Analysis of Test Case Prioritization in Regression Testing Using Genetic Algorithm

Pradiptakumar Mishra\textsuperscript{1} and B.K.S.S Pattanaik\textsuperscript{2}

\textsuperscript{1}CUTM, Bhubaneswar, Odisha, India.
\textsuperscript{2}GITA, Bhubaneswar, Odisha, India.

\begin{abstract}
Testing is an accepted technique for improving the quality of developed software. With the increase in size and complexity of modern software products, the importance of testing is rapidly growing. Regression testing plays a vital role for software maintenance when software is modified. The main purpose of regression testing is to ensure the bugs are fixed and the new functionality that are incorporated in a new version of software do not unfavorably affect the correct functionality of the previous version. So to revalidate the modified software, regression testing is the right testing process. Though it is an expensive process which requires executing maintenance process frequently but it becomes necessary for subsequent version of test suites. To evaluate the quality of test cases which are used to test a program, testing requires execution of a program. In this paper we propose a new test case prioritization technique using genetic algorithm. The proposed technique separate subsequences of the original test suite so that the new suite, which is run within a time-constrained execution environment, will have a superior rate of fault detection when compared to rates of randomly prioritized test suites. This experiment analyzes the genetic algorithm with regard to effectiveness and time overhead by utilizing structurally-based criterion to prioritize test cases.
\end{abstract}

\section{Introduction}

It is a basic phenomenon that, changes in software is one of the most foreseeable part when computer based system are made. So often it has been observed that the functionalities which were working in the previous version are still working in the new version also. With the existing errors and necessity for changing requirements propels the software to be reworked. Through the new uses of the previous version software, one can obtain new functionality that is not originally conceived in the requirements. To properly manage these changing concepts play a crucial role in the enduring efficacy of the software.

The basic purpose of regression testing is to revalidate the old functionality that was inherited from the previous version. Once the new functionality is added to the system then it can be accommodated by the distinctive software development process. The new version is required to behave exactly same in the previous version except the new included part. In other way, regression tests for a system may be perceived as partial operational requirements intended for the new version of a system.

Let’s take an example of a sequence of time intervals during the life cycle of software which is depicted in the Fig. 1. It is already a known fact that the regression testing intervals take up a considerable portion of the system’s life time for which the importance of regression testing can not be overlooked. Due to time constraint a complete regression testing can not be always possible for an updated version of the software which may result in the get away of costly, in appropriate changes into the field. Whatever it may be in the software revalidation process one can not completely omit or randomly reduce the regression testing interval.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{Fig_1.png}
\caption{Sequence of time intervals during a software’s life cycle}
\end{figure}

When a software/program is modified, we should not only look at whether the modifications work properly or not but it should be checked whether there have been any adverse side effect in the unmodified parts of the software/program because even if a small change in one part of a program can affect with the unrelated part of the program. It may happen the modified program may yield correct results on specially designed test cases for modification testing whereas it may produce incorrect results on other test cases on which the original program produced right outputs. To ensure the modified program’s effectiveness, the modified program is executed on all existing regression tests to ascertain that it still works the same way as the original program, except the modified parts of the program where the change is expected. Like other application programs, object oriented application programs also requires testing. Therefore, for proper testing of object oriented software, specific type of testing classes is required.
In class testing technique, in the first step, a sequence of methods with varying orders are invoked. Then after each sequence, it is verified whether the resulting state of the objects manipulated by the method is correct or not as proposed in [10],[11],[12]. By utilizing a distinctive approach for performing class test, a test driver is used which invokes a sequence of methods. The test driver performs its task in different steps. In the first step, the driver performs setup tasks which is comprised of tasks like constructor routines calling, as well as different initialization methods are utilized also. In the second step, the sequence of methods under test is invoked by the test driver. At the end, a special method called oracle method is invoked by the test driver to ascertain whether objects have attained proper states or not. As specified in [11],[13] when a class is modified it is required to retest the class itself as well classes derived by that class. As found in [13], [14] a function which has been effectively tested in segregation, may not be sufficiently tested in combing with other functions for which it is suggested to test the application programs which use the modified class. But reality is that, it is impossible to retest all such application programs. Therefore the testers should be well aware about the bottleneck. To overcome this problem, the testers can able to reduce the associated risks by regression testing those application programs which test cost will be economical.

