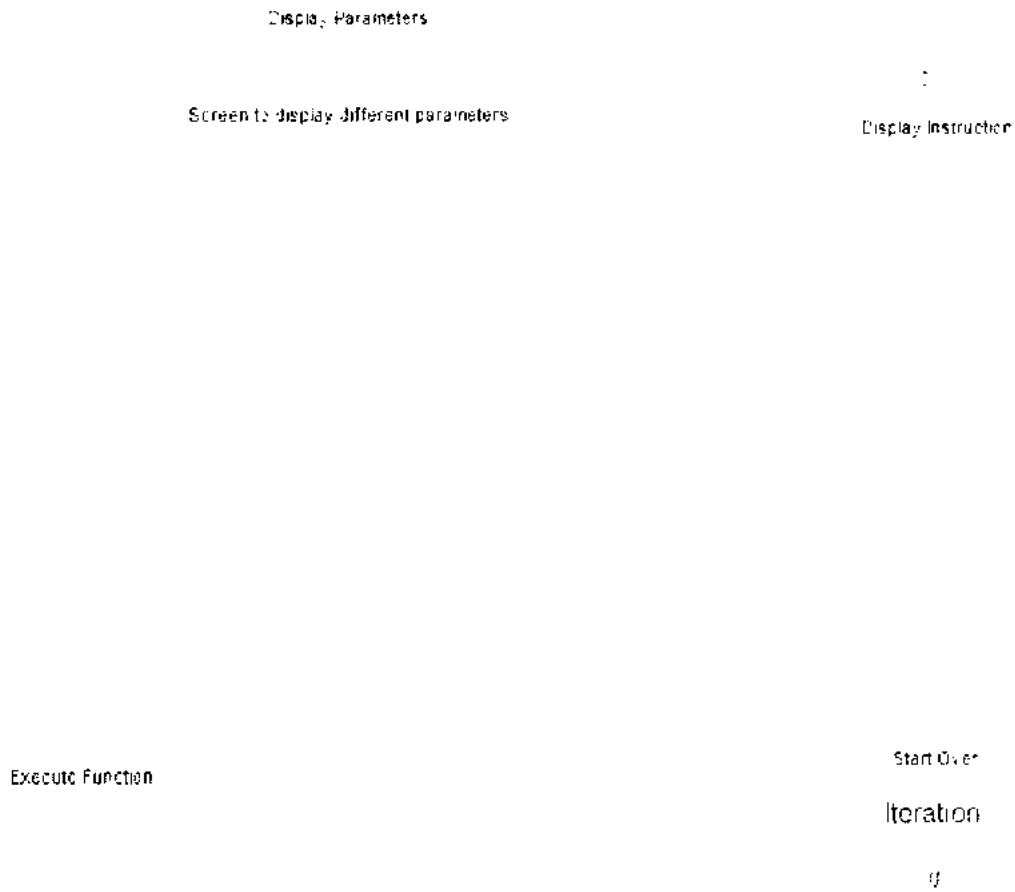


Fig 6.14: Zoomed View of the interface



### Initialize Initial Population

Initialize the process by pressing the Execute Function button. After the process is initialized the Create test Cases Button is activated and made clickable.

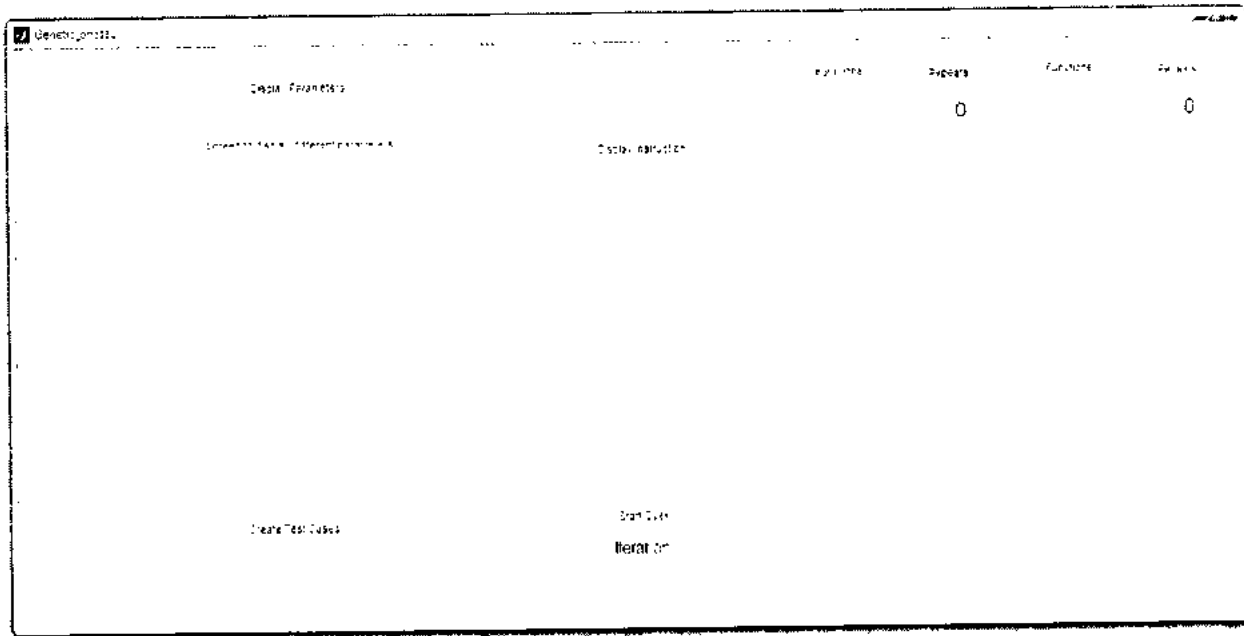


Fig 6 15: Initialize the process

After clicking the 'create test cases' button, the initial population of random test Cases is generated. Each line represents 1 test cases and the numbered part is that which contains methods while the part with zeroes is the empty part of the matrix.

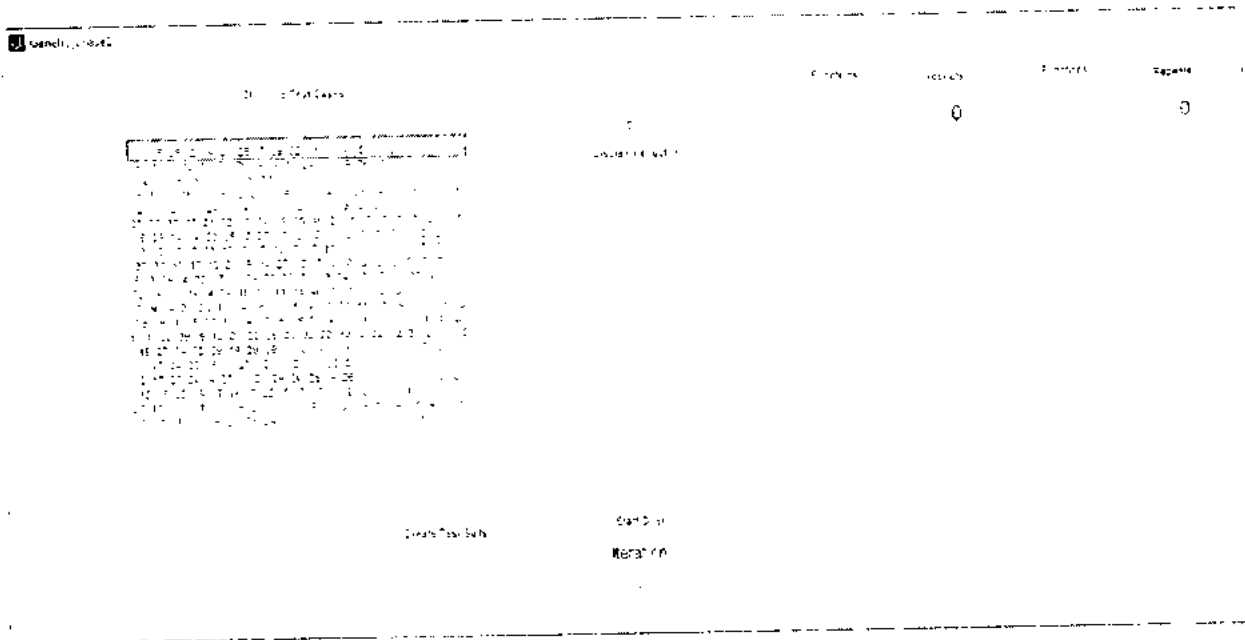


Fig 6.16: Create test Cases

Randomly generate the test Suites by clicking the Create test Suites Button.

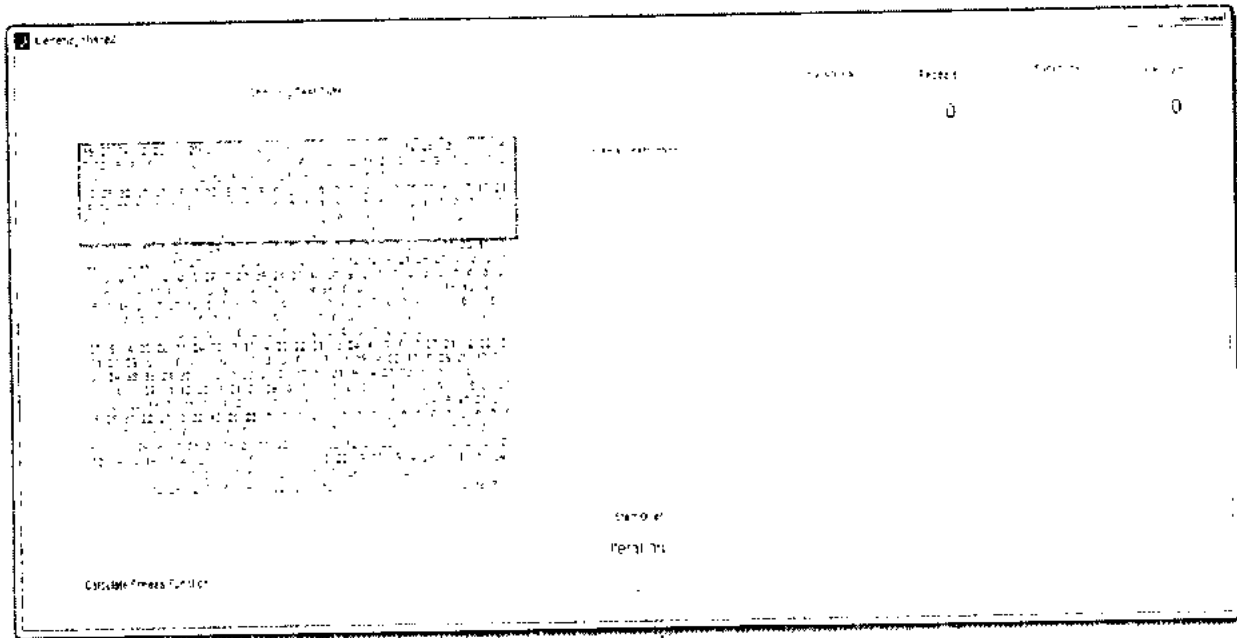


Fig 6.17: Create test Suites

This will generate a random number of test Suites from the test Cases created in the previous step. After this the Create test Suites button is disabled and the Calculate fitness function

is activated. The generated test suites and the test cases contained in them are showed in the panel on the left.

