CHAPTER 1

INTRODUCTION

The rapid development of software made testing the significant choice of researchers because of its major and challenging role in software development. Testing is a process of locating defects and evaluating the features of the software. Even when the software is developed with utmost care using a precise regulation, it may contain a significant number of bugs. Testing is needed to guarantee the software quality and reliability. When the testing is performed, it should be done in a well-defined manner so as to locate all the defects of the software.

To develop bug free quality software, test cases are very important. They should be written in such a way to adequately test the software throughout its life cycle. This is because, even after the software system is implemented, there may still be a need for testing again due to technology change, software bugs, addition or deletion of any functionality of the software. So, the maintenance activity called regression testing can be used to ensure that changes made in the program for the purposes of increasing functionality or removing faults does not have any negative impact on the correctness of the software. The effectiveness of the test case can still be improved by having multi objective test cases. Because, in real world situations single objective test cases are obsolete.

This thesis focuses on prioritizing the multi objective test cases in a system while conducting the regression testing so as to reduce the number of test cases in a test suite and prioritize them while retaining the high
percentage of the original test suite’s fault detection effectiveness. In this work, in order to maximize the effectiveness of the system the reduction, optimization and prioritization of test cases are combined using mRMR feature selection, birds flocking and genetic algorithm. So, an integrated approach is introduced for multi objective test case prioritization for regression testing.

Feature selection removes redundancy and selects maximum relevance test cases. The flocking algorithm solves intricate optimization. Genetic algorithm is a robust search technique, often utilized to identify the accurate or approximate solutions for multi-objective optimization and prioritization problems.

1.1 TESTING AND ITS SIGNIFICANCE IN SOFTWARE

Testing (IEEE 1990) is defined as the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

Testing is the integral part of any software development as shown in Figure 1.1. In today’s world everything is software centric, that is, everything is becoming automatic. Usually software is developed based on the user requirements. After development, it is to be checked whether the requirements of the user are satisfied or not and also the correctness of the software using some technique. The technique of checking is actually the testing. Testing is the process of analyzing a software system to detect the differences between existing and required conditions in order to evaluate the features of the software system.

Figure 1.1 clearly shows the significance of testing in a Software Development Life Cycle (SDLC). SDLC is incomplete without testing.
Testing can be done initially and also whenever there is change in the software. If there is a change in any part of software then the software should be tested for its correctness even after the change using regression testing.

![Software Development Life Cycle](image)

**Figure 1.1 Software development life cycle**

### 1.2 REGRESSION TESTING

Regression testing (IEEE 1990) is defined as the selective retesting of a system or component to verify that modifications have not caused unintended effects that the system or component still complies with its specified requirements.

Regression testing is done after a change in the existing system features. The change may be due to change in code, change in user requirements, change in functionality after a change in technology. And the change may also be due to inclusion, deletion or modification of some
features in the existing system. After the change, the software has to be checked for its correctness, because sometimes bugs may be introduced due to the changes. Regression testing is mainly a maintenance activity. The correctness of the software system is checked by using the test suite. Test suite is the collection of test cases.

Test case (IEEE 1990) is defined as a set of test inputs, execution conditions, and expected results developed for a particular objective such as to exercise a particular program path or to verify compliance with specific requirement.

So, a test case is a set of conditions or variables under which a tester will determine if a requirement upon an application is partially or fully satisfied. A good test case should detect maximum bugs, minimize time and cost, assess conformance to specification, minimize risk, assess quality, verify correctness of the product and ensure quality.

The test cases play a major role in regression testing. So, they must be written in such a way that it ensures the correctness of both initial software and the changed one. In regression testing, the original test suite may be used or a changed test suite may be used to test the program to ensure that the program works properly and it does not have any negative impact on the functionality, correctness and reliability of the software even after the change that has been made in the software.