In the last two decades solutions to these problems for traditional programs have been proposed by researchers Fischer, Harrold, Laski, Leung, Prathar, and white in [15], [1],[16],[29],[42],[18],[28] respectively. However, less attention had given to testing of OO programs and regression testing of object-oriented programs has not considered at all. Many new concepts like inheritance, encapsulation, polymorphism and dynamic binding are introduced by object-oriented model for software development process. Due to these new concepts a complex relationship occurs in between classes and their members.

As a result of which new testing problems find their place in the software field which are acknowledged by the different researchers as proposed in [24], [43], [38], [35]. Along with that it also raises a tough challenge for conducting regression testing for object-oriented programs. Although the existing results can be functional to regression testing of member functions of a class at the unit and integration levels, they lack suitability for testing object oriented components at higher levels, like a class, a group of classes, or class libraries. Basically the traditional approaches lack three measure difficulties while handling object-oriented features.

i) The complex relationships and dependencies, such as inheritance, aggregation, and association properties which exist between different classes cannot be addressed by traditional approaches.

ii) As most traditional approaches are control flow model oriented whereas class models exhibit state dependant behavior which has various changing nature for which traditional approaches cannot be applied to the class testing.

iii) Traditional approaches use test stubs to simulate the modules that are invoked but in OO programs this is difficult and costly because it requires understanding of many related classes, their member functions, and how the member functions are invoked in [38] and [27].

The organization of this paper is as follows. In section II, we have specified the existing regression testing technique. In section III, briefly identified some importance about previous related work. In section IV we have describe about some available soft computing approach. In section V we elaborately discussed our proposed model prioritization technique using genetic algorithm. It followed with conclusion.

Existing test case prioritization Techniques

As the regression testing is quite expensive, for reducing the cost, researchers have done many works other than test selection technologies. The test case prioritization technology is addressed in [25],[26],[21]. They prioritize the test cases according to certain measures. After which in the regression testing cycle, the test cases will be used to test the modified program P' in accordance with the same order, so that the “better” test cases can able to run first. The purpose of the prioritization is either to increase the rate of fault detection or, increase the rate of code coverage.

Here we have taken example-1 for better understanding about this technique.

Example 1

Let T1, T2, T3 and T4 are four test cases.

Suppose, T1 has the coverage of 60%
T2 has the coverage of 15%
T3 has the coverage of 35% and
T4 has the coverage of 45%

According to the second goal, by applying such technology, the test cases can be run in the order of T1, T4, T3, and T2. Like that according to the first goal, the order of the four test cases will depend on their ability to expose the fault.

As it is proposed in [30] test case prioritization techniques can be divided into three categories as depicted in Fig. 2.

Fig. 2. Phases of Prioritization Test Case Techniques

But there are 18 different test case prioritization techniques which are numbered from P1 to P18 in these three categories which are discussed below.

Comparator Techniques

Random ordering (P1): Here the test cases in test suite are randomly prioritized.

Optimal ordering (P2): In this case the test cases are prioritized to optimize rate of fault detection.

Statement level Techniques (Fine Granularity)

Total statement coverage prioritization (P3): Here, the test cases are prioritized in terms of total number of statements according to sorted order of coverage achieved. If test cases are having same number of statements they can be ordered pseudo randomly.

Additional statement coverage prioritization (P4): It is likely similar to total coverage prioritization which depends upon feedback about coverage attained to focus on statements that are not yet covered.

Total FEP prioritization (P5): Here, prioritization is done on the probability of exposing faults by test cases. Mutation analysis is used for approximation of the Fault-Exposing-Potential (FEP) of a test case. Additional FEP prioritization (P6): In this case, the total FEP prioritization is extensive to additional FEP prioritization due to the total statement coverage.
prioritization is also extended to additional statement coverage prioritization.

**Function level Techniques: (Coarse Granularity)**

Total function coverage prioritization (P7): Though it is similar to total statement coverage but here functions are used. As it has possessed coarse granularity level so the process of collecting function level traces in total statement coverage is cheaper as compared to the process of collecting statement level traces in total statement coverage.

Additional function coverage prioritization (P8): Though it is similar to total statement coverage prioritization with a slight difference i.e. it considers function level coverage instead of statements.

Total FEP prioritization (function level) (P9): It is equivalent to Total FEP prioritization with one difference that is instead of using statements, functions are used here.

Additional FEP prioritization (function level) (P10): This technique also similar to additional FEP prioritization here, instead of using statements it is using functions.