### Fitness Calculation

Calculate the Fitness of each test suite that was generated by clicking the Calculate Fitness function button. When this button is pressed, the first fitness function for the entire population is calculated. The calculated fitnesses are displayed in the panel on the left side. After this the 'Calculate Fitness function' button is disabled and the 'Fitness Function2' button is activated.

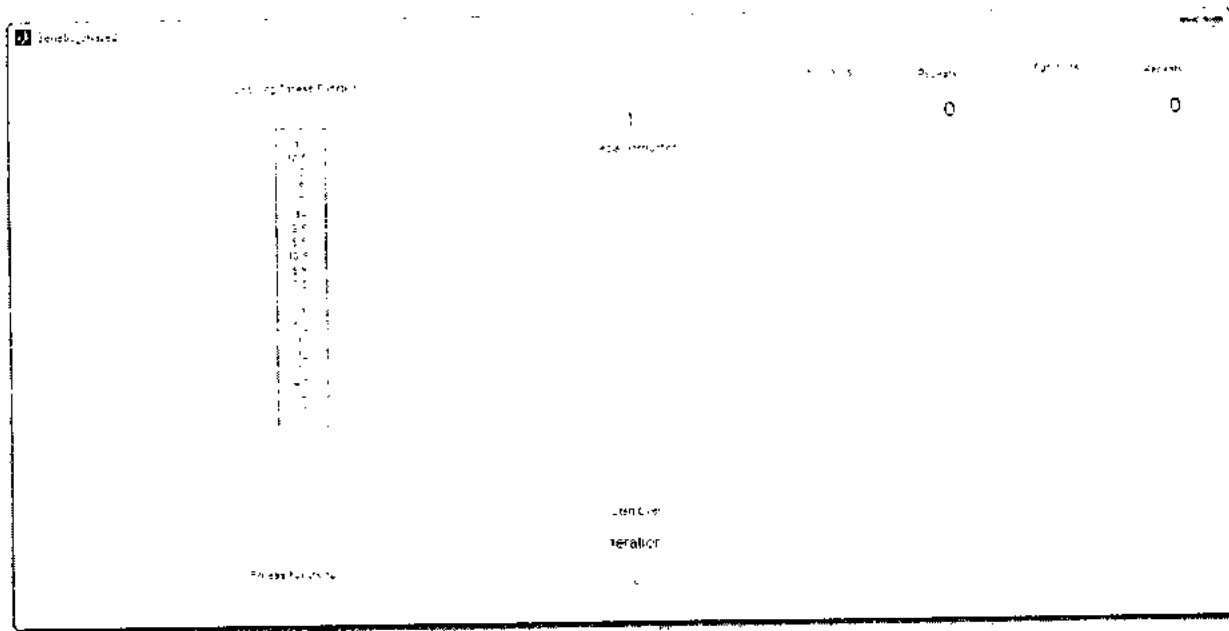


Fig 6.18: Calculate the Fitnesses

Calculate the Second Fitness function for all the test Suites by clicking the 'Fitness Function2' button. When this button is pressed, the second fitness function for the entire population is calculated. The calculated fitnesses are displayed in the panel on the left side. After this the 'Fitness function2' button is disabled and the 'Crowding Distance' button is activated.

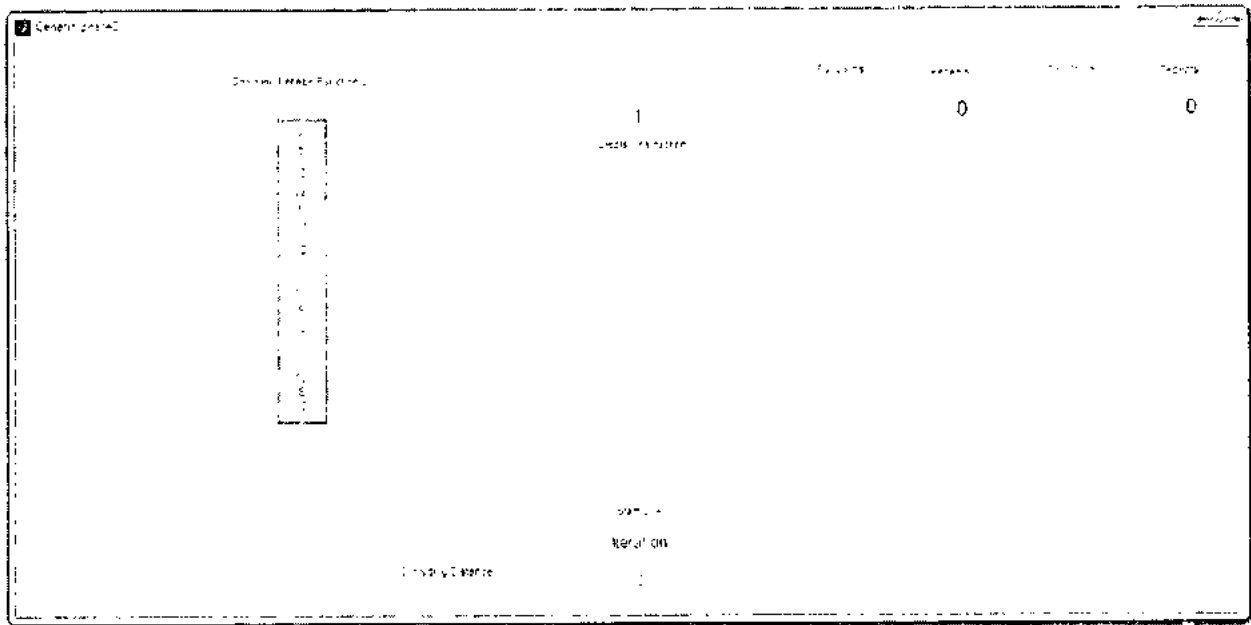


Fig 6.19: Calculate the Fitness Function2

### Crowding Distance

Calculate the crowding distance of the population by clicking the 'Crowding Distance' button. When this button is pressed, two things are done. The first is that the entire population is divided into fronts by performing the Fast Non-Dominant Sort on it and then the crowding distance for each individual in all the fronts is calculated. The calculated crowding distances are displayed in the panel on the left. This concludes the part of the program which is meant to generate the test data.

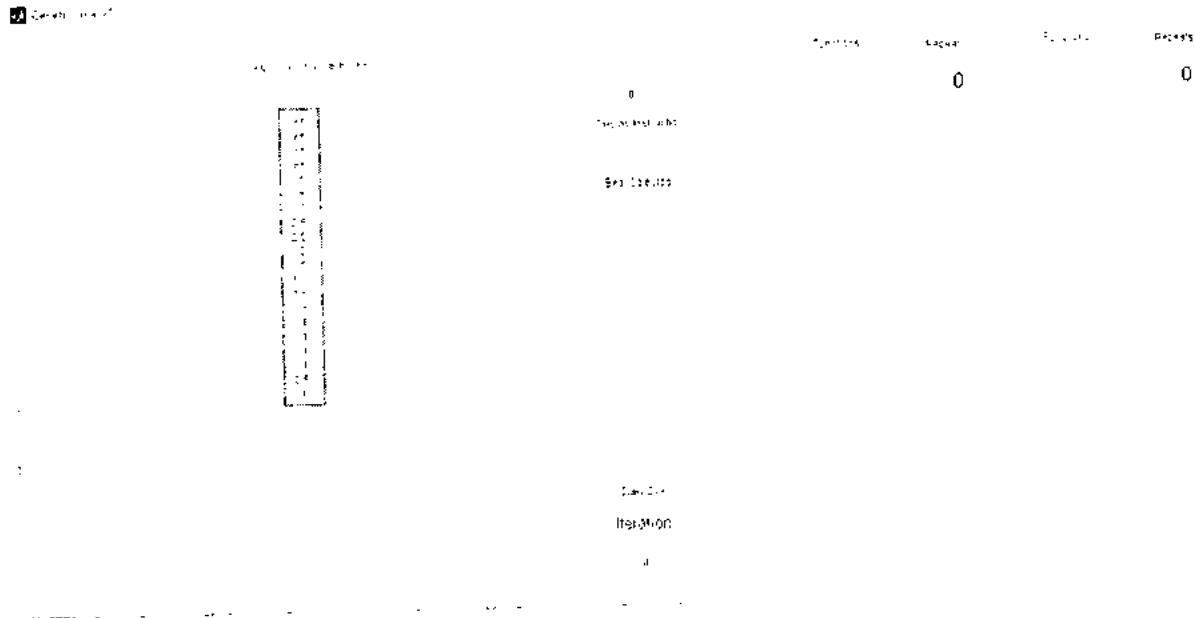


Fig 6.20: Calculate the crowding distance

### Selection

Perform Selection by clicking the Best Selected button. On clicking this button the test Suites with the best fitness are selected and the button for selection is disabled.

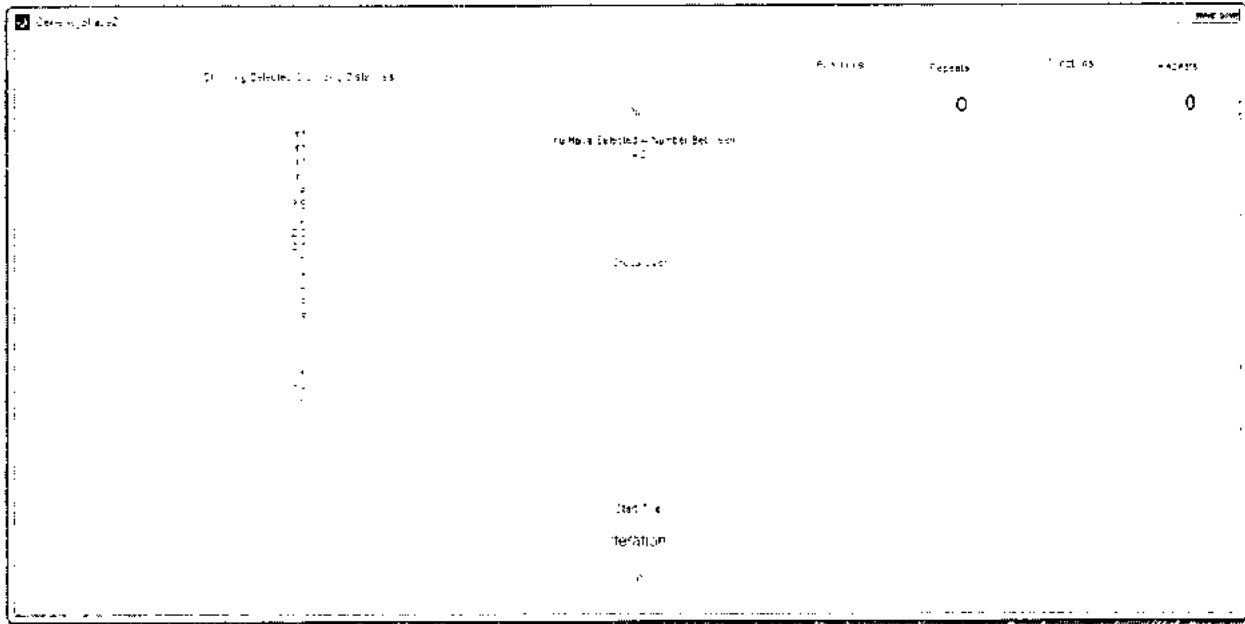


Fig 6.21: Perform Selection.

## Crossover

Perform crossover by clicking the Crossover button. On clicking this button the test Suites with the best fitness are crossed over and the button for crossover is disabled.