1.2.1 Techniques of Regression Testing

Figure 1.2 shows the techniques of regression testing. The simple method of regression testing uses the concept of retest all test cases. In this, all the test cases are executed again using the changed system. But the main overhead of this type of regression testing is the cost it incurs. The cost may
be due to additional time, memory, manpower, etc., that is needed to test the system again with the original set of test cases.

![Diagram](image)

**Figure 1.2 Techniques of regression testing**

There are many ways for reducing the cost, since testing is a must for the maintenance of the system. Test case reduction, test case optimization and test case prioritization are the mostly used techniques for reducing cost and memory usage.

Test suite reduction is a process in which the redundant and irrelevant test cases are eliminated from the test suite based on a criterion.

In test case optimization, only a subset of the test cases from an existing test suite is selected based on some criterion. But the disadvantage of this technique is that the fault detection rate may be less than that of the original test suite.

Test case prioritization is another technique in regression testing usually used to reduce the cost. It may not be possible to execute all the test cases in all the testing iteration due to resource constraints. Therefore, prioritization is done in which the test cases are ordered such that those with higher priorities are run earlier than those with lower priorities based on some
criterion. Prioritization techniques are usually preferred than optimization techniques because prioritization deals with original test suite and they do not eliminate any tests from the initial test suite. Test case prioritization techniques order test cases in an order that increases their effectiveness in testing.

So it is clearly shown that, while performing regression testing, the effectiveness of testing depends mainly on the test cases that are used. The effectiveness of the test cases in turn depends on the number of objectives like execution time, completeness, memory usage and fault coverage. So, the testing effectiveness can be improved by having a test suite with multi objective test cases.

1.3 SINGLE OBJECTIVE OR MULTI OBJECTIVE OPTIMIZATION

Test case optimization may be single objective or multi objective.

1.3.1 Single Objective Optimization

Single objective optimization is used to find the best solution based on a single criterion which corresponds either to the minimum or maximum value of that single criterion.

![Figure 1.3 Single objective optimization](image)

Figure 1.3 Single objective optimization

Figure 1.3 gives the basic structure of a single objective function optimization. In this optimization, the test suite is optimized based on a single objective function like minimizing cost or maximizing fault coverage etc.
This type of optimization is a useful tool for the decision makers with insights into the nature of the problem. But usually it cannot provide a set of alternative solutions that trade off different objectives against each other.

1.3.2 Multi Objective Optimization

Many real-world decision making problems need to use several conflicting objectives like minimize cost, maximize coverage, minimize memory usage and maximize fault detection rate etc. So, it is necessary to use multi objective optimization as depicted in Figure 1.4, which deals with conflicting objectives.

![Figure 1.4 Multi objective optimization](image-url)
Multi objective optimization or multi objective programming or pareto optimization (Steuer 1986) also known as multi-criteria or multi-attribute optimization is the process of simultaneously optimizing two or more conflicting objectives subject to certain constraints. The interaction among different objectives gives a set of compromised solutions, largely known as the trade-off, non dominated, non inferior or Pareto-optimal solutions.

Figure 1.4 gives the basic structure of multi objective function optimization. In this type of optimization, the test suite is optimized based on multiple objective functions like minimizing cost, maximizing fault coverage, minimizing memory usage and maximizing coverage etc.

1.4 OBJECTIVES OF THE WORK

The main objective of the present work is to prioritize the multi objective test cases in a regression testing system while retaining the high percentage of the original suite’s fault detection effectiveness.

The work reported in the proposed thesis is organized into 7 Chapters. Chapter 1 presents the problem under investigation, objectives of the present work, scope; organization of the thesis and forming the objective function. Chapter 2 gives a brief review of literature on regression testing, feature selection, birds flocking and genetic algorithm for test case prioritization. In Chapter 3, the reduction of test suite using mRMR feature selection is discussed. In Chapter 4, the optimization of test suite using birds flocking algorithm is discussed. In Chapter 5, the prioritization of test suite using genetic algorithm is discussed. Chapter 6 focuses on the performance analysis of the real time implementation of test case prioritization system. Chapter 7 presents the highlights of the work done, summary, conclusion and the directions for the further research. The references are listed at the end.
Figure 1.5 An integrated system for test case prioritization
Problem Statement:

To prioritize the multi objective test cases in a regression testing system while retaining the high percentage of the original suite’s fault detection effectiveness.