Total Fault Index (FI) prioritization (P11): In this technique a measurable software attribute called fault proneness is used.

Additional Fault Index prioritization (P12): Here, the total function coverage prioritization is extended to additional function coverage prioritization and the total Fault Index prioritization is extended to additional Fault Index prioritization similarly also.

Total Fault Index with FEP coverage prioritization (P13): It combines both total Fault Index and FEP coverage prioritization to achieve a superior rate of fault detection.

Additional Fault Index with FEP coverage prioritization (P14): Here also, the total function coverage prioritization is extended to additional function coverage prioritization as well as the total Fault Index with FEP coverage prioritization is extended to Additional Fault Index with FEP coverage prioritization in a similar manner.

Total Diff prioritization (P15): It is similar to Total Fault Index prioritization but here, the total FI prioritization requires collection of metrics whereas total Diff prioritization requires only the calculation of syntactic differences between the program and the modified program. Diff means that merely syntactic differences are given consideration.

Additional Diff prioritization P16: Here, the total Diff prioritization is extended to additional Diff prioritization in the same manner as the total. As a result of which new testing problems find their place in the software field which are acknowledged by the different researchers as proposed in [24], [43], [38], [35]. Along with that it also raises a tough challenge for conducting regression testing for object-oriented programs. Although the existing results can be functional to regression testing of member functions of a class at the unit and integration levels, they lack suitability for testing object oriented components at higher levels, like a class, a group of classes, or class libraries. Basically the traditional approaches lack three measure difficulties while handling object oriented features.

iv) The complex relationships and dependencies, such as inheritance, aggregation, and association properties which exist between different classes cannot be addressed by traditional approaches.

v) As most traditional approaches are control flow model oriented whereas class models exhibit state dependant behavior which has various changing nature for which traditional approaches cannot be applied to the class testing.

vi) Traditional approaches use test stubs to simulate the modules that are invoked but in OO programs this is difficult and costly because it requires understanding of many related classes, their member functions, and how the member functions are invoked in [38] and [27].

The organization of this paper is as follows. In section II, we have specified the existing regression testing technique. In section III, briefly identified some importance about previous related work. In section IV we have describe about some available soft computing approach. In section V we elaborately discussed our proposed model prioritization technique using genetic algorithm. It followed with conclusion.

**Existing test case prioritization Techniques**

As the regression testing is quite expensive, for reducing the cost, researchers have done many works other than test selection technologies. The test case prioritization technology is addressed in [25],[26],[21]. They prioritize the test cases according to certain measures. After which in the regression testing cycle, the test cases will be used to test the modified program P in accordance with the same order, so that the “better” test cases can able to run first. The purpose of the prioritization is either to increase the rate of fault detection or, increase the rate of code coverage.

Here we have taken example-1 for better understanding about this technique.

**Example 1**

Let T1, T2, T3 and T4 are four test cases.

Suppose, T1 has the coverage of 60%
T2 has the coverage of 15%
T3 has the coverage of 35% and
T4 has the coverage of 45%

According to the second goal, by applying such technology, the test cases can be run in the order of T1, T4, T3, and T2. Like that according to the first goal, the order of the four test cases will depend on their ability to expose the fault.

As it is proposed in [30] test case prioritization techniques can be divided into three categories as depicted in Fig. 2.

![Fig. 2. Phases of Prioritization Test Case Techniques](image)

But there are 18 different test case prioritizations techniques which are numbered from P1 to P18 in these three categories which are discussed below.

**Comparator Techniques**

Random ordering (P1): Here the test cases in test suite are randomly prioritized.

Optimal ordering (P2): In this case the test cases are prioritized to optimize rate of fault detection.

**Statement level Techniques (Fine Granularity)**

Total statement coverage prioritization (P3): Here, the test cases are prioritized in terms of total number of statements according to sorted order of coverage achieved. If test cases are having same number of statements they can be ordered pseudo randomly.

Additional statement coverage prioritization (P4): It is likely similar to total coverage prioritization which depends
upon feedback about coverage attained to focus on statements that are not yet covered.

Total FEP prioritization (P5): Here, prioritization is done on the probability of exposing faults by test cases. Mutation analysis is used for approximation of the Fault-Exposing-Potential (FEP) of a test case. Additional FEP prioritization (P6): In this case, the total FEP prioritization is extensive to additional FEP prioritization due to the total statement coverage prioritization is also extended to additional statement coverage prioritization.