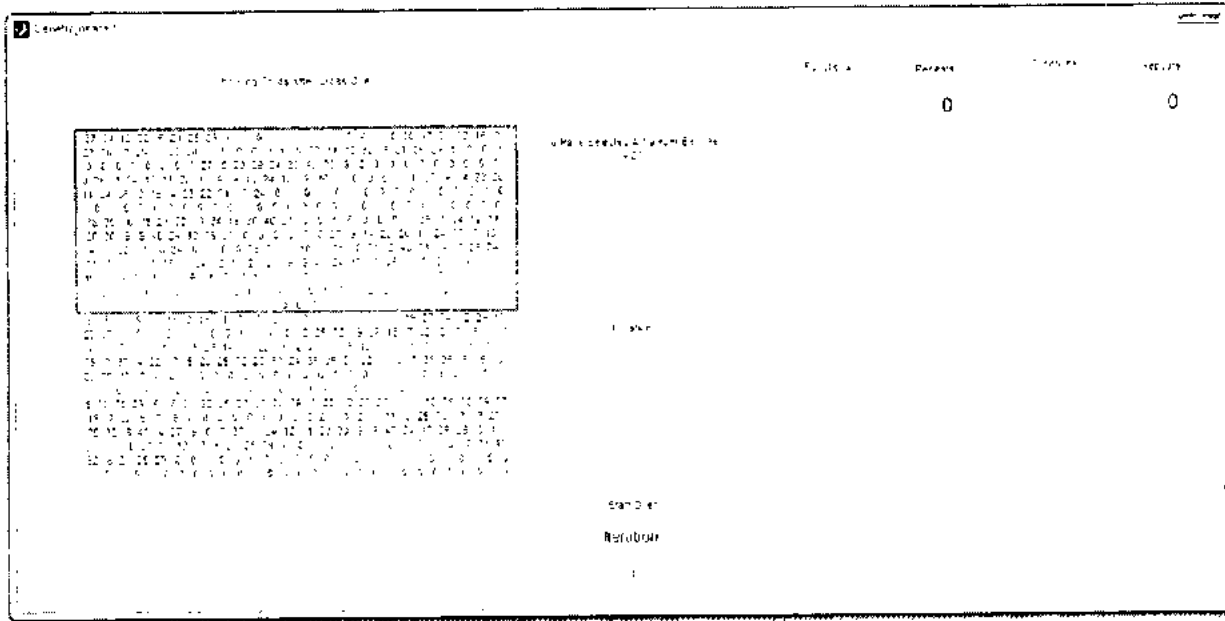


Fig 6.22. Perform Crossover

Clicking this button will generate the child population after the crossover and the resulting test Suites are displayed in the panel in the left side.

## Mutation

Perform mutations by clicking the Crossover button. On clicking this button the child population is mutated. Mutation is performed in three different ways

- A random test case is added to the Test Suite.
- A random test Case is removed from the Test Suite
- A Random test case is modified in the Test Suite

The test Suite after Mutation is displayed in the panel on the left. The mutation button is disabled and the child fitness function is enabled.

2 Generations

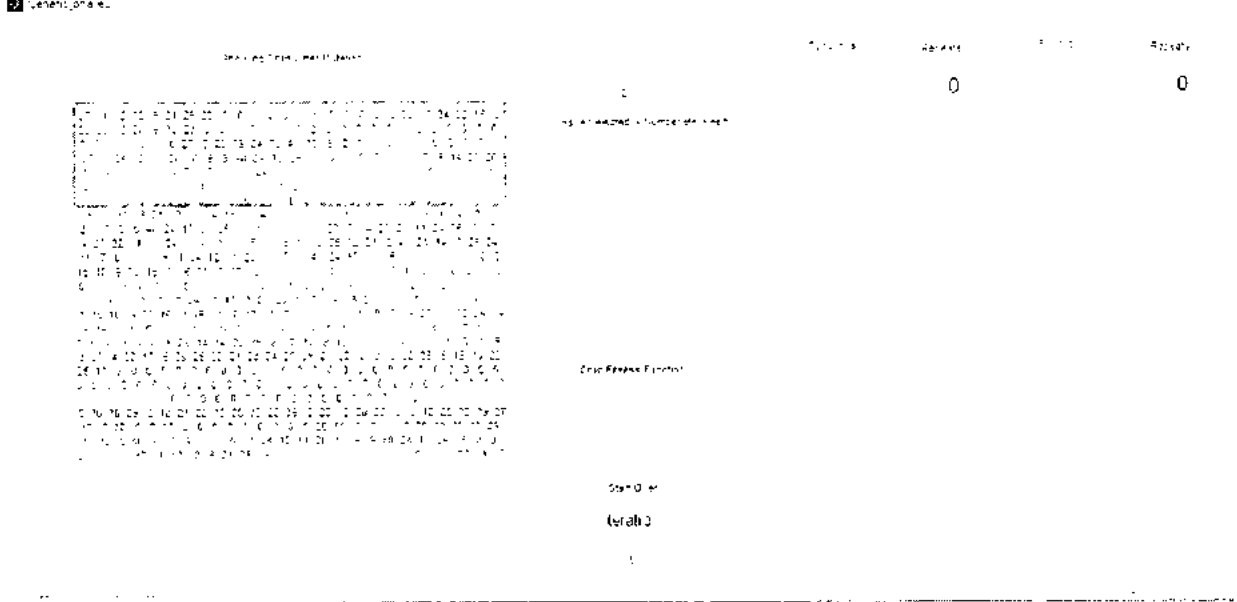


Fig 6.23: Perform Mutation



## General Program Flow

Calculate the fitness of the child population. The child population is a new set of test Suites whose fitness value is as yet unknown. This step calculates the fitness value for all the test Suites in the child population. The fitness values for the new population is displayed in the panel on the left. After this the child fitness button is disabled.

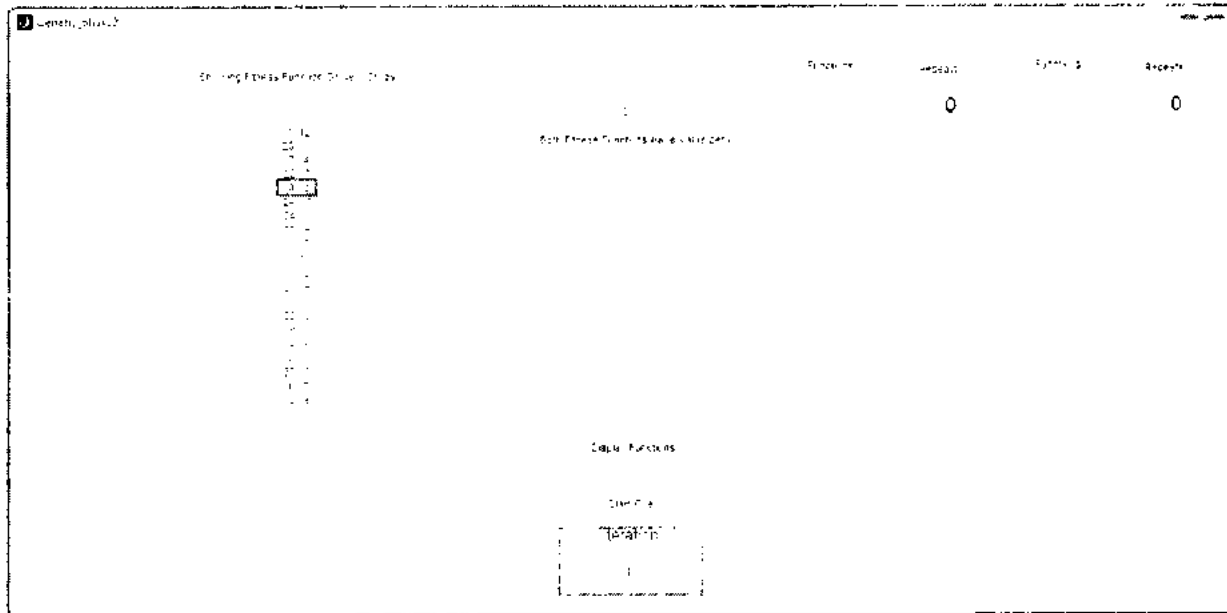


Fig 6.24: Performing Iterations

When this button is pressed the program performs the subsequent iterations of GA automatically without any intervention from the user. The iterations are repeated until a test Suite with both its fitness functions as zero is found. When this is done the number of iterations required to reach this result are shown. The two fitness functions of the entire test Suite are displayed in the panel on the left side.

Display the results by clicking the 'display Functions' button. The test Suite which has the fitness value for both its Fitness functions as zero is displayed in the panel on the right side, when the 'display functions' button is clicked.

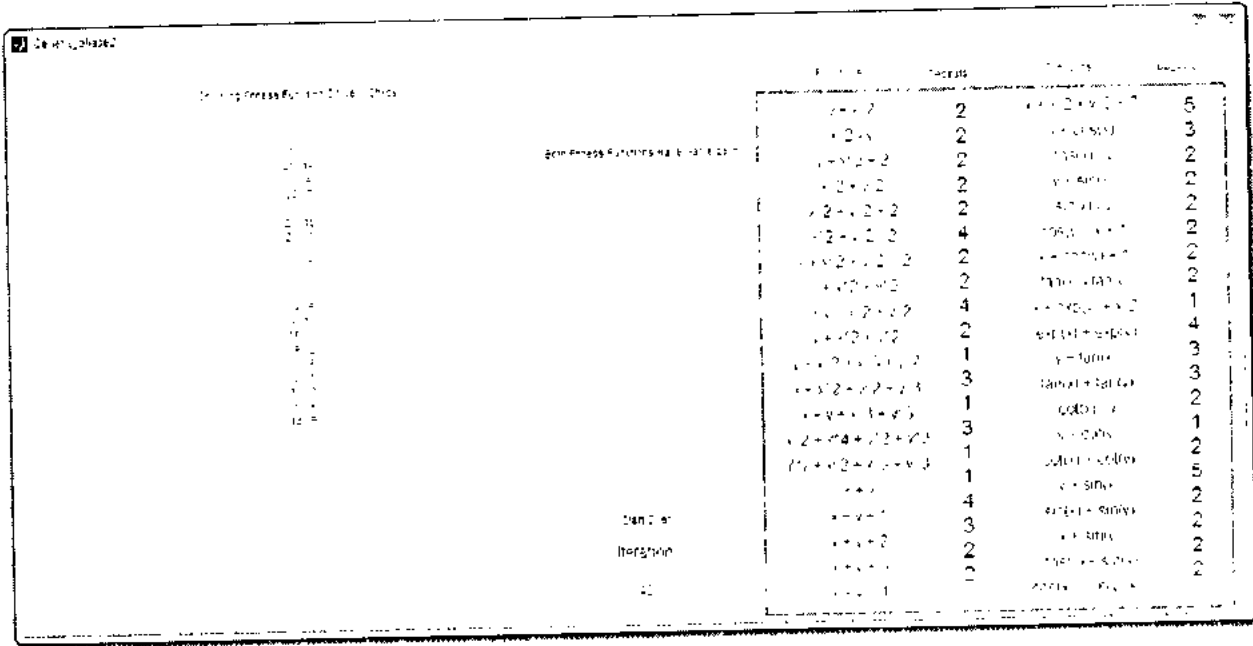
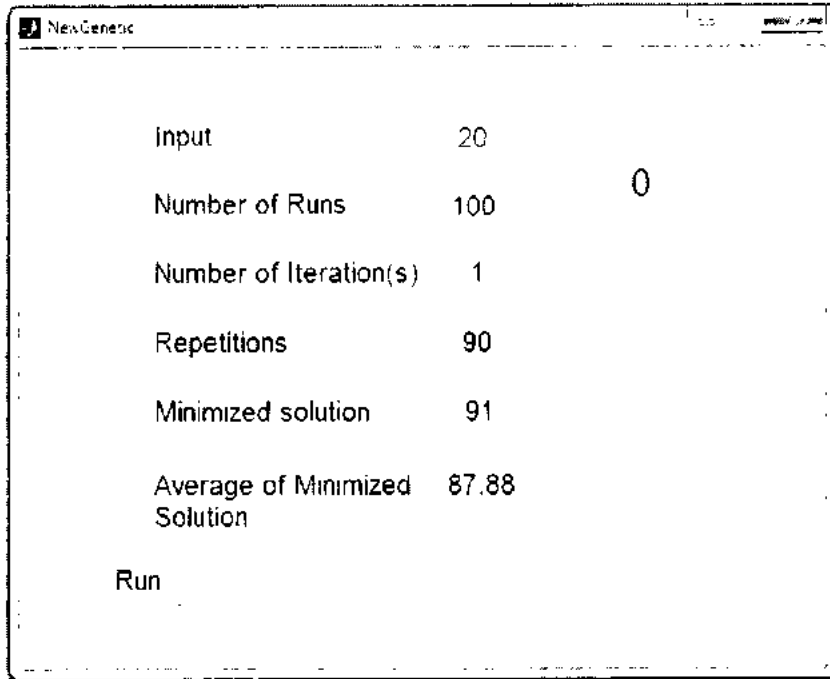