An integrated approach proposed for test case prioritization is represented in Figure 1.5 includes,

- Test case reduction using minimum-Redundancy-Maximum-Relevance (mRMR) feature selection.
- Test case optimization using Birds Flocking Algorithm.
- Test case prioritization using Genetic Algorithm (GA).

1.4.1 Test Case Reduction using mRMR Feature Selection

The minimum Redundancy Maximum Relevance (mRMR) feature selection algorithm is used to select test cases from the test suite with relevant objectives such as maximum fault coverage, minimum execution time, maximum completeness and minimum memory usage. The algorithm identifies test cases from test suite that are relevant to the objectives used and is normally called Maximum Relevance. These test cases often contain test cases which is relevant but redundant and mRMR attempts to address this problem by removing those redundant test cases.

1.4.2 Test Case Optimization using Birds Flocking Algorithm

The flocking algorithm imitates the flocking activities of birds on a computer. Particularly, in flocking algorithm, there is no leader meaning no global control. In birds flocking algorithm, the motion of flocks of birds are characterized as individual objectives. The birds flock normally in groups.
And the group of birds will have similar behavior. The flocking model consists of three behaviors: separation, cohesion and alignment.

The output from mRMR feature selection is used as input to birds flocking. Test cases are separated from one another in separation. They are pushed separately to keep them away from crashing into one another by maintaining the distance from nearby clusters. That is a certain distance is maintained from other surrounding neighbors. Thus, wide range of area is used.

In cohesion, a steering force is applied to move test cases to the objective functions to which they are related. This will make the group of test cases to move towards each other. In general, cohesion gives the ability to coalesce with other nearby test cases to form a cluster.

Alignment drives the test cases to form clusters with matching objectives. The test cases are aligned to the flock mates in neighborhood which have the similar objective. In general, the alignment gives test case the potential to align with other nearby clusters.

The flocking algorithm removes the noise as well as the outliers. It identifies the core points and thus the needed clusters are built. These clusters have relatively uniform group of points. The birds flocking algorithm outputs the test case clusters.

1.4.3 Test Case Prioritization using Genetic Algorithm

After using the flocking algorithm, a genetic algorithm is used to discover the most desirable test cases. In order to obtain the desirable test cases, the output of the flocking is subjected to genetic algorithm. The output from birds flocking algorithm is taken as the input for genetic algorithm.
In genetic algorithm, there are four main steps

- Chromosome generation
- Selection
- Crossover
- Mutation.

**Chromosome generation**: The first step in genetic algorithm is chromosome generation. The genetic algorithm solves the optimization problem by creating a population of chromosomes, which is a set of possible solutions for the problem.

**Selection**: In the selection process, the GA works with a set of chromosomes called population. First of all, several individual solutions are randomly created in order to form initial population from flocking algorithm. Normally, the population is randomly initialized. The population gives better solutions as the search evolves and in the long run when it converges. In order to form a new generation, a proportion of the existing population is chosen during each consecutive generation. For selecting the fittest among the population, the fitness factor is calculated. A fitness based process is performed to select the fitter solution. The fitter solution is a measure of fitness function.

**Crossover**: The salient operator in GA is crossover. In crossover, two parent chromosomes are pooled together to form new chromosomes called offspring. From the existing chromosomes, the parents are chosen with preference towards fitness so that offspring is expected to appear good genes that make the parents fitter.
**Mutation:** Mutation is playing an imperative role in genetic algorithm, which reintroduces genetic diversity back into the population and assists the search escape from local optima. In mutation, the characteristics of chromosomes are changed randomly. Normally, it is applied at the gene level. The mutation rate is very small in implementations, usually less than 1%.