Function level Techniques: (Coarse Granularity)

Total function coverage prioritization (P7): Though it is similar to total statement coverage but here functions are used. As it has possessed coarse granularity level so the process of collecting function level traces is cheaper as compared to the process of collecting statement level traces in total statement coverage.

Additional function coverage prioritization (P8): Though it is similar to additional statement coverage prioritization with a slight difference i.e. it considers function level coverage instead of statements.

Total FEP prioritization (function level) (P9): It is equivalent to Total FEP prioritization with one difference that is instead of using statements, functions are used here.

Additional FEP prioritization (function level) (P10): This technique also similar to additional FEP prioritization here, instead of using statements it is using functions.

Total Fault Index (FI) prioritization (P11): In this technique a measurable software attribute called fault proneness is used.

Additional Fault Index prioritization (P12): Here, the total function coverage prioritization is extended to additional function coverage prioritization and the total Fault Index prioritization is extended to additional Fault Index prioritization similarly also.

Total Fault Index with FEP coverage prioritization (P13): It combines both total Fault Index and FEP coverage prioritization to achieve a superior rate of fault detection.

Additional Fault Index with FEP coverage prioritization (P14): Here also, the total function coverage prioritization is extended to additional function coverage prioritization as well as the total Fault Index with FEP coverage prioritization is extended to Additional Fault Index with FEP coverage prioritization in a similar manner.

Total Diff prioritization (P15): It is similar to Total Fault Index prioritization but here, the total FI prioritization requires collection of metrics whereas total Diff prioritization requires only the calculation of syntactic differences between the program and the modified program. Diff means that merely syntactic differences are given consideration.

Additional Diff prioritization (P16): Here, the total Diff prioritization is extended to additional Diff prioritization in the same manner as the total

1. Identify the execution time of each test case.
2. Find out code coverage of test case to be prioritized.
3. Generate using GA the remaining test case after severe test case have been detected.

To practically implement our algorithm we have configure JUnit, Ant and Eclipse Emma to work with Eclipse editor. First the execution time of all the test case are determined by running all the test case using ant make file in JUnit environment. Eclipse Emma is used to find out code coverage of each test case. The summary of all test case is given in table 1.

Once these value are determined next we implement the algorithm using MetLab to find out remaining test case for the test suit after severe test case is added. Below is the metlab program code of our implementation. In metlab implementation we take combination of test case as initial population. Calculate fitness value for each population, do crossover and mutation with given mutation probability. This sequence is repeated for given number of generation. A Test case number, time of execution, code coverage derived by JUnit and Eclipse Emma taken for GA implementation is shown in table I.

Table I

<table>
<thead>
<tr>
<th>Test case number, time of execution, code coverage derived by JUnit and Eclipse Emma taken for GA implementation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

After running the metlab implementations with the detail given in Fig. 7 it gives the following sequence of test case as prioritized test case and the graph is given in Fig 8.

**Fig. 9. Prioritize test suite generated by our algorithm**

Experiments and Results

Experiments are run in order to analyze (i) the effectiveness and the efficiency of the genetic algorithm. In order to compare this test suit with

**Fig. 9. Graph showing generation and best test suit**
other available prioritization technique we choose “random test case prioritization”. we do an experiment by randomly taking test case from the test case given in Fig 10.

**Random test cases:**

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Fig. 10. Prioritize test suite generated by our algorithm

The experiment show that test case generated by GA takes less time as compared to test case generated by random test, goal oriented and path oriented test case prioritization, which is depicted in Fig 11, Fig 12 and Fig 13.

![Fig. 11. Time of execution comparison between random test case prioritization and our algorithm](image1.png)

![Fig. 12. Time of execution comparison between goal oriented case prioritization and our algorithm](image2.png)

![Fig. 13. Time of execution comparison between path oriented case prioritization and our algorithm](image3.png)

Fig. 14. Code coverage comparison between random test case prioritization and our algorithm

Conclusions

In this paper we have describe to generate test case which are severe as per customer, takes less time and cover more code. We have compared test case generated by GA with test case generated by goal oriented, random and path oriented test prioritization technique and found some better result in terms time and code coverage.

References


[34]. N. Wilde and R. Huitt. Issues in the maintenance of object-oriented programs.” University of West Florida and Bell Communications Research, 1991.