Fig 6.25: Display the Results.

Clicking the 'Start Over' button can allow us to repeat the process without having to restart the program.

### 6.3.3 Calculate Average number of runs.

To calculate the average number of runs the following part of the program is used. The panel shown below calculates the average of the results after running the 'Whole test Suite Generation' technique for a given number of times



The screenshot shows a window titled 'NewGenetic' with a list of parameters and their values. The parameters are: Input (20), Number of Runs (100), Number of Iteration(s) (1), Repetitions (90), Minimized solution (91), and Average of Minimized Solution (87.88). There is a 'Run' button at the bottom left. A large '0' is displayed to the right of the 'Number of Runs' value.

Input	20	
Number of Runs	100	0
Number of Iteration(s)	1	
Repetitions	90	
Minimized solution	91	
Average of Minimized Solution	87.88	

Run

Fig 6.26: Calculate Average of the existing approach's results

The average calculation for the 'Multi-Objective Whole test Suite Generation' technique is given below.

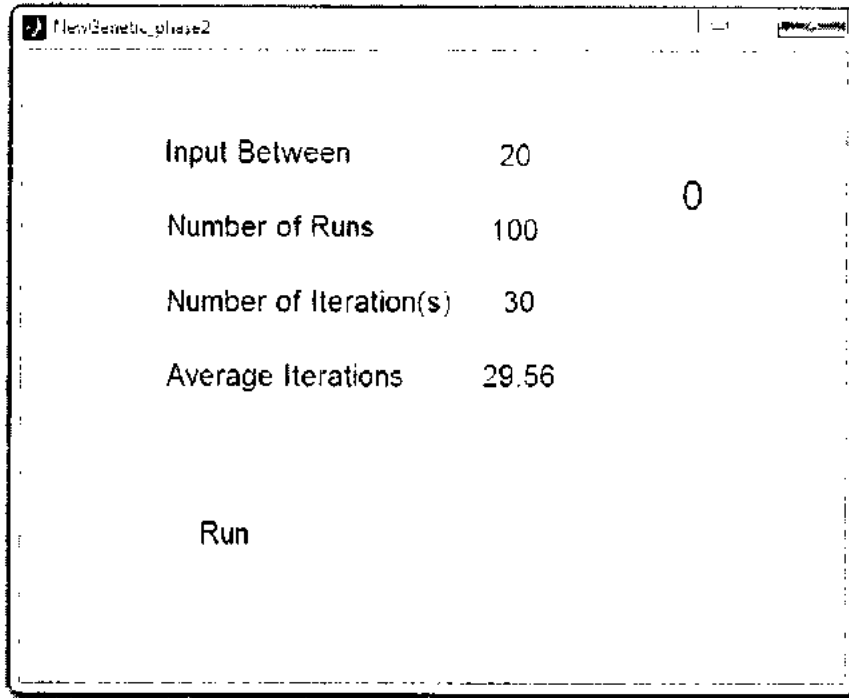


Fig 6.27: Calculate Average of the implemented approach's results.



# **Chapter 7**

## **EXPERIMENTAL DESIGN, RESULTS AND DISCUSSIONS**

## EXPERIMENTAL DESIGN, RESULTS AND DISCUSSIONS:

### 7.1 Experimental Design:

This chapter includes the evaluation of results and discussion, the experimental design that is set for the implementation is as follows:

#### 7.1.1 Dataset

Following are needed for the experiment:

- Code the 'Whole Test Suite Generation' technique.
- Code the 'Multi-Objective Whole Test Suite Generation' technique.
- System to be tested.
- Control flow diagram of the System Under test.

#### 7.1.2 Performance Measurement

Performance is measured on the basis of the comparison between the Existing approach and the implemented approach. The following two factors are taken into consideration

- Efficiency
- Effectiveness

#### 7.1.3 Parameter Setting

The parameters set in the experiment are the parameters of the Genetic Algorithm. The basic parameters of the Experiment are as follows:

<b>Initial Population Size</b>	50
<b>Selection Method</b>	Tournament Selection
<b>Crossover Method</b>	Single point crossover
<b>Mutation Method</b>	One point Mutation

Table 7.1: Parameters of GA

### 7.1.4 Experiment 1:

The Final Results of the 'Whole Test Suite Generation' technique are as follows. The population size is 50 here. The program is run 50 time and the average iterations and repetitions are calculated. This is repeated 10 times for confirmation, i.e the program is run 500 times with the initial population set as 50.

#### Whole test Suite Generation Results

The Results after running the test suite for a total of 500 times are given in the table below. The Average of every fifty readings is shown. The population size is 50.

Sr no	Total Repetition	Total Iterations
1	69	84.14
2	69	82.32
3	66	86.14
4	72	84.92
5	50	80.14
6	68	84.54
7	92	89.98
8	58	86.78
9	86	87
10	141	86.02

Table 7.2: Experiment 1 Existing approach Readings



The bar Chart for the total number of iterations, after calculation the average of the results obtained from every 50 runs is given below.

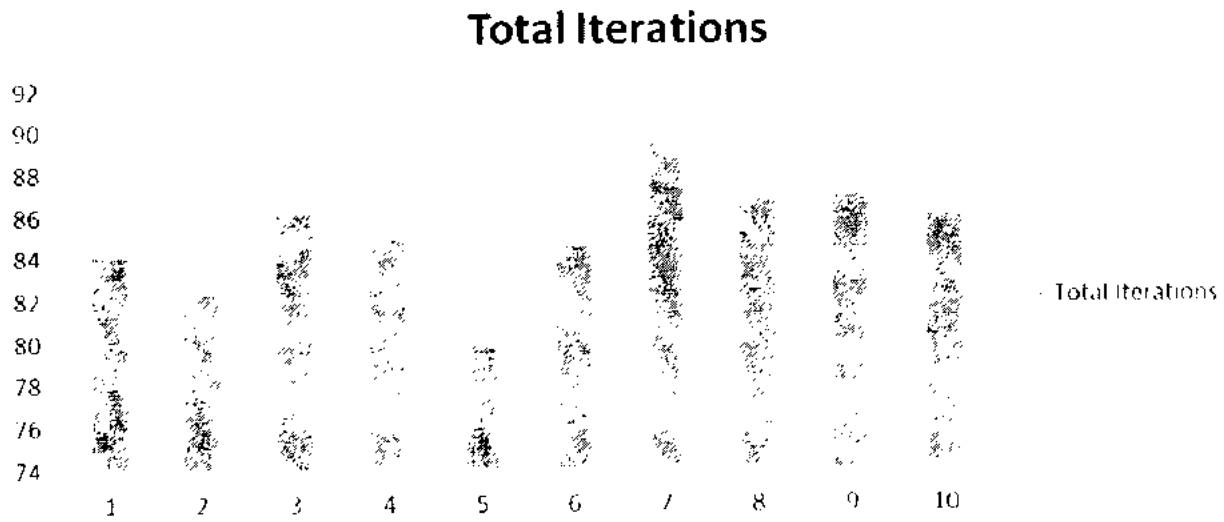


Fig 7 1: Total iterations Bar Chart, Experiment 1

The pie Chart of the readings obtained is given below. Each reading is an average of the readings obtained after running the program 50 times.

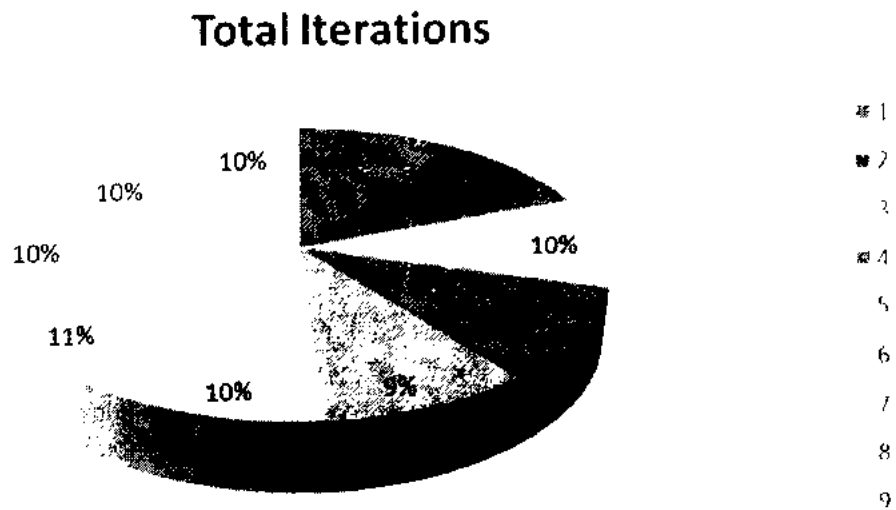


Fig 7.2. Total iterations Pie Chart, Experiment I

Repetitions here mean redundancy. The redundancy calculated in the program in 10 runs is shown below.