The completion of these four steps completes one cycle in GA. The process of GA is stopped when the stopping criterion is met. The prioritized test cases are the outcome from GA.

### 1.4.4 Formation of Objective Function

In this research work, the test case reduction using mRMR feature selection, the test case optimization using birds flocking and test case prioritization using genetic algorithm are done based on the objective function which considers the objectives like memory usage, execution time (cost), completeness (coverage) and fault coverage.

The final objective function $Z$ as in Equation (1.1) includes 4 sub objective functions \{f_1, f_2, f_3, f_4\} which describes the objectives. The objective set $X$ is given as \{x_1, x_2, x_3, x_4\}.

$$
\text{Objective function } Z = \{f_1, f_2, f_3, f_4\}
$$

(1.1)

Where

The objective function ($f_i$) for memory usage ($x_i$) is to select test cases from the test suite $T$ which satisfies Equation (1.2)

$$
\min f_i (x_i, T), \quad x_i = \text{mu}_i - \text{mf}_i \quad \text{for all } i = 1 \text{ to } n
$$

(1.2)

where $\text{mu}_i$ memory used after the execution of $i^{th}$ test case

$\text{mf}_i$ memory free before the execution of $i^{th}$ test case
getRuntime returns the runtime object associated with the current Java application. Most of the methods of class Runtime are instance methods and must be invoked with respect to the current runtime object. The methods used for memory calculation are `Runtime.getRuntime().totalMemory()` and `Runtime.getRuntime().freeMemory()` in Java.

```java
long totalMemory = Runtime.getRuntime().totalMemory();
long freeMemory = Runtime.getRuntime().freeMemory();
```

The `Runtime.getRuntime().totalMemory()` returns the total amount of memory in Java Virtual Machine (JVM) that is the total amount of memory currently available for current and future objects, measured in bytes. The `Runtime.getRuntime().freeMemory()` returns the total amount of free memory in the JVM. The Memory utilised is calculated as the difference between memory used after the execution of the \(i^{th}\) test case and memory free before the execution of the \(i^{th}\) test case.

The objective function \(f_2\) for execution Time \(x_2\) is to select test cases from the test suite \(T\) which satisfies Equation (1.3)

\[
\min f_2(x_2, T), \quad x_2 = ft_i - st_i \quad \text{for all } i = 1 \text{ to } n \quad (1.3)
\]

where \(ft_i\) is the finish time of the \(i^{th}\) test case

\(st_i\) is the start time of the \(i^{th}\) test case

System.currentTimeMillis() is used for getting current time in milliseconds. Two Variables `startTime` and `endTime` are used. Before executing the test case, the current time is stored in a variable `startTime`. After execution, the current time is stored in `endTime`. Then the Execution Time is calculated as the difference between `endTime` and `startTime`. 
The objective function \( f_3 \) for Completeness \( x_3 \) is to select test cases from the test suite \( T \) which satisfies Equation (1.4)

\[
\max f_3(x_3, T), \quad x_3 = sc_i + pc_i + bc_i + fc_i \quad \text{for all } i = 1 \text{ to } n \quad (1.4)
\]

where
- \( sc_i \) is the statement coverage value of test case \( t_i \)
- \( pc_i \) is the path coverage value of test case \( t_i \)
- \( bc_i \) is the branch coverage value of test case \( t_i \)
- \( fc_i \) is the function coverage value of test case \( t_i \)

Completeness means the coverage ability of the test case. Test coverage is expressed as the extent to which testing covers the product’s complete functionality. This metric is an indication of the completeness of the testing. Coverage includes code coverage (statement, branch, path) and function coverage.