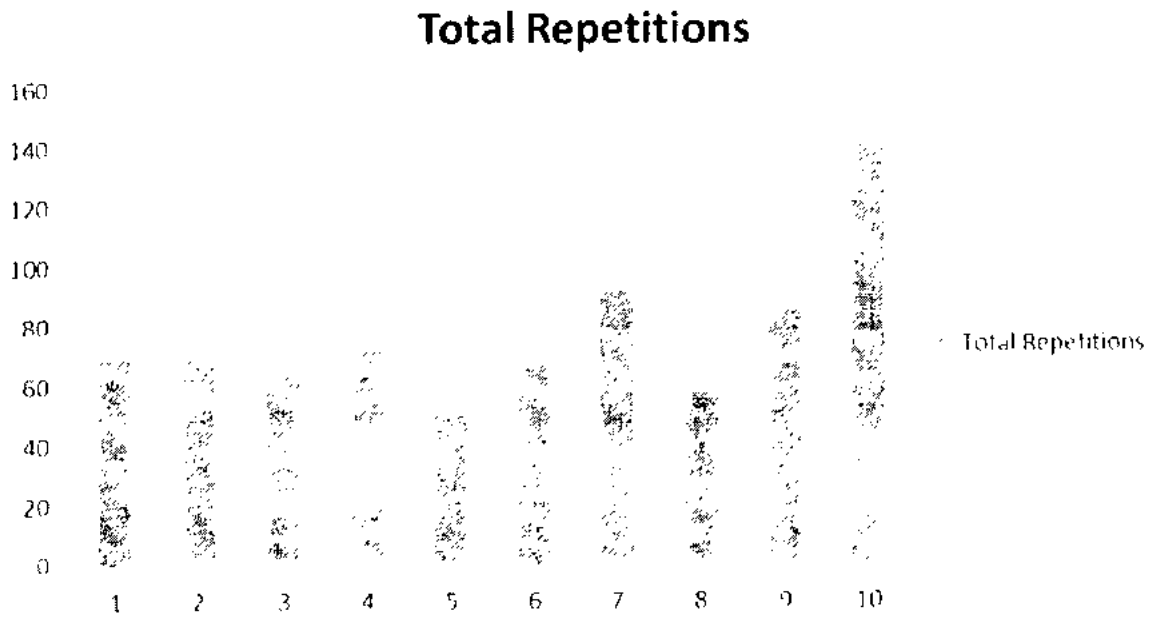


Fig 7.3: Total Repetitions Bar Chart, Experiment I

The pie chart of the redundancy that was calculated is given below.

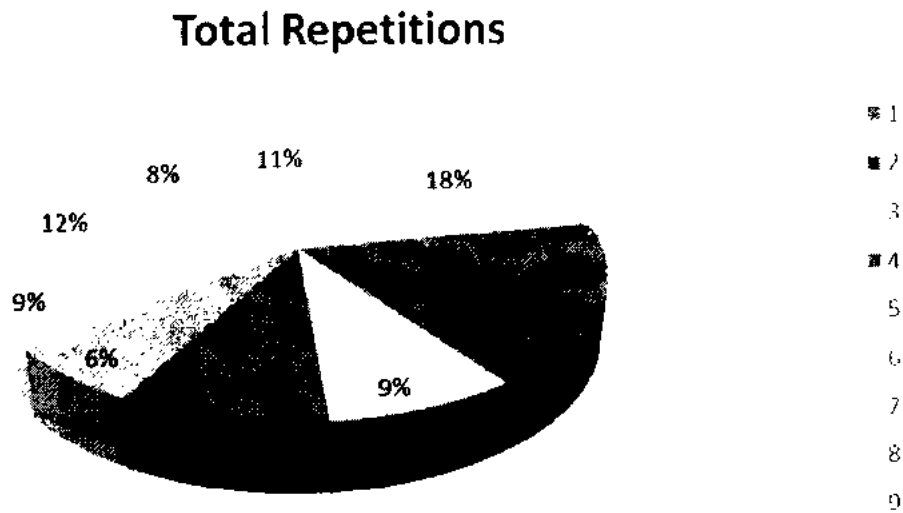


Fig 7.4: Total Repetitions Pie Chart, Experiment I

## Multi-Objective Whole test Suite Generation Results

The Readings after running the MO-WTS implementation after a total 500 times are shown. The value are the average of value obtained from 50 runs.

Sr no	Total Repetition	Total Iterations
1	0	25.96
2	0	27.62
3	0	30.26
4	0	27.54
5	0	25.54
6	0	25.9
7	0	29.88
8	0	29.28
9	0	31.6
10	0	25.36

Table 7.3. Experiment 1 Implemented approach Readings

The average of the Total Iterations after every 50 runs in the MO-WTS Gen implementation is shown below in the Bar chart.

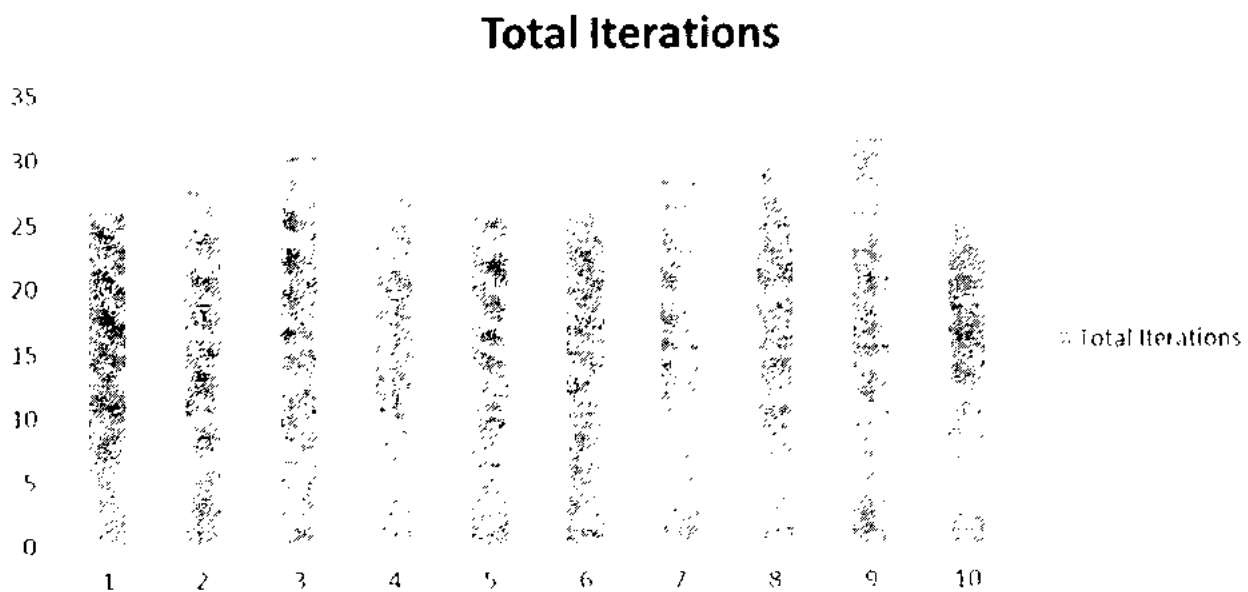


Fig 7.5. Total Iterations Bar Chart, Experiment 1

The Pie Chart of the total Iterations is given below.

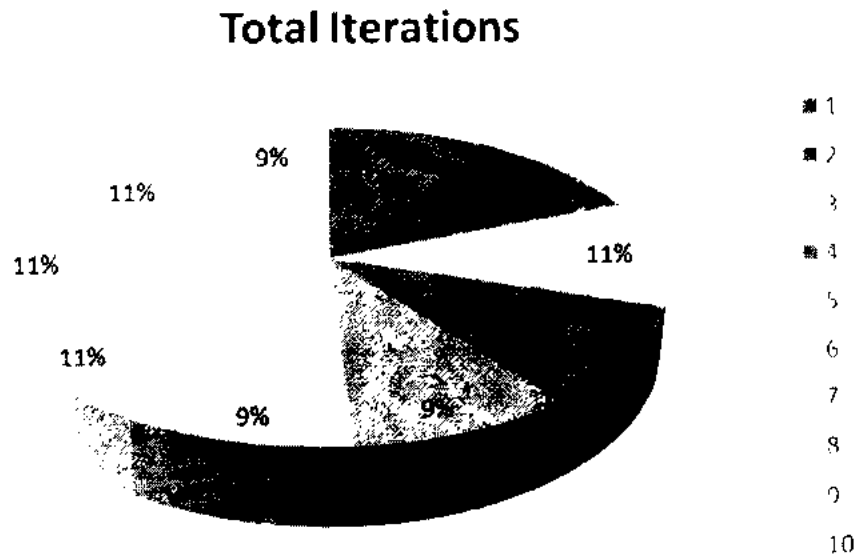


Fig 7.6: Total Iterations Pie Chart, Experiment 1

### Final Cumulative Results

The Average of all the above results is shown in the table below for comparison

<b>Whole test Suite Generation iterations</b>	<b>85.198</b>
<b>Multi-Objective test Suite Generation iterations</b>	<b>27.894</b>

Table 7.4: Experiment 1 Cumulative Results

The Total iterations for the two techniques are given below in the bar chart.

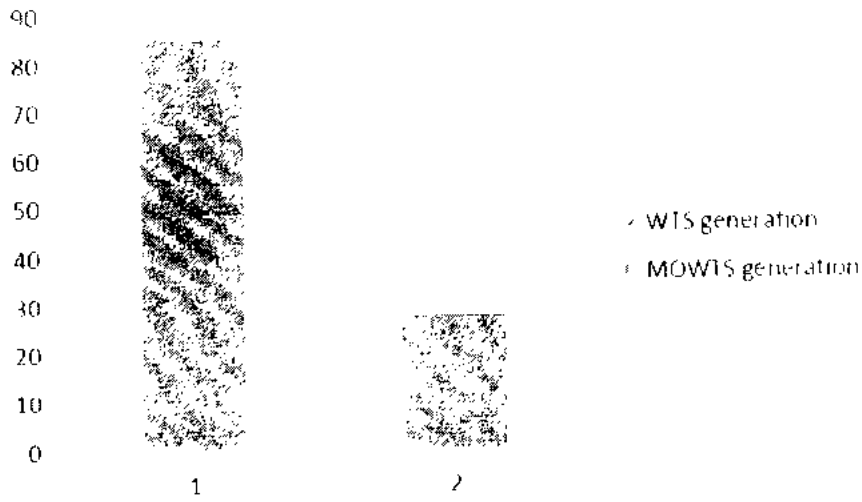


Fig 7.7: Iterations of both techniques

The pie Chart of the percentage of total number of Iterations in the two techniques is given below.

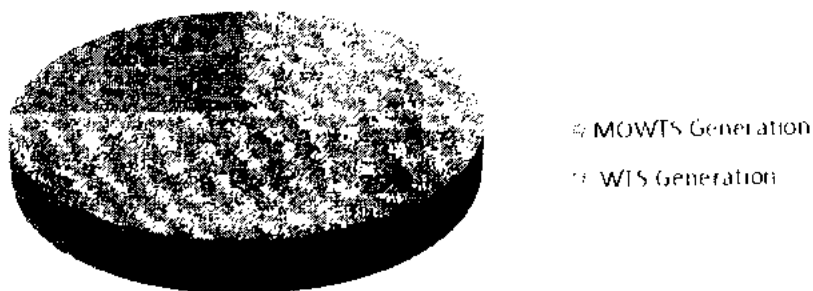


Fig 7.8: Pie Chart of the Iterations of both techniques

### 7.1.14 Final Total Average Results:

The total final summed up results of the existing approach and implemented approach that is shown below:

Experiment no	Existing Approach	Implemented approach
1	85.198	27.894
2	85.584	27.406
3	82.374	26.8
4	82.42	29.428
5	82.594	29.104
6	84.55	29.438
7	82.842	28.152
8	83.82	28.086
9	84.732	28.814
10	85.268	28.394

Table 7.32: Average of the Results

The bar Chart of the number of iterations in all the experiments in the existing and implemented approaches is given below.

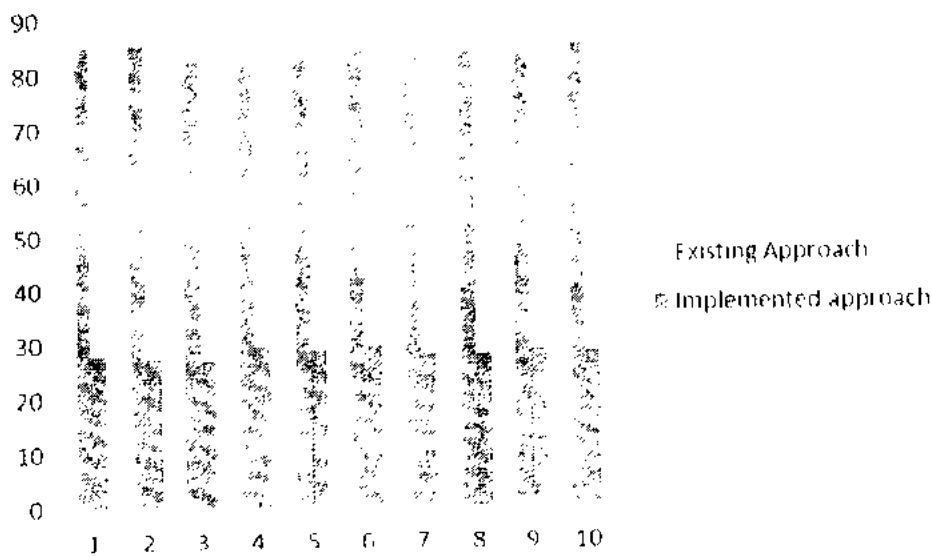


Fig 7.81: Cumulative Results bar Chart

The Pie Chart for the results obtained in all the experiments for the Whole Test Suite Generation approach is given below.

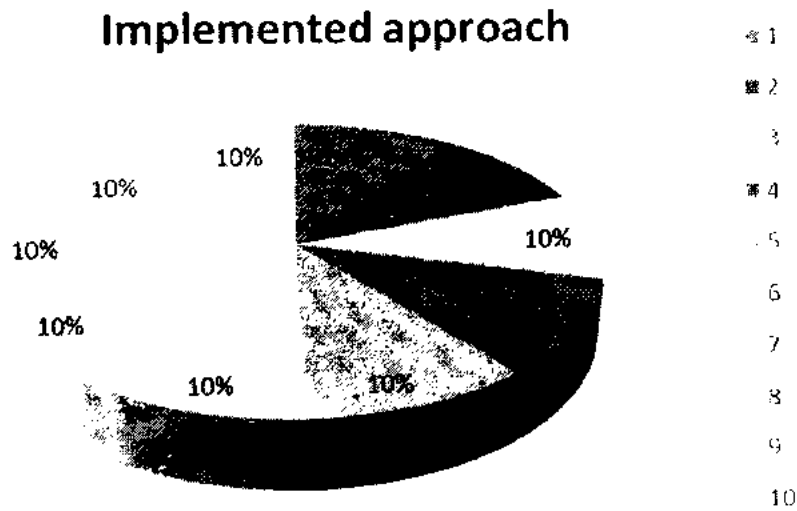


Fig 7.82: Implemented Approach Pie Chart

The Pie Chart for the results obtained in all the experiments for the Multi-Objective Whole Test Suite Generation approach is given below.

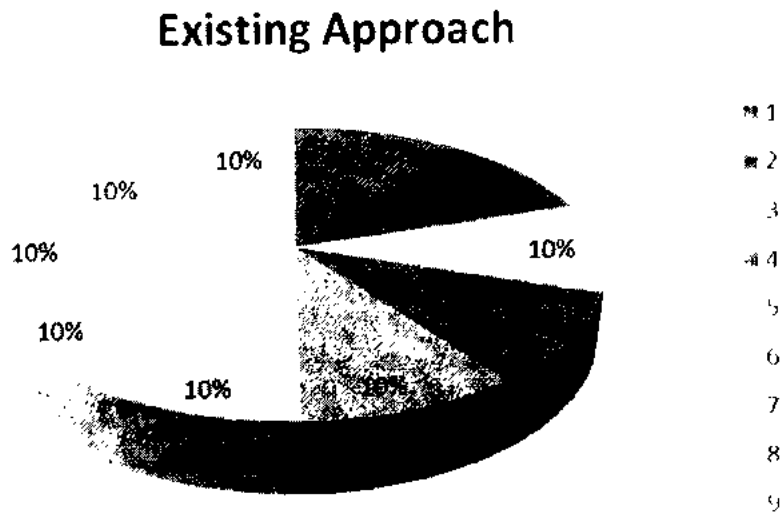


Fig 7.83: Existing Approach Pie Chart

## Total Average Sum-up of Results

The average of the results achieved in all the experiments for both the approaches are given below.

<b>Whole test Suite Generation iterations</b>	<b>83.9382</b>
<b>Multi-Objective test Suite Generation iterations</b>	<b>28.3516</b>

Table 7.33. Final Sum-up of Readings

The average of the 'number of iterations' achieved in all the experiments for both the approaches is given below in the Bar Chart.

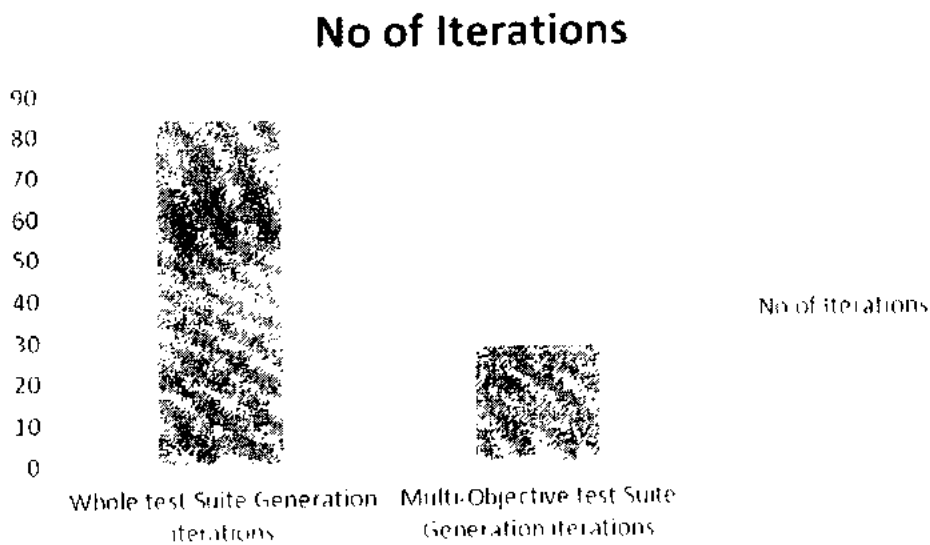


Fig 7 8-1: Average Cumulative results Bar chart



The pie chart of the average no of iterations in in all the experiments for both the approaches is given below.

## No of Iterations



Fig 7.85: Average Cumulative results Pie chart

## 7.2 Discussion

In section 7.1 of "experimental design", we discussed what parameters were used in the experiment and how the experiment was performed considering 10 different initial values of the initial population starting from 50 and reaching up to 10000.

After running the program with each initial value the results were shown in a bar chart and the pie chart. This section will compare the results with the existing work to access the improvements made in the implemented approach.

We implemented both the existing approach and the proposed approach and ran them on the same example with the same initial test data pool and then compared the results, the results showed a clear improvement in not just one but three area of the technique. The improvements were made in the following areas.

- The no of iterations to achieve an optimized test Suite was reduced.
- The redundancy in the Final test Suite was removed along with the optimization.
- Optimization and minimization were done simultaneously instead of one after the other and multiple objectives were achieved in a single run of the Genetic Algorithm.

- Hence a reduction in the size of the test suite was achieved, the time taken to achieve the complete coverage was reduced (Iterations) and the redundant coverage was eliminated.

The results were compared on the following basis.

- No of Iterations of GA
- Amount of Redundant coverage.
- No of Objectives achieved.

Both the approaches were run multiple times on the chosen example and the results that were achieved are shown in the table below.

Experiment No	Population Size	WTS Generation Approach		MOWTS generation Approach	
		No of Iterations	Average Total Repetitions	No of Iterations	Average Total Repetitions
1	50	85.198	77.1	27.894	0
2	100	85.584	83.2	27.406	0
3	500	82.374	100.7	26.8	0
4	1000	82.42	84.9	29.428	0
5	1500	82.594	88.6	29.104	0
6	3000	84.55	81.8	29.438	0
7	4000	82.842	80.4	28.152	0
8	5000	83.82	82	28.086	0
9	7000	84.732	74.6	28.814	0
10	10,000	85.268	76.6	28.394	0
<b>Average</b>		<b>83.9382</b>	<b>82.99</b>	<b>28.3516</b>	<b>0</b>

Table 7 34: Comparison of the Readings

### 7.3 Threats to Validity

In threats to validity, factors which have affected the results are as follows:

- If the tools are tested on a different example which is more complex and larger in scale then the results might differ.
- The results may vary when tested with different combinations of the GA parameters.
- The population size may also affect the results.

# **Chapter 8**

## **CONCLUSION**

## CONCLUSION AND FUTURE WORK:

This section sums up our main contribution for this thesis, gives some direction for the future work and also includes some concluding remarks. It proposes some enhancements to the implemented approach too.

### 8.1 Conclusion

Testing is the most time-consuming phase in software Development life-cycle. That is the reason why a lot of work is done in the literature in the direction of automating software testing. One of the parts of software test automation is the automation of the generation of test data.

Our work focuses on the automated generation of the test Data. The approach has two parts, the first part deals with the generation of the test suite, and the second part deals with the optimization of the test data i.e. minimization. Thus our main contribution with this approach is to create a tool that generates the test data which is optimized. That is the test Suite generated is not redundant while it provides complete coverage.

The technology our work is based on is the genetic algorithm. The existing technique used the genetic algorithm to achieve coverage but our approach uses Multi Objective Genetic Algorithms to achieve multiple targets. The tools was validated after its creation through the use of multiple experiments. On the basis of the selected parameters our work is more efficient as compared to Arcuri et al(2012). We proved our hypothesis that by using a multi-objective genetic algorithm an improvement can be made in the 'Whole test Suite Generation' technique.

### 8.2 Future Work

For the future work we aim to apply the technique on a large scale industrial case study to further validate the results we achieved. Since this work focuses on the white-box testing of the data we are planning on working on the black-box testing with this technique too.

As this technique is highly adaptable, it can generate interesting results to change the completion criteria from branch coverage to some other criteria like mutation detection.

Categorizing our results on the basis of the GA parameters and the initial population will help us further understand the finding of our research in more detail.

Currently only two targets are being considered in the Multi Objective test Suite generation of the whole test suite technique. It can be considered to add further objectives to the technique and assess the results. The new Objectives can be run-time reduction etc.