For calculating all these coverages (statement, branch, path and function coverage) a Control Flow Graph (CFG) is used. A control flow graph (CFG) is used to represent a program. CFG is a directed graph, where each node represents a statement, each edge represents the control flow and branches occur at conditional statements. Every execution of a program corresponds to a path in the control-flow graph, known as the execution path, from the entry node to the exit node. The entry node represents the entry point of the program and is represented with no inbound edges, and the exit node represents the point of program termination and is represented with no outbound edges. Decision node is a node with more than one outgoing edge, Junction node is a node with more than one incoming edge and the processing node is a node with one incoming and one outgoing edge.
Statement Coverage Criterion (Hong Zhu et al 1997) is a set $P$ of execution paths satisfies the statement coverage criterion if and only if for all nodes $n$ in the flow graph, there is at least one path $p$ in $P$ such that node $n$ is on the path $p$.

Statement coverage means the coverage of all statements in the program that is all nodes in CFG and all are executed at least once by a single test case or by suitable combination of test cases. The 100% statement coverage of the program should be achieved by the suitable combination of test cases which is important in statement coverage.

Branch Coverage Criterion (Hong Zhu et al 1997) is a set $P$ of execution paths satisfies the branch coverage criterion if, and only if, for all edges $e$ in the flow graph, there is at least one path $p$ in $P$ such that $p$ contains the edge $e$.

Branch coverage which is also known as decision coverage requires that all branches of the program that is all edges in CFG are to be traversed at least once. The 100% branch coverage of the program can be achieved by the execution of a single test case or by the suitable combination of test cases.

Branch coverage is better than statement coverage. Therefore, if a test set satisfies the branch coverage criterion then it also satisfies the statement coverage criterion.

Path Coverage Criterion (Hong Zhu et al 1997) is a set $P$ of execution paths satisfies the path coverage criterion if, and only if, $P$ contains all execution paths from the begin node to the end node in the flow graph.
A path (IEEE 1990) is a sequence of instructions that may be performed in the execution of a computer program. It usually represents the flow of execution from the start of a program to its end. Path coverage is a reliable technique used to check if every possible path through a given part of the code has been executed which is usually achieved by proper statement coverage and branch coverage. Path coverage is better than branch coverage, since, in the former all possible execution paths can be explored.

Function coverage checks whether each function or procedure in a program is invoked or not and it is checked by checking whether every possible call and return of the function has been executed.

The objective function \( f_i \) for fault coverage \( (x_4) \) is to select test cases from the test suite \( T \) which satisfies Equation (1.5)

\[
\max f_i(x_4, T), x_4 = \frac{fd_i}{N} \text{ for all } i = 1 \text{ to } n
\]

where \( fd_i \) is the faults detected by the \( i^{\text{th}} \) test case

\( N \) is the total number of faults in the system.

The number of faults covered by \( n^{\text{th}} \) test case is expressed as faults detected by \( n^{\text{th}} \) test case per total system faults. Fault coverage refers to the percentage of fault that can be detected during the test of any system.

If there are \( N \) test cases \( T = \{t_1, t_2, t_3, t_4, \ldots, t_n\} \) then each of the objective in the objective set \( X \) will have the second dimension which is given as \( \{x_1, x_2, x_3, x_4\} \) will have \( n \) values. So, the objective set can be rewritten as for the sake of coding as \( X = \{x_{1j}, x_{2j}, x_{3j}, x_{4j}\} \) where \( 1 \leq j \leq n \).
In case of the third objective $x_{3j}$ the third dimension is introduced since $x_{3j}$ is expressed as the combination of the statement coverage value of test case $t_i$, the path coverage value of test case $t_i$, the branch coverage value of test case $t_i$ and the function coverage value of test case $t_i$. Then the third objective can be expressed as $x_{3jk}$ where $1 \leq j \leq n$ and $1 \leq k \leq n$.

In general, the objective set $X$ can be expressed as $\{x_{1j}, x_{2j}, x_{3k}, x_{4j}\}$ where $1 \leq j \leq n$ and $1 \leq k \leq n$.

After forming the objective function the test case prioritization is done using the proposed integrated method.

The next chapter covers the detailed literature survey done on the research area.