# REFERENCES

## REFERENCES

- [1] Koopman, Pieter, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. "Gast: Generic automated software testing." In *Implementation of Functional Languages*, pp. 84-100. Springer Berlin Heidelberg, 2003.
- [2] Gupta, R., Aditya P. Mathur, and Mary Lou Soffa. "Generating test data for branch coverage." *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*. IEEE, 2000.
- [3] Melanie, Mitchell. "An introduction to genetic algorithms." Cambridge, Massachusetts London, England, Fifth printing 3 (1999). ISBN:0-262-13316-4.
- [4] Pacheco, Carlos, and Michael D. Ernst. "Randoop: feedback-directed random testing for Java." *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007.
- [5] Harman, Mark, et al. "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem." *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*. IEEE, 2010.
- [6] Ribeiro, José Carlos Bregieiro. "Search-based test case generation for object-oriented java software using strongly-typed genetic programming." *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*. ACM, 2008.
- [7] Tonella, Paolo. "Evolutionary testing of classes." *ACM SIGSOFT Software Engineering Notes*. Vol. 29, No. 4. ACM, 2004.
- [8] Wappler, Stefan, and Frank Lammermann. "Using evolutionary algorithms for the unit testing of object-oriented software." *Proceedings of the 2005 conference on Genetic and evolutionary computation*. ACM, 2005.
- [9] Blue, Dale, et al. "Interaction-based test-suite minimization." *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.
- [10] Jeffrey, Dennis, and R. Gupta. "Improving fault detection capability by selectively retaining test cases during test suite reduction." *Software Engineering, IEEE Transactions on* 33.2 (2007): 108-123.



- [11] Korel, Bogdan. "Automated software test data generation." *IEEE Transactions on Software Engineering*, 16.8 (1990): 870-879.
- [12] Harman, Mark, and Phil McMinn. "A theoretical and empirical study of search-based testing: Local, global, and hybrid search." *IEEE Transactions on Software Engineering*, 36.2 (2010): 226-247.
- [13] Goldberg, Allen, Tie-Cheng Wang, and David Zimmerman. "Applications of feasible path analysis to program testing." *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 1994.
- [14] Fraser, Gordon, and Andrea Arcuri. "Evolutionary generation of whole test suites." *Quality Software (QSIC)*. 2011 11th International Conference on. IEEE, 2011.
- [15] Blackwell, Barry Mark, et al. "Testing tool comprising an automated multidimensional traceability matrix for implementing and validating complex software systems." U.S. Patent No. 7,490,319. 10 Feb. 2009.
- [16] Rothermel, Gregg, et al. "An empirical study of the effects of minimization on the fault detection capabilities of test suites." 1998. *Proceedings.. International Conference on Software Maintenance*. IEEE, 1998.
- [17] Andrews, James H., et al. "Using mutation analysis for assessing and comparing testing coverage criteria." *IEEE Transactions on Software Engineering*, 32.8 (2006): 608-624.
- [18] Michael, Christoph C., et al. "Genetic algorithms for dynamic test data generation." *Automated Software Engineering*, 1997. *Proceedings, 12th IEEE International Conference*. IEEE, 1997.
- [19] Ferguson, Roger, and Bogdan Korel. "The chaining approach for software test data generation." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5.1 (1996): 63-86.
- [20] Korel, Bogdan, and Janusz Laski. "Dynamic slicing of computer programs." *Journal of Systems and Software* 13.3 (1990): 187-195.

- [21] Mala, D. Jeya, and V. Mohan. "ABC Tester-Artificial bee colony based software test suite optimization approach." *International Journal of Software Engineering* 2.2 (2009): 15-43.
- [22] Rothermel G, Harrold MJ. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering* 1996; 22(8):529-551.
- [23] Rothermel G, Harrold M, Ronne J, Hong C. Empirical studies of test suite reduction. *Software Testing, Verification, and Reliability* 2002; 4(2):219-249.
- [24] Goldberg, David E., Bradley Korb, and Kalyanmoy Deb. "Messy genetic algorithms: Motivation, analysis, and first results." *Complex systems* 3, no. 5 (1989): 493-530.
- [25] Golberg, David E. "Genetic algorithms in search, optimization, and machine learning." Addison wesley 1989 (1989). ISBN:0201157675.
- [26] Konak, Abdullah, David W. Coit, and Alice E. Smith. "Multi-objective optimization using genetic algorithms: A tutorial." *Reliability Engineering & System Safety* 91.9 (2006): 992-1007.
- [27] Yoo, Shin, and Mark Harman. "Regression testing minimization, selection and prioritization: a survey." *Software Testing, Verification and Reliability* 22.2 (2012): 67-120.
- [28] Garey MR, Johnson DS. "Computers and Intractability: A Guide to the Theory of NP-Completeness. " W.H.Freeman and Company: New York, NY, 1979. ISBN-10: 0716710455.
- [29] Leung HKN, White L. "Insight into regression testing. " *Proceedings of the International Conference on Software Maintenance (ICSM 1989)*, IEEE Computer Society Press: Silver Spring, MD, 1989; 60-69.
- [30] Rothermel G, Harrold MJ. "A framework for evaluating regression test selection techniques." *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*. IEEE Computer Press: Silver Spring, MD, 1994; 201-210.
- [31] Wong WE, Horgan JR, London S, Mathur AP. "Effect of test set minimization on fault detection effectiveness." *Software Practice and Experience* 1998; 28(4):347-369.

- [32] Harrold MJ. "Testing evolving software." *The Journal of Systems and Software* 1999; 47(2-3):173-181.
- [33] Rothermel G, Untch RH, Chu C, Harrold MJ. "Test case prioritization: An empirical study." *Proceedings of the International Conference on Software Maintenance (ICSM 1999)*. IEEE Computer Press: Silver Spring, MD, 1999; 179-188.
- [34] Rothermel G, Untch RH, Chu C, Harrold MJ. "Test case prioritization: An empirical study." *Proceedings of the International Conference on Software Maintenance (ICSM 1999)*. IEEE Computer Press: Silver Spring, MD, 1999; 179-188.
- [35] Srinivas, Nidamarthi, and Kalyanmoy Deb. "Multiobjective optimization using nondominated sorting in genetic algorithms." *Evolutionary computation* 2,3 (1994): 221-248.
- [36] Elbaum S, Gable D, Rothermel G. "Understanding and measuring the sources of variation in the prioritization of regression test suites." *Proceedings of the Seventh International Software Metrics Symposium (METRICS 2001)*. IEEE Computer Press: Silver Spring, MD, 2001; 169-179.
- [37] Elbaum, Sebastian, Alexey Malishevsky, and Gregg Rothermel. "Incorporating varying test costs and fault severities into test case prioritization." *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001.
- [38] Malishevsky A, Rothermel G, Elbaum S. "Modeling the cost-benefits tradeoffs for regression testing techniques." *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*. IEEE Computer Press: Silver Spring, MD, 2002; 230-240.
- [39] Rothermel, Gregg, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia. "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. ACM, 2002.
- [40] Rothermel G, Untch RJ, Chu C. "Prioritizing test cases for regression testing." *IEEE Transactions on Software Engineering* 2001; 27(10):929-948.
- [41] Budd TA. "Mutation analysis of program test data." PhD Thesis, Yale University, New Haven, CT, U.S.A., 1980.

- [42] Leon D, Podgurski A. "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases." Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2003). IEEE Computer Press: Silver Spring, MD, 2003: 442--456.
- [43] Tonella P, Avesani P, Susi A. "Using the case-based ranking methodology for test case prioritization." Proceedings of the 22nd International Conference on Software Maintenance (ICSM 2006). IEEE Computer Society: SilverSpring, MD, 2006: 123--133.
- [44] Yoo S, Harman M, Tonella P, Susi A. "Clustering test cases to achieve effective & scalable prioritization incorporating expert knowledge." Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2009). ACM Press: New York, 2009: 201-211.
- [45] Srikanth H, Williams L, Osborne J. "System test case prioritization of new and regression test cases." Proceedings of the International Symposium on Empirical Software Engineering. IEEE Computer Society Press: Silver Spring, MD, 2005: 64-73.
- [46] Walcott KR, Soffa ML, Kapfhammer GM, Roos RS. "Time aware test suite prioritization." Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006). ACM Press: New York, 2006: 1-12.
- [47] Yoo S, Harman M. "Pareto efficient multi-objective test case selection." Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007). ACM Press: New York, 2007: 140--150.
- [48] Do H, Mirarab SM, Tahvildari L, Rothermel G. "An empirical study of the effect of time constraints on the cost-benefits of regression testing". Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM Press: New York, 2008: 71-82.
- [49] Chen TY, Lau MF. "Dividing strategies for the optimization of a test suite." Information Processing Letters 1996; 60(3):135-141.
- [50] Harrold MJ, Gupta R, Soffa ML. "A methodology for controlling the size of a test suite." ACM Transactions on Software Engineering and Methodology 1993; 2(3):270-285.

- [51] Horgan J, London S. "ATAC: A data flow coverage testing tool for c. Proceedings of the Symposium on Assessment of Quality Software Development Tools." IEEE Computer Society Press: Silver Spring, MD. 1992: 2-10.
- [52] Offutt J, Pan J, Voas J. "Procedures for reducing the size of coverage-based test sets." Proceedings of the 12th International Conference on Testing Computer Software. ACM Press: New York. 1995: 111-123.
- [53] Rothermel G, Harrold MJ, Ostrin J, Hong C. "An empirical study of the effects of minimization on the fault detection capabilities of test suites". Proceedings of the International Conference on Software Maintenance (ICSM 1998). IEEE Computer Press: Silver Spring, MD. 1998: 34-43.
- [54] Wong WE, Horgan JR, London S, Mathur AP. "Effect of test set minimization on fault detection effectiveness." *Software Practice and Experience* 1998; 28(4):347-369.
- [55] Wong WE, Horgan JR, Mathur AP, Pasquini A. "Test set size minimization and fault detection effectiveness: A case study in a space application " *The Journal of Systems and Software* 1999; 48(2):79-89.
- [56] Yu, Yanbing, James A. Jones, and Mary Jean Harrold. "An empirical study of the effects of test-suite reduction on fault localization." Proceedings of the 30th international conference on Software engineering. ACM, 2008.
- [57] Zhong, Hao, Lu Zhang, and Hong Mei. "An experimental comparison of four test suite reduction techniques." Proceedings of the 28th international conference on Software engineering. ACM, 2006.
- [58] McMaster S, Memon AM. "Call stack coverage for test suite reduction." Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05). IEEE Computer Society: Washington, DC, U.S.A., 2005: 539-548.
- [59] Edvardsson, Jon. "A survey on automatic test data generation." Proceedings of the 2nd Conference on Computer Science and Engineering, 1999.
- [60] W. H. Deason, D. Brown, K. Chang, and J. H. Cross II. "A rule-based software test data generator." *IEEE Transactions on Knowledge and Data Engineering*, 3(1):108-117, March 1991.

- [61] R. Ferguson and B. Korel. "The chaining approach for software test data generation." *IEEE Transactions on Software Engineering*, 5(1):63-86, January 1996.
- [62] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. "On the automated generation of program test data." *IEEE Transactions on Software Engineering*, SE-2(4):293-300, December 1976.
- [63] W. H. Deason, D. Brown, K. Chang, and J. H. Cross II. "A rule-based software test data generator." *IEEE Transactions on Knowledge and Data Engineering*, 3(1):108-117, March 1991.
- [64] N. Gupta, A. P. Mathur, and M. L. Soffa. "Automated test data generation using an iterative relaxation method." In *Proceedings of the ACM SIGSOFT sixth international symposium on Foundations of software engineering*, pages 231-244, November 1998.
- [65] R. E. Prather and J. P. Myers, Jr. "The path prefix software testing strategy." *IEEE Transactions on Software Engineering*, SE-13(7):761-765, July 1987.
- [66] N. Tracey, J. Clark, and K. Mander. "Automated program flaw finding using simulated annealing." In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, volume 23, pages 73-81, March 1998.
- [67] C. Michael and G. McGraw. "Automated software test data generation for complex programs." In *13th IEEE International Conference on Automated Software Engineering*, pages 136-146, October 1998.
- [68] McMinn, Phil. "Search-based software test data generation: a survey." *Software testing, Verification and reliability* 14.2 (2004): 105-156.
- [69] M. Harman and B. Jones. "Search-based software engineering." *Information and Software Technology*, 43(14):833-839, 2001.
- [70] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. "Reformulating software engineering as a search problem." *IEEE Proceedings-Software*, 150(3):161-175, 2003.
- [71] Fraser, Gordon, and Andrea Arcuri. "Evolutionary generation of whole test suites." *Quality Software (QSIC)*, 2011 11th International Conference on. IEEE, 2011.

- [72] M. Harman, S. G. Kim, K. Lakhota, P. McMinn, and S. Yoo., "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem." in SBST'10: Proceedings of the International Workshop on Search-Based Software Testing. IEEE Computer Society, 2010, pp. 182- 191.
- [73] Glover.F. C. Macmillan (1986). "The General Employee Scheduling Problem: An Integeration of Management Science and Artificial Intelligence." *Computers and Operations Research*, 15:5, 563-593.
- [74] Blum, Christian, and Andrea Roli. "Metaheuristics in combinatorial optimization: Overview and conceptual comparison." *ACM Computing Surveys (CSUR)* 35.3 (2003): 268-308.
- [75] Holland, John H. "Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence." U Michigan Press, 1975. ISBN 0472084607 9780472084609.
- [76] Fogel, Lawrence J., Alvin J. Owens, and Michael J. Walsh. "Artificial intelligence through simulated evolution." (1966).
- [77] Kirkpatrick, Scott. "Optimization by simulated annealing: Quantitative studies." *Journal of statistical physics* 34.5-6 (1984): 975-986.
- [78] Dorigo, Marco, Vittorio Maniezzo, and Alberto Colomi. "Ant system: optimization by a colony of cooperating agents." *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* 26.1 (1996): 29-41.
- [79] Eberhart, Russ C., and James Kennedy. "A new optimizer using particle swarm theory." *Proceedings of the sixth international symposium on micro machine and human science*. Vol. 1, 1995.
- [80] Karaboga, Dervis. "An idea based on honey bee swarm for numerical optimization." Vol. 200. Technical report-tr06, Erciyes university, engineering faculty, computer engineering department, 2005.
- [81] Glover, Fred. "Tabu search-part I." *ORSA Journal on computing* 1.3 (1989): 190-206.
- [82] Geem, Zong Woo, Joong Hoon Kim, and G. V. Loganathan. "A new heuristic optimization algorithm: harmony search." *Simulation* 76.2 (2001): 60-68.

- [83] Lukasik, Szymon, and Slawomir Zak. "Firefly algorithm for continuous constrained optimization tasks." *Computational Collective Intelligence. Semantic Web, Social Networks and Multiagent Systems*. Springer Berlin Heidelberg, 2009. 97-106.
- [84] Asoudeh, Nesa, and Yvan Labiche. "A multi-objective genetic algorithm for generating test suites from extended finite state machines." *Search Based Software Engineering*. Springer Berlin Heidelberg, 2013. 288-293.
- [85] Gong, Dunwei, Tian Tian, and Xiangjuan Yao. "Grouping target paths for evolutionary generation of test data in parallel." *Journal of Systems and Software* 85.11 (2012): 2531-2540.
- [86] Obuchowicz, Andrzej. "Multidimensional mutations in evolutionary algorithms based on real-valued representation." *International Journal of Systems Science* 34.7 (2003): 469-483.
- [87] Deb, Kalyanmoy. "Multi-objective optimization using evolutionary algorithms." Vol. 16. John Wiley & Sons, 2001, ISBN: 978-0-471-87339-6.
- [88] Deb, Kalyanmoy, et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II." *IEEE Transactions on Evolutionary Computation*, 6.2 (2002): 182-197.



## REFERENCES

# APENDIX A: GENERATED CODE



**GENERATED CODE:****Algorithm for whole test Suite generation**

Algorithm for Single Objective Whole test Suite Generation is as follows:

**Generate Initial Population**

ARRAY AllStatements[]: (Populate array with statements from the system under Test):

.....Generate TestCases.....

FOR(i=0 TO MAXTestCase)

    ARRAY TestCase[] = RAND(AllStatements[])

END FOR

.....Generate TestSuites.....

FOR(i=0 TO MAXTestSuite)

    ARRAY TestSuite []= TestSuite[] + TestCase[RAND]

END FOR

**Selection**

.....Select the fittest Solutions.....

FOR(i=0 TO MAXTestSuite)

    ARRAY Fitness[i] = MTotal[i] – Mcovered[i]

END FOR

FOR(i=0 TO MAXPopulation)

    ARRAY Selected[] = max(Fitness[])

END FOR

**Crossover**

```

.....Crossover Selected Solutions.....
FOR (i=1 TO MAXPopulation: i=i+2)
  FOR (j=1 TO MAXTestCase)
    Child[i][j] = Selected[i][j]to(j/2)+Selected[i+1][j/2toMAX]
    Child[i+1][j] =Selected[i+1][j]to (j/2)+Selected[i][j/2to MAX]
  END FOR
END FOR

```

### **Mutation**

```

.....Perform Mutation on the child Population.....
MutationProb = RAND(1-3)
.....Remove a Random TestCase.....
If MutationProb = 1
  MutationTestCase[] = TestCase[RAND]
  Remove MutationTestCase [RAND]
  .....Add a Random TestCase.....
ELSE IF MutationProb = 2
  MutationTestCase[] = TestCase[RAND]
  Add MutationTestCase [RAND]
  .....Replace a Random test Case.....
ELSE IF MutationProb = e
  MutationTestCase[] = TestCase[RAND]
  Modify MutationTestCase [RAND]
END IF

```

**Main Flow of Algorithm**

Initialize Population

.....Genetic Algorithm Iteration.....

FOR(I=1 TO MAXIteration)

    Perform Selection

    Perform Crossover

    Perform Crossover

    Check Fitness

IF (Fitness = RequiredFitness)

    Terminate program

    Output TestSuite

END IF

END FOR

**Algorithm for Multi-Objective Objective Whole test Suite Generation is as follows:**

**Generate Initial Population**

ARRAY AllStatements[]: (Populate array with statements from the system under Test):

.....Generate TestCases.....

FOR(i=0 TO MAXTestCase)

    ARRAY TestCase[] = RAND(AllStatements[])

END FOR

.....Generate TestSuites.....

```

FOR(i=0 TO MAXTestSuite)
  ARRAY TestSuite []= TestSuite[] · TestCase[RAND]
END FOR

```

### **Fast Non-Dominated Sort**

```

.....Calculate both fitnesses.....
FOR(i=0 TO MAXTestSuite)
  ARRAY Fitness1[i] = MTotal[i] -- Mcovered[i]
  j=1
  WHILE NOT End of TestCase
    ARRAY [j] = SUM(TestCaseMethodRepetitions[j])
    j=j+1
  END WHILE
END FOR
i=0
k=1
.....Fast Non Dominant Sorting .....
WHILE ANY POPULATION NOT EMPTY
  WHILE NOT END OF POPULATION
    FOR(j=1 TO Population)
      IF Fitness1[i] AND Fitness2[i] > Fitness1[j] AND Fitness2[j]
        ARRAY DominantPool[] = TestSuite[i] AND REMOVE TestSuite[i] from Population
      ELSE IF Fitness1[i] AND Fitness2[i] < Fitness1[j] AND Fitness2[j]
        ARRAY Remove TestSuite[j] From Dominant Pool[] AND Population = TestSuite [i]

```

```

    Add TestSuite[i] to DominantPool[]
ELSE IF Fitness1[i] AND Fitness2[i] ≠ Fitness1[j] AND Fitness2[j]
    Add TestSuite[i] AND TestSuite[j] to DominantPool[] AND Remove TestSuite[i] AND
    TestSuite[j] from Population
END IF
i=i+1
END WHILE
ARRAY Front[k] =DominantPool[]
k=k+1
END WHILE

```

### Calculate Crowding Distance

```

.....Calculate Crowding distance according to first fitness function.....
FOR (i=0 TO MAXFronts)
    Sort TestSuites by Fitness1
    FOR (j=0 TO MAXTestSuites)
        If (j=1 OR j=MAX)
            crowdingDistance1[j] = ∞
        ELSE
            crowdingDistance1[j] = ABSOLUTE(Fitness1(TestSuite[j-1]) - Fitness1(TestSuite[j+1]))
        END IF
    END FOR
.....Calculate Crowding distance according to second fitness function.....
Sort TestSuites by Fitness2

```



```

FOR (j=0 TO MAXTestSuites)
  If (j=1 OR j=MAX)
    crowdingDistance2[j] = ∞
  ELSE
    crowdingDistance2[j] =ABSOLUTE(Fitness2(TestSuite[j-1]) - Fitness2(TestSuite[j+1]))
  END IF
END FOR

```

.....Calculate The Cumulative Crowding Distance.....

```

FOR (j=0 TO MAXTestSuites)
  crowdingDistance[j] = crowdingDistance1[j] + crowdingDistance2[j]
END FOR

```

### **Selection**

.....Select a pre-defined number of random test Suites.....

```

FOR(i=0 TO PredefinedRand)
  ARRAY RandomChoice[] = RAND(TestSuite[])
END FOR

```

.....Choose the fittest among the randomly chosen.....

```

FOR(i=0 TO PredefinedRand)
  ARRAY Selected[] = Selected[] + MAX(CrowdingComparison Operator(TestSuite[i]))
  REMOVE MAX(CrowdingComparison Operator(TestSuite[i]))
END FOR

```

**Crossover**

```

.....Crossover Selected Solutions.....
FOR (i=1 TO MAXPopulation; i=i+2)
  FOR (j=1 TO MAXTestCase)
    Child[i][j] = Selected[i][jto(j/2)]+Selected[i+1][jtoMAX]
    Child[i+1][j] =Selected[i+1][jto (j/2)]+Selected[i][jto MAX]
  END FOR
END FOR

```

**Mutation**

```

.....Perform Mutation on the child Population.....
MutationProb = RAND(1-3)
If MutationProb = 1
  MutationTestCase[] = TestCase[RAND]
  Remove MutationTestCase [RAND]
ELSE IF MutationProb = 2
  MutationTestCase[] = TestCase[RAND]
  Add MutationTestCase [RAND]
ELSE IF MutationProb = e
  MutationTestCase[] = TestCase[RAND]
  Modify MutationTestCase [RAND]
END IF

```

**Main Program Flow**

```

.....Iteratively Run the genetic Algorithm until optimized, minimized solution found.....
Multi-Objective Whole test Suite Generation

```

Generate Initial Population

FOR(i=0 TO Population)

    Fitness1 = fitness(TestSuite[i])

    Fitness2 = fitness(TestSuite[i])

Front[i] = FastNonDominantSort(Population)

END FOR

FOR(i=0 TO NO OF Fronts)

    WHILE NOT END OF FRONT

        crowdingDistance[i] = crowdingDistance(TestSuite[i])

    END WHILE

END FOR

Selected [] = TournamentSelection(Fronts[])

ChildPopulation[] = Crossover(Selected[])

MutatedPopulation = Mutation(ChildPopulation[])

If (Fitness1(Population) And Fitness2(Population) = RequiredFitness1 AND RequiredFitness2 )

    Terminate Program

    Return TestSuite[] with Required Fitnesses

END